

10 HERENCIA

A lo largo de los capítulos anteriores se ha insistido en que la programación orientada a objetos exalta la reutilización. Pero la reutilización no debe basarse en copiar texto de código. Se reutiliza cuando se echa mano a una de las tantas clases ya compiladas y probadas que se pueden encontrar en los ensamblados que ya vienen con el propio Framework .NET o en ensamblados creados por otros programadores. En la tecnología .NET, un ensamblado generado por el compilador de un lenguaje (que no necesariamente tiene que ser C#) puede ser utilizado incluso desde otro lenguaje diferente.

Aunque por ejercitación y entrenamiento es posible que Ud. haya desarrollado clases que ya estaban disponibles, en la práctica profesional real si no se tiene disponible ninguna clase que se ajuste a lo que se necesita, entonces se definen nuevas clases basándose en los tipos primitivos o en otras clases previamente definidas y probadas por otros. Así por ejemplo, se han visto ejemplos que utilizan la clase `Stopwatch` o la clase `Fecha` y por supuesto clases predefinidas en .NET como `String`. Esto se denomina *composición*, porque los objetos de las nuevas clases están compuestos por objetos instancias de las clases ya existentes. En este proceso de diseño e implementación de clases se ha insistido en ocultar información, es decir, las clases deben mostrar sólo aquellos métodos y propiedades que las hacen útiles a las demás (su interfaz) y ocultar todo lo solo sea necesario para su implementación, cuidando que toda modificación de alguna variable de instancia (su "estado interno del objeto") se haga siempre a través de los métodos y propiedades que la propia clase ofrece para ello. Esto es primordial en el éxito de la orientación a objetos: la clase es buen recurso para el modelado y a la vez un artefacto lo suficientemente cerrado como para hacer más fácil, confiable, reutilizable y seguro el proceso de construcción de software. Este y otros principios básicos del paradigma orientado a objetos se abordan con más detalle en el Capítulo 16.

Pero el propio mundo que se modela con las clases es cambiante, a la vez que los humanos que desarrollamos software no somos infalibles. Ocurre entonces que las clases de las que propugnamos sus virtudes y bondades debieran poder estar sujetas a ampliaciones, correcciones, mejoras y adaptaciones. ¿Cómo lograr esto sin dar al traste con la confiabilidad, reutilización y seguridad que se logra por el encapsulamiento y por la aplicación del principio de ocultar información? Es decir, ¿cómo lograr el efecto de modificar o cambiar una clase sin tener que disponer del texto o código escrito en el lenguaje del que se generó la misma?

La **herencia** es la respuesta a estas preguntas. Mediante la herencia se podrá definir una nueva clase a partir de una existente como si se estuviera adaptando o modificando la clase existente. Con la herencia se logra el efecto de "añadir" o "adaptar" el código de una clase ya existente, pero sin tener que disponer del código de esta y mucho menos tener que modificarlo. Es decir, sin correr el riesgo de afectar la funcionalidad que ya existe y que puede seguir siendo útil para las otras clases que la están utilizando.

Por medio de la herencia el programador "no toca" ni "tiene que ver las interioridades" del código de la clase existente, es el compilador quien hace el trabajo de buscar en el ensamblado donde está la clase original, todo lo que necesita reutilizar.

La herencia se considera como el rasgo o recurso más "distintivo" de la programación orientada a objetos. La herencia trae consigo varios beneficios porque permite expresar jerarquías de tipos que favorecen el modelado y el diseño, y además propicia dar flexibilidad al código mediante el polimorfismo y el principio de sustitución. Todos estos temas serán estudiados en el presente capítulo.

10.1 Herencia Por Ampliación

Tomemos de nuevo la clase `Cuenta` que se introdujo en el Capítulo 6. Se analizó implementación de `Cuenta` basada en la información y funcionalidades básicas de cuenta bancaria. El `public class Cuenta` {

```
public string Titular { get; private set; }
public float Saldo { get; private set; }

public Cuenta(string titular, float saldoInicial) {
    Titular = titular;
    if (saldoInicial < 0)
        throw new Exception("Saldo inicial debe ser mayor que 0");
    Saldo = saldoInicial;
}

public void Deposita(float cantidad) {
    if (cantidad <= 0)
        throw new Exception("Cantidad a depositar debe ser mayor que cero");
    Saldo += cantidad;
}

public void Extrae(float cantidad) {
    if (cantidad <= 0)
        throw new Exception("Cantidad a extraer debe ser mayor que cero");
    else if (Saldo - cantidad < 0)
        throw new Exception("No hay suficiente saldo para extraer");
    Saldo -= cantidad;
}
}
```

Listado 10-1 muestra este código que define la clase `Cuenta`.

```
public class Cuenta {
    public string Titular { get; private set; }
    public float Saldo { get; private set; }

    public Cuenta(string titular, float saldoInicial) {
        Titular = titular;
        if (saldoInicial < 0)
            throw new Exception("Saldo inicial debe ser mayor que 0");
        Saldo = saldoInicial;
    }
}
```

```
public void Deposita(float cantidad) {
    if (cantidad <= 0)
        throw new Exception("Cantidad a depositar debe ser mayor que cero");
    Saldo += cantidad;
}

public void Extrae(float cantidad) {
    if (cantidad <= 0)
        throw new Exception("Cantidad a extraer debe ser mayor que cero");
    else if (Saldo - cantidad < 0)
        throw new Exception("No hay suficiente saldo para extraer");
    Saldo -= cantidad;
}
}
```

Listado 10-1. Clase Cuenta

Suponga que ahora se quiere poder transferir saldo a otra cuenta. Para esto se puede definir una clase `CuentaTransferencia` que tenga el mismo código de `Cuenta` y añadirle el método `Transfiere` (Listado 10-2)

```
public class CuentaTransferencia {
    public String Titular { get; private set; }
    public float Saldo { get; private set; }

    public CuentaTransferencia(string titular, float saldoInicial) {
        Titular = titular;
        if (saldoInicial < 0)
            throw new Exception("Hay que abrir cuenta con saldo > 0");
        Saldo = saldoInicial;
    }

    public void Deposita(float cantidad) {
        if (cantidad <= 0)
            throw new Exception("Cantidad a depositar debe ser > 0");
        Saldo += cantidad;
    }

    public void Extrae(float cantidad) {
        if (cantidad <= 0)
            throw new Exception("Cantidad a extraer debe ser > 0");
        else if (Saldo - cantidad < 0)
            throw new Exception("No hay suficiente saldo para extraer");
        Saldo -= cantidad;
    }

    public void Transfiere(float cantidad, Cuenta cuentaDestino) {
        if (cantidad <= 0)
```

```
        throw new Exception("Cantidad a transferir debe ser > 0");
    else if (Saldo >= cantidad) {
        cuentaDestino.Deposita(cantidad);
        Extrae(cantidad);
    }
    else
        throw new Exception("No hay saldo para hacer el traspaso");
    }
}
```

Listado 10-2. Clase Cuenta con método Transfiere

Este método **Transfiere** extrae la cantidad a transferir de la cuenta actual, si hay saldo disponible, y lo deposita en la cuenta destino.

Hay dos inconvenientes en esta vía de solución:

- Para poder añadir el nuevo método dentro de la clase **Cuenta** hay que tener el texto fuente de dicha clase. Esto no siempre es posible, más aún en la filosofía de .NET, en que el ensamblado en el que se encuentre la versión compilada de una clase puede haberse obtenido por la red desde otro confín del planeta¹.
- Aun cuando se disponga del texto fuente, si la inclusión de la nueva funcionalidad se obtiene, como en el caso anterior, por la vía de introducir modificaciones dentro del texto de la clase y volverla a compilar, entonces ¿qué garantías se le da a los viejos clientes de la clase de que esta se seguirá comportando como antes, pero ahora con la nueva funcionalidad? Es decir, ¿cómo se le puede asegurar a los programadores viejos clientes de la clase **Cuenta** que no tienen que hacer cambios en sus clases si ellos no están interesados en utilizar la nueva funcionalidad **Transfiere**?
- Puede pensarse que el método de *corta y pega* de segmentos de texto en otro (lo cual es algo que se puede hacer en prácticamente cualquier herramienta de desarrollo ligada al compilador²) permite hacer con facilidad y "seguridad" los cambios deseados en el texto viejo para obtener el texto nuevo. La clase vieja y la clase nueva tienen que estar entonces en espacios de nombre (*namespaces*) diferentes o si se quieren ubicar en el mismo espacio de nombres entonces habría que darle un nombre diferente a la nueva clase. De todos modos, el inconveniente mayor es que no hay nada que formalmente ligue a la nueva clase con la original. Es decir, en la ejecución de una aplicación un objeto instancia de esta nueva clase no podría sustituir a un objeto instancia de la clase original aunque aparentemente haga todo lo que la original hace.

¹ Esto no está en contradicción con la filosofía de código abierto (Open Source). Usted es libre de mostrar su código para compartir su conocimiento. Lo que se promueve aquí es que no tenga que ser una exigencia modificar el código fuente para reutilizarlo o añadir nuevas funcionalidades.

² Como es el caso de Visual Studio .NET

- Esto último, que se conoce como **Principio de Sustitución**³, es una cualidad importante del modelo orientado a objetos que se puede lograr con la herencia y que permite hacer software flexible y adaptable.

Los inconvenientes anteriores están relacionados con el principio de ocultar información. La solución se logra con **la herencia**. A través de la herencia se define una nueva clase que **extiende** o **amplía** la clase **Cuenta** **heredando** todo lo que ésta tiene y añadiendo lo nuevo pero **sin necesidad de tener que disponer del texto de la clase Cuenta y mucho menos tener que hacer cambios dentro de éste**.

Suponiendo que clase original **Cuenta** está en un espacio de nombres **Finanzas** y que al compilar la clase a continuación se dispone en el proyecto del ensamblado donde está el compilado de **Cuenta**, entonces se puede definir la nueva clase haciendo:

```

1. public class CuentaTransferencia : Cuenta {
2.     public CuentaTransferencia(string titular, float saldoInicial) :
3.         base(titular, saldoInicial) { }

4.     public void Transfiere(float cantidad, Cuenta cuentaDestino) {
5.         if (cantidad <= 0)
6.             throw new Exception("Cantidad a transferir debe ser > 0");
7.         else if (Saldo >= cantidad) {
8.             cuentaDestino.Deposita(cantidad);
9.             Extrae(cantidad);
10.        }
11.        else
12.            throw new Exception("No hay saldo para hacer el traspaso");
13.    }
14. }
```

Listado 10-3. Clase CuentaTransferencia con el método Transfiere

Observe que en esta nueva clase **¡sólo se ha incluido el código del nuevo método!**, inada de copia y pega ni reescritura de texto!

De modo que ahora podemos disponer de dos clases, la vieja clase **Cuenta** que no tiene el método **Transfiere** y otra clase **CuentaTransferencia** que incluye, por heredar de **Cuenta**, todos los métodos de **Cuenta** junto con el nuevo método **Transfiere**. Es decir, si **nuevaCuenta** es un objeto de tipo **CuentaTransferencia** se le pueden aplicar también, además del método **Transfiere** definido en el Listado 10-3, todos los métodos y propiedades que están definidos en **Cuenta** y que se comportarán como tales. De modo que es correcto hacer **nuevaCuenta.Deposita(200)**, **nuevaCuenta.Extrae(50)** y **nuevaCuenta.Saldo** al igual que **nuevaCuenta.Transfiere(100, otraCuenta)**. Pero si **viejaCuenta** es un objeto de tipo **Cuenta** no se puede hacer **viejaCuenta.Transfiere(100, otraCuenta)** (sería reportado error por el compilador), ya que la clase **Cuenta** no tiene ningún método **Transfiere**.

³ Este es uno de los principios SOLID que se estudian en el Capítulo 16

Note que la propia implementación (líneas 8 y 9) del método `Transfiere` hace uso de los métodos `Extrae` y `Deposita` definidos en la clase base.

Además del método `Transfiere` se ha añadido un constructor que recibe los mismos parámetros que la clase `Cuenta`, y que llama al constructor de la clase `Cuenta` mediante la construcción `base`. En el siguiente epígrafe se aborda el comportamiento de los constructores en relación con la herencia.

De modo que cada programador podrá decidir cuál es la clase de su conveniencia: si la nueva clase `CuentaTransferencia`, porque le interesa usar el nuevo método `Transfiere`, o la vieja clase `Cuenta` de la que tiene la garantía que no ha sido afectada.

La clase de la cual se hereda (`Cuenta`) se denomina **clase base** y la clase que extiende o hereda de esta clase base (`CuentaTransferencia`) se denomina **clase derivada**⁴. `Cuenta` se dice que es un supertipo de `CuentaTransferencia` y `CuentaTransferencia` es un subtipo de `Cuenta`.

Como se ha visto en este ejemplo, la notación sintáctica para definir una clase que hereda de otra es de la forma

```
class <nombre de la subclase>:<nombre de la superclase> {  
    <definición de los nuevos recursos a incluir en la subclase>  
}
```

10.2 Los Constructores En Las Subclases

Como se puede apreciar en el Listado 10-3, se ha añadido un nuevo constructor que no tiene código. Este nuevo constructor de la clase `CuentaTransferencia` se limita a llamar al constructor de la superclase `Cuenta`, que se encarga de inicializar las propiedades `Titular` y `Saldo`. De esta manera, la clase `CuentaTransferencia` **delega** en la clase `Cuenta` la inicialización de las variables y propiedades de la cuenta. Esta delegación ocurre al llamar al constructor de la clase base al hacer `base(titular, saldoInicial)`. En este caso, `CuentaTransferencia` no necesita inicializar variables o propiedades diferentes a las de `Cuenta` y es por eso que el constructor de la clase heredera no tiene en este caso ningún código.

Aun cuando el constructor de la clase heredera no tiene código, si se omite el constructor, el intento de compilar la clase `CuentaTransferencia` será reportado error de compilación ¡porque `CuentaTransferencia` está obligada a utilizar el constructor de la clase base `Cuenta`!

Esta restricción tiene un objetivo: si el programador ha decidido definir una clase `CuentaTransferencia` derivada de `Cuenta` es porque quiere reutilizar en `CuentaTransferencia` los recursos de `Cuenta`. Por lo tanto, si `Cuenta` tiene un constructor, sería lógico que para crear un objeto de tipo `CuentaTransferencia` haya que pasar al menos por el proceso de construcción de una `Cuenta`.

⁴ También denominadas respectivamente *superclase*, *clase ancestro* o *clase padre*, y *subclase*, *clase heredera* o *clase descendiente*. En esto los autores y los diseñadores de lenguajes no han adoptado una terminología única, tal vez en un intento superficial de diferenciarse.

Es decir, la clase derivada **está obligada a llamar a un constructor de la clase base**. Si la clase base tuviese, entre otros posibles, un constructor sin parámetros y un constructor de la clase derivada no hace una llamada explícita de la forma `base(...)`, entonces al crear un objeto del tipo de la clase derivada se llama implícitamente al constructor sin parámetros de la clase base.

Si ahora a partir de la clase `Cuenta` se define una clase derivada `CuentaConjunta` (Listado 10-4) que tiene dos titulares, el constructor de esta clase puede definirse como se indica en la línea 3. Note que lo que interesa aquí es hacer todo lo mismo que hace el constructor de la clase base (línea 4) y además añadir lo propio de la clase derivada, que es en este caso añadir el otro titular de la cuenta (línea 5). Aunque no puede generalizarse y afirmar que toda clase derivada siga este mismo patrón, este no deja de ser un escenario frecuente: **si la clase derivada amplía las capacidades de la clase base, entonces para crear un objeto de esta "clase ampliada" es lógico hacer lo mismo que para crear un objeto de la clase base, y posiblemente algo más, ya que el nuevo objeto debe ser más complejo**.

Note que en este caso el constructor de la clase derivada tiene que incluir tres parámetros: dos parámetros que son los mismos que los de la clase base y un tercer parámetro, en este caso para indicar el otro titular.

```
1. class CuentaConjunta : Cuenta {
2.     public string CoTitular { get; private set; }
3.     public CuentaConjunta(string tit1, string tit2, int depositoInicial) :
4.         base(tit1, depositoInicial) {
5.         CoTitular = tit2;
6.     }
7. }
```

Listado 10-4. Clase `CuentaConjunta` heredera de `Cuenta`

10.3 Herencia Por Especialización

Una clase también puede *especializar* (*cambiar, modificar, adaptar*) el comportamiento de la clase base.

Para ilustrar mejor esto, se definirá una clase `CuentaCredito` que permita extraer aun cuando no haya suficiente saldo en la cuenta. Para ello se quisiera "modificar" el comportamiento del método `Extrae` como se muestra en el Listado 10-5.

```
1. public class CuentaCredito : Cuenta {
2.     public CuentaCredito(string titular, float saldoInicial) :
3.         base(titular, saldoInicial) { }
4.
5.     public new void Extrae(float cantidad) {
6.         if (cantidad <= 0)
7.             throw new ArgumentException("Cantidad a extraer debe > 0");
8.         else if (Saldo >= cantidad)
9.             Saldo -= cantidad;
```

```

10.         else //No hay saldo. Se va extraer a crédito con 5% de interés
11.             Saldo -= (cantidad + (cantidad * 5 / 100));
12.         }
13.     }

```

Listado 10-5. Clase CuentaCredito

En la nueva implementación de `Extrae`, si no existe suficiente saldo en la cuenta, se extrae pero con un cargo adicional del 5% de la cantidad que se pide extraer. En un escenario más realista esto requiere de operaciones más complejas que aquí se han obviado en beneficio de la simplicidad.⁵ Por ejemplo, intente el lector añadir a la clase `CuentaCredito` el código necesario para limitar la cantidad a extraer bajo crédito.

Algunos aspectos característicos de cuando se hereda para especializar o adaptar un comportamiento se pueden observar en el código anterior:

- La nueva definición del método va precedida de la palabra `new`⁶ (línea 5).
- La nueva implementación del método `Extrae` modifica la propiedad `Saldo` definida en la clase `Cuenta` (línea 9 y 11).

De acuerdo con la definición de la clase `Cuenta`, si se hace

```

Cuenta cuenta = new Cuenta("Juan", 65);
cuenta.Extrae(120); // Error al ejecutar, no hay saldo suficiente

```

Al ejecutar este código se ejecuta el método `Extrae` definido en la clase `Cuenta`, que produce un error al intentar extraer una cantidad mayor de la que se dispone en el saldo. Por otra parte, si se tiene el siguiente código

```

CuentaCredito cuentaCredito = new CuentaCredito("Pedro", 100);
cuentaCredito.Extrae(200); // Se extrae aplicando 5% de interés
Console.WriteLine(cuentaCredito.Saldo); // Saldo = -110

```

Al intentar extraer de `CuentaCredito` una cantidad mayor que la disponible en el saldo no se produce error, sino que se ejecuta el nuevo método `Extrae` definido en la clase `CuentaCredito` que aplica un interés del 5% sobre la cantidad extraída, quedando el saldo de la cuenta negativo, lo cual indica que el titular debe dinero al banco.

En muchos casos cuando se hereda, ya sea para ampliar o para especializar la funcionalidad de una clase, se necesita acceder a las "interioridades" de la clase base. Es decir, a miembros de la clase base que no forman parte de su "interfaz" pública. En la clase `Cuenta` (Listado 10-1), la parte `set` de la propiedad `Saldo` se ha definido `private`, para evitar que quien usa la cuenta pueda modificar el saldo a su antojo. En lugar de permitir modificar el saldo directamente, se han añadido los métodos `Deposita` y `Extrae` que regulan la lógica para incrementar o disminuir el valor del saldo. Por esta razón el compilador de C# reportaría error al tratar de compilar el código de la clase `CuentaCredito`.

⁵ Esto solo es un ejemplo simplificado, no se pretende emular el software de Visa o MasterCard. ©

⁶ Esta utilización de la palabra reservada `new` no guarda ninguna relación con la utilización de `new` para crear un objeto.

Pero para redefinir el funcionamiento del método `Extrae` en `CuentaCredito` se necesita modificar directamente la propiedad `Saldo`, y esto implica utilizar la parte `set` de la propiedad. ¿Cómo lograr esto sin definir la parte `set` como `public` y comprometer con ello el funcionamiento de la clase `Cuenta` y la de sus herederos?

10.4 Visibilidad Con La Herencia. La Especificación Protected

En correspondencia con el principio de ocultar información que defiende la POO, la parte `set` de las propiedades `Titular` y `Saldo` se definieron `private` y por tanto solo pueden ser utilizadas desde la propia clase `Cuenta`. De modo que desde cualquier otra clase que utilice una variable `c` de tipo `Cuenta` no se puede hacer `c.Saldo = 1200` o `c.Titular = "Juan"`. Sin embargo, en la especialización del método `Extrae` de la clase `CuentaCredito` se utiliza directamente la propiedad `Saldo` (Listado 10-5).

Según las reglas de visibilidad de C#, el compilador reportaría un error de compilación al intentar compilar el código de la clase `CuentaCredito` porque el método `Extrae` intenta modificar la propiedad `Saldo`, lo cual no está permitido por ser `private`. La solución que ofrece C# para esto es una nueva especificación de visibilidad relacionada con una clase y las clases que heredan o se derivan de ésta. Esta especificación de visibilidad se logra a través de la palabra reservada `protected`. Las variables, métodos o propiedades definidos como `protected` (protegidos) no son visibles para las clases clientes (estén o no en el mismo ensamblado), pero sí son visibles para las clases herederas o derivadas (aunque estén en otro ensamblado).

De modo que para que el método `Extrae` de la subclase `CuentaCredito` pudiese modificar el saldo, en la clase `Cuenta` la parte `set` de la propiedad `Saldo` debió definirse `protected` (Listado 10-6).

```
public class Cuenta {  
    public string Titular { get; private set; }  
    public float Saldo { get; protected set; }  
  
    // Demás métodos de la clase Cuenta  
}
```

Listado 10-6. Clase `Cuenta` con `Saldo` `protected`

A los efectos de una clase cliente `A` que use a la clase `Cuenta` estas variables son privadas, pero no a los efectos de una clase que extienda o herede de la clase `Cuenta`

```
class A {  
    //...  
    void HazmeMillonario(Cuenta c) {  
        c.Saldo = 5000000; // Error de compilación  
    }  
}
```

De modo que C# permite establecer una diferenciación entre la visibilidad para las clases clientes y la visibilidad para las subclases. Si se quiere que un recurso (variable de instancia, propiedad o método) sea privado, pero pudiera ser de interés a una clase extendida, entonces este debe definirse como `protected`.



Se le pudiera criticar a este enfoque que requiere cierta previsión en el momento de definir una clase, pues hay que prever sus posibles necesidades futuras de ampliación (lo que se conoce como Principio de la Clarividencia). En todo caso, lo más razonable sería hacer el razonamiento inverso, es decir, si estamos seguros que ningún descendiente tiene por qué usar una variable, propiedad o método entonces se debe especificar `private`.

10.5 El Principio de Sustitución y el Polimorfismo

La herencia establece una **jerarquía de tipos** bajo la que todo objeto del tipo de una clase derivada se puede asignar a una variable o a un parámetro que sea del tipo de la clase base. Se dice que puede sustituir a un objeto de la clase base. Es decir, si `cuenta` y `cuentaCredito` son variables definidas de la siguiente manera

```
Cuenta cuenta = new Cuenta("Juan", 65);
CuentaCredito cuentaCredito = new CuentaCredito("Pedro", 100);
```

La siguiente asignación es permitida

```
cuenta = cuentaCredito;
```

Lo que significa que ahora `cuenta` y `cuentaCredito` se refieren al mismo objeto, tal y como se muestra en la Figura 10-1. Es decir, que ambas variables hacen referencia a la cuenta de Pedro con un saldo igual a 100.

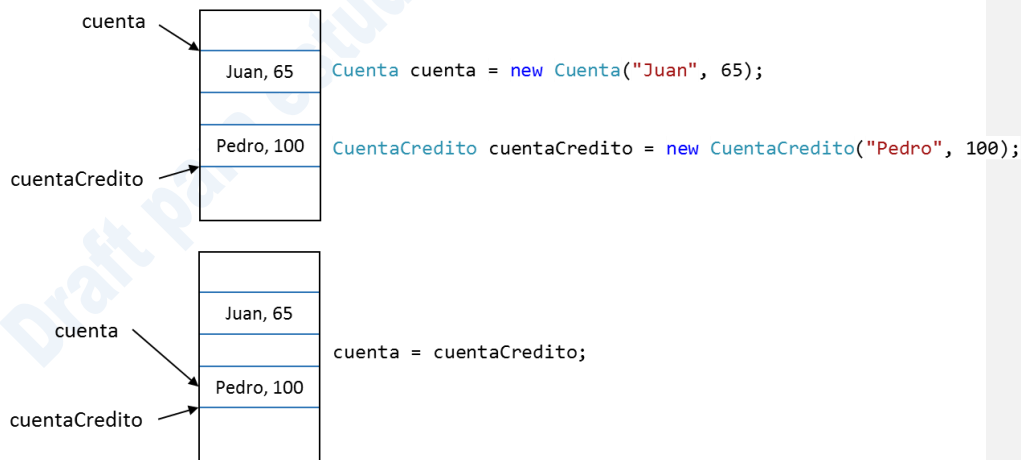


Figura 10-1. Polimorfismo y objetos en memoria

Sin embargo, si después de hacer esta asignación se hiciera `cuenta.Extrae(120);`

¿Qué ocurriría? La respuesta es que se produciría un error pues la cuenta no tiene suficiente saldo. Es decir, aunque `cuenta` y `cuentaCredito` se están refiriendo la misma cuenta de Pedro, de tipo `CuentaCredito`, al hacer `cuenta.Extrae(120)` se aplica el método que está definido en `Cuenta` y no el nuevo definido en `CuentaCredito`. Pero si se hace

```
cuentaCredito.Extrae(120);
```

Entonces se extrae sin producir error aplicando un interés, que es lo esperado en este caso ya que la nueva definición de `Extrae` en `CuentaCredito` oculta la que se hubiese heredado de `Cuenta`.

¿Cómo lograr que los objetos de una clase derivada conserven su "personalidad" cuando se utilicen en lugar de un objeto de la clase base? Lo que significaría en un ejemplo como el anterior que siempre pudiera extraerse de un objeto de tipo `CuentaCredito`, con independencia del tipo de la variable que la referencia. La solución que ofrece C# a esto se analiza en el subepígrafe a continuación.

10.5.1 Las especificaciones virtual y override

Una clase puede especificar que un método es `virtual`; esto quiere decir que una clase derivada puede hacer entonces una redefinición del método que **suplante** la definición de la clase base. De esta forma, cuando el objeto del tipo de la clase derivada se esté utilizando a través de una variable del tipo de la clase base seguirá aplicando los métodos redefinidos en la clase derivada. Para lograr el efecto deseado al extraer de una cuenta, y que siempre se ejecute el método redefinido, y no el de la clase base, el método `Extrae` en la clase `Cuenta` debe especificarse como `virtual` (Listado 10-7).

```
public class Cuenta {
    public virtual void Extrae(float cantidad) {
        if (cantidad <= 0)
            throw new Exception("Cantidad a extraer debe ser mayor que cero");
        else if (Saldo - cantidad < 0)
            throw new Exception("No hay suficiente saldo para extraer");
        Saldo -= cantidad;
    }
    // Otros métodos y propiedades de la clase Cuenta
}
```

Listado 10-7. Método `Extrae` en la clase `Cuenta` con el modificador `virtual`

Ahora al implementar el método `Extrae` en la clase `CuentaCredito` debe indicarse que se trata de una redefinición del método heredado mediante la palabra reservada `override` (Listado 10-8).

```
public class CuentaCredito : Cuenta {
    public override void Extrae(float cantidad) {
        if (cantidad <= 0)
            throw new Exception("Cantidad a extraer debe ser mayor que cero");
        else if (Saldo >= cantidad)
            Saldo -= cantidad;
        else //No hay saldo suficiente se va extraer a crédito con 5% de interés
            Saldo -= (cantidad + (cantidad * 5 / 100));
    }
}
```

```

    }
    // Otros métodos y propiedades de la clase CuentaCredito
}

```

Listado 10-8. Método Extrae en la clase CuentaCredito con el modificador override

Suponga que utilizando estas dos clases se ejecuta el siguiente segmento de código

```

Cuenta c;
c = new CuentaCredito("Pedro", 100);
c.Extrae(200); // Se ejecuta el Extrae de CuentaCredito
c = new CuentaConjunta("Juan", "José", 100);
c.Extrae(200); // Se ejecuta el Extrae de Cuenta, produce error

```

¿Qué ocurre al intentar extraer de la cuenta de Pedro? En este caso, como el método Extrae se definió como **virtual** en la clase base y luego se redefinió (**override**) en la clase heredera **CuentaCredito**, se ejecutará este último, independientemente del tipo de la variable que hace referencia al objeto. Por su parte, la segunda invocación del método Extrae se realiza sobre un objeto de tipo **CuentaConjunta**. En ese caso se ejecutará el método definido en la clase **Cuenta**, pues la clase heredera no ha proporcionado una nueva definición.

Esta capacidad de que gracias a las especificaciones **virtual** y **override** la llamada **c.Extrae(200)** utiliza el método Extrae según el tipo del objeto que realmente tenga como valor la variable **c** se denomina **polimorfismo**⁷.

De esta manera, el programador no tiene que estar determinando cuál de las diferentes implementaciones de un método es la que se debe llamar, sino que solo debe decidir cuál tipo de objeto crear (si de una clase base o de una clase derivada) de acuerdo al comportamiento que desea. En lo adelante serán los propios objetos los que determinarán (según su tipo) cuál es la implementación del método o propiedad que deberán aplicar.

10.5.2 Adaptabilidad con el polimorfismo

El equilibrio anterior entre seguridad y flexibilidad es una de las claves del éxito de la orientación a objetos y de lenguajes como C#. Ud. puede tener una clase **C**, ya probada y confiable, la cual trabaja con objetos de un tipo **T**. Por requerimientos de su aplicación Ud. se ve ahora en la necesidad de ampliar el tipo **T** usando herencia y define una clase **T1** derivada de **T**. Pero por otra parte Ud. también desea seguir utilizando la clase **C** sin tener que cambiar nada en **C**. Lo que nos aporta el polimorfismo es que los objetos de tipo **T1** pueden sustituir a los objetos de tipo **T** sin tener que cambiar nada dentro de **C**. Pero lo que es más interesante aún: el comportamiento de los métodos y propiedades de la clase **C** se adaptará transparentemente (sin que tenga que mediar una programación extra) a aquellas redefiniciones que **T1** haya hecho de los métodos de **T**.

Considere por ejemplo una clase **Tienda**, que tiene un método **Vende**. Este método recibe un **Producto** y una **Cuenta**, y extrae de la cuenta el precio que corresponde al producto (Listado 10-9).

⁷ Del latín *muchas formas*.

```

class Producto {
    public string Nombre { get; private set; }
    public float Precio { get; private set; }
    public Producto(string nombre, float precio) {
        Nombre = nombre; Precio = precio;
    }
}

class Tienda {
    public void Vende(Producto item, Cuenta cliente) {
        cliente.Extrae(item.Precio);
        Console.WriteLine("Vendido {0} a {1} por {2}",
            item.Nombre, cliente.Titular, item.Precio);
    }
}

```

Listado 10-9. Clases Producto y Tienda

De acuerdo con el principio de sustitución, como el parámetro del método `Vende` es de tipo `Cuenta` puede recibir tanto objetos de tipo `Cuenta`, como objetos de cualquier tipo que herede de `Cuenta`. Esto permite que podamos escribir el código del Listado 10-10.

```

Tienda t = new Tienda();
Producto tv = new Producto("TV", 500);
Producto dvd = new Producto("DVD Player", 150);
Producto refri = new Producto("SmartPhone", 700);
CuentaConjunta juan = new CuentaConjunta("Juan", "José", 200);
CuentaCredito pedro = new CuentaCredito("Pedro", 500);
t.Vende(dvd, juan);
t.Vende(refri, pedro);

```

Listado 10-10. Polimorfismo en acción

Note que al definir el método `Vende` de esta manera, si en un futuro se define una clase `CuentaAhorro` que también herede de `Cuenta`, objetos de este nuevo tipo podrán pasarse al método `Vende` y este seguirá funcionando como antes **sin que haya que hacerle ningún cambio**.

10.6 Clases abstractas

Suponga que se quieren implementar tipos que caractericen figuras geométricas a las que se les pueda pedir el cálculo de su área y su perímetro. Los tipos de figuras pudieran ser círculos, rectángulos, triángulos, etc. No sería posible representar todas las figuras mediante una única clase, pues según el tipo de figura será la forma en que se calcule su área y su perímetro. Una primera aproximación a la solución sería definir una clase para representar cada tipo de figura. Para simplificar solo consideremos las clases `Circulo` y `Rectangulo` (Listado 10-11).

```
public class Circulo {
    public Circulo(int radio) { Radio = radio; }
    public double Perimetro { get { return 2 * Math.PI * Radio; } }
    public double Area { get { return Math.PI * Radio * Radio; } }
    public int Radio { get; protected set; }
}
public class Rectangulo {
    public Rectangulo(int ancho, int alto) { Ancho = ancho; Alto = alto; }
    public double Perimetro { get { return Ancho * 2 + Alto * 2; } }
    public double Area { get { return Ancho * Alto; } }
    public int Ancho { get; protected set; }
    public int Alto { get; protected set; }
}
```

Listado 10-11. Clases Circulo y Rectangulo

Sin embargo, la aplicación que se quiere implementar necesita ordenar una colección de figuras según su perímetro (o su área). Es decir, que se necesita implementar un método `Ordena` que recibe como parámetro una colección (por ejemplo un array) de figuras que se quiere ordenar. ¿De qué tipo se define ese array? Si es un array de tipo `Circulo`, entonces no serviría para pasarle un array de tipo `Rectangulo`. No sería razonable definir un método `Ordena` según cada tipo de figura, no se estaría haciendo una buena reusabilidad. Para resolver este problema necesitamos poder aplicar el principio de sustitución para los elementos que se pasen al método `Ordena`. Es decir, que el tipo de los elementos del array que recibe el método `Ordena` debe ser **más general** que los tipos que representan figuras específicas. Este proceso de identificar un tipo o concepto más general y común a varios se conoce con el nombre de **factorización**, que nos lleva a una **abstracción** común a todas las figuras y es fundamental en el paradigma de la programación orientada a objetos.

Para este ejemplo, ¿cuál podría ser un concepto más general que identifique a círculos, rectángulos, triángulos y otras figuras geométricas? Podría considerarse que todas se agrupan bajo el concepto de **figura**. Es decir, que para resolver nuestro problema podríamos definir una clase `Figura` de la cual heredasen todas las clases que representen figuras concretas, como es el caso de `Circulo` y `Rectangulo`. Entonces sería mejor que los elementos del array que recibe el método `Ordena` fuesen de tipo `Figura`. Por el principio de sustitución, cada elemento de este array podría contener entonces lo mismo un objeto de tipo `Circulo` que de tipo `Rectangulo` o de cualquier otro tipo que herede de `Figura` (Listado 10-12).

```
public class Figura { }
public class Circulo : Figura {
    public Circulo(int radio) { Radio = radio; }
    public double Perimetro { get { return 2 * Math.PI * Radio; } }
    public double Area { get { return Math.PI * Radio * Radio; } }
    public int Radio { get; protected set; }
}
```

```

public class Rectangulo : Figura {
    public Rectangulo(int ancho, int alto) { Ancho = ancho; Alto = alto; }
    public double Perimetro { get { return Ancho * 2 + Alto * 2; } }
    public double Area { get { return Ancho * Alto; } }
    public int Ancho { get; protected set; }
    public int Alto { get; protected set; }
}

```

Listado 10-12. Jerarquía de figuras

De acuerdo con esto, la signatura del método `Ordena` quedaría

```
void Ordena(Figura[] figuras)
```

Para implementar este método se necesita acceder a la propiedad `Perimetro` de cada figura en el array, y comparar su valor con el de las demás figuras (Listado 10-13).

```

1. void Ordena(Figura[] figuras) {
2.     Figura aux;
3.     for (int i = 0; i < figuras.Length - 1; i++) {
4.         for (int j = i + 1; j < figuras.Length; j++) {
5.             if (figuras[j].Perimetro < figuras[i].Perimetro) {
6.                 aux = figuras[i];
7.                 figuras[i] = figuras[j];
8.                 figuras[j] = aux;
9.             }
10.        }
11.    }
12. }

```

Listado 10-13. Método para ordenar figuras basado en su perímetro

Pero hasta ahora, según la jerarquía de figuras del Listado 10-12, en ninguna parte se ha dicho que los objetos que se asignen a los elementos del array `figuras` dispongan de la propiedad `Perimetro` (línea 5), por lo que su compilación daría error. Para que el código del Listado 10-13 compile correctamente se debe disponer de una propiedad `Perimetro` en la clase `Figura`. Pero, ¿qué implementación daríamos a `Perimetro` en este caso, si no sabemos qué figura concreta podrá ser? En tanto el perímetro (y el área) son conceptos específicos de cada tipo de figura, no pueden ser implementados para un concepto abstracto general como `Figura`.

Este fenómeno es muy frecuente en la programación orientada a objetos, y para representarlo se dispone en C# del recurso de **clase abstracta**. De este modo una clase abstracta sería como una clase "incompleta" porque no se sabe cómo definir algunos de sus métodos o propiedades, pero sí se sabe que esos métodos o propiedades deben existir (implementados por las clases concretas específicas que deriven de ella).



Aunque realmente C# no fuerza a ello. Para que una clase sea abstracta basta con que se especifique con la palabra `abstract` aun cuando tenga implementados todos sus métodos; en este caso el ser abstracta lo único que significa es que no se pueden crear instancias de la clase. Desde el punto de vista del diseño el uso de `abstract` para tales situaciones es poco frecuente.

```
public abstract class Figura {  
    public abstract double Perimetro { get; }  
    public abstract double Area { get; }  
}
```

Listado 10-14. Clase `Figura` abstracta

La clase `Figura` es abstracta (Listado 10-14) porque tiene al menos una propiedad abstracta, lo cual se indica con la palabra `abstract` en la definición de la propiedad (o del método) y porque solamente aparece la signatura de la propiedad (o del método) sin ninguna implementación. Cuando una clase se define como `abstract` no se pueden crear instancias de ella (no tendría sentido crear instancias de algo si tiene métodos que no están implementados). Hacer `Figura f = new Figura();` provocaría un error de compilación. Esta restricción es coherente y está a tono con la filosofía de promover el desarrollo de software seguro. Si se permitiese una "creación a medias" como la anterior, ¿cómo se interpretaría luego la llamada `f.Perimetro` de una propiedad abstracta a dicho "objeto a medias"?

El método `Ordena` ya puede acceder a la propiedad `Perimetro`, pero ¿qué código se ejecuta al invocar la línea 5 del Listado 10-13? Lo deseable sería que se ejecute la implementación de `Perimetro` que diese cada clase heredera de `Figura`. Para lograr esto debe indicarse en cada clase heredera, mediante la palabra `override`, que se está dando una implementación de los miembros abstractos de la clase base. Es decir, que un miembro `abstract` es en cierto modo equivalente a un miembro `virtual` pero que está por implementar.

La clase `Figura` sirve entonces como **clase base** o **ancestro** a la definición de otras clases "concretas" que extienden a la clase `Figura`. De modo que si se dispone de la clase `Figura`, entonces se pueden definir con más facilidad las clases `Circulo` y `Rectangulo` como descendientes de `Figura` (Listado 10-15).

```
public class Circulo : Figura {  
    public override double Perimetro { get { return 2 * Math.PI * Radio; } }  
    public override double Area { get { return Math.PI * Radio * Radio; } }  
    // Otros miembros de la clase Circulo  
}  
  
public class Rectangulo : Figura {  
    public override double Perimetro { get { return Ancho * 2 + Alto * 2; } }  
    public override double Area { get { return Ancho * Alto; } }  
    // Otros miembros de la clase Rectangulo  
}
```


}

Listado 10-15. Clases Circulo y Rectangulo herederas de Figura

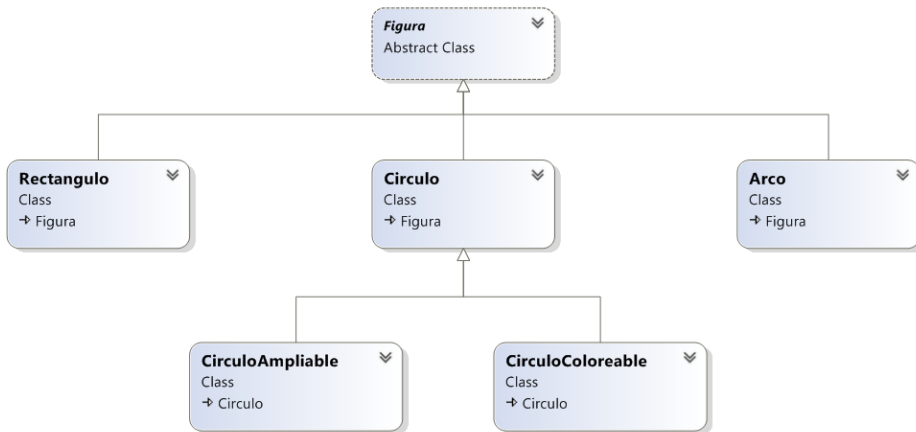


Figura 10-2. Jerarquía de Figuras

Esta posibilidad de que la clase abstracta sirva de raíz o punto de partida para la definición de varias clases que heredan de ella (la extienden) tiene múltiples beneficios:

- *Propicia un mejor modelado del mundo real.* Las clases abstractas nos ayudan a formar una jerarquía de clases en la que en la medida en que subimos en la jerarquía las clases son más generales, más abstractas y en la medida en que descendemos las clases son más específicas, más concretas.

Similarmente ocurre en la biología (de ahí que por analogía la orientación a objetos haya adoptado el término herencia); no tenemos ningún objeto que sea vertebrado a secas, ni mamífero a secas, ni ave a secas, pero sí tenemos un objeto gato Tom que tiene todas las propiedades comunes a los mamíferos o un objeto canario Piolín que tiene todas las propiedades de las aves (Figura 10-3).

La ciencia ha demostrado que este enfoque jerárquico (que no tiene que ver con jefaturas de mando) es útil para "organizar" la información y para apreciar la realidad por niveles de abstracción, que van de lo más general a lo más particular.

- *Facilitar el trabajo de diseño y programación de las clases.* Al programar las clases que heredan de una clase abstracta no hay que replicar monótonamente un mismo código en cada una de las clases descendientes lo cual no sólo simplifica el trabajo, sino que lo hace menos propenso a errores.

- *Eficiencia de implementación.* Una aplicación que trabaje con más de una de las clases descendientes, economiza el espacio ocupado por el código *IL* generado por el compilador de C#, ya que el código

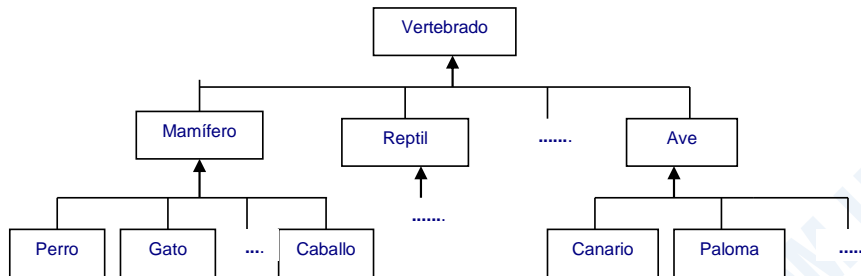


Figura 10-3. Jerarquía de vertebrados

generado para los métodos de una superclase no tienen que repetirse para cada una de sus subclases.

Una clase abstracta como [Figura](#) es un ejemplo en el que la clase abstracta se corresponde con un tipo que expresa lo que tienen en común distintos subtipos ([Circulo](#), [Rectangulo](#), etc). En este ejemplo cada subclase concreta modela a un tipo distinto de objetos de la realidad. Pero una clase abstracta también puede servir para forzar a distintos programadores a cumplir con cierto diseño común impuesto por quien define la clase abstracta; estos tienen la obligación de implementar los métodos abstractos, pero tienen la libertad de implementar los métodos a la manera de cada cual.

10.7 Una jerarquía de circuitos electrónicos

Consideremos que un circuito electrónico posee una propiedad llamada **impedancia**. La impedancia de cada circuito al aplicarle una cierta frecuencia de voltaje depende del tipo de circuito y de las características propias de cada circuito⁸.

Los circuitos más simples pueden ser **resistencias**, **inductores** o **capacitores**. En una resistencia la impedancia es constante y depende solo de las características del circuito, en un inductor la impedancia es proporcional a la frecuencia que se le aplique, y en un capacitor la impedancia es inversamente proporcional a la frecuencia.

Los circuitos se pueden conectar en serie o en paralelo para formar circuitos más complejos. Dada una frecuencia, la impedancia de un circuito en serie es igual a la suma de la impedancia de cada uno de sus circuitos constituyentes y la impedancia de un circuito en paralelo es igual al inverso de la suma de los inversos de las impedancias de sus circuitos constituyentes.

Nos interesa definir una jerarquía que exprese esta variedad de circuitos y que permita calcular para cualquiera de ellos cuál sería su impedancia si se le aplica un voltaje que tiene determinada frecuencia. La intención con esta jerarquía es que ésta estratifique al

⁸ De acuerdo con la Física este ejemplo no es correcto, el lector debe considerarlo solo con objetivos didácticos.

máximo los recursos y propiedades. Es decir, se pretende que en cada nivel de la siguiente jerarquía de clases no haya nada que sea común con las restantes clases del mismo nivel, y que por tanto debería estar definido en un nivel superior de la jerarquía. Por el contrario, todo lo que sea específico de un tipo específico no debería estar definido en un nivel superior de su jerarquía.

La clase más general que se puede definir para modelar este problema es la clase `Circuito`. Lo único común para cualquier circuito es que se le puede pedir que nos dé su impedancia si le damos una frecuencia. De modo que `Circuito` puede verse como una clase abstracta de la que lo único que se sabe es que tiene un método abstracto `Impedancia`

```
public abstract class Circuito {
    public abstract float Impedancia(float frecuencia);
}
```

Listado 10-16 Clase Abstracta Circuito

Para el próximo nivel de estratificación en la jerarquía se pueden apreciar dos grupos de circuitos: los circuitos simples (los cuales no tienen circuitos constituyentes) y los circuitos compuestos (en serie o paralelo). Todos los circuitos simples tienen en común que tienen un valor interno propio de cada circuito que lo necesitará para calcular su impedancia según sea resistencia, inductor o capacitor. Este valor interno lo caracterizaremos como una variable de instancia `c` que se deberá inicializar al construir el circuito. De modo que podemos tener una clase `CircuitoSimple` (Listado 10-17) que herede de `Circuito` y que exprese esto común a todos los simples. Esta clase seguirá siendo abstracta porque aún no sabemos cómo se calcula la impedancia si no sabemos qué clase de circuito simple es.

```
public abstract class CircuitoSimple : Circuito {
    protected float c; // constante propia a cada circuito simple
    protected CircuitoSimple(float constante) {c = constante;}
}
```

Listado 10-17. Clase abstracta CircuitoSimple

Una subclase puede heredar de una clase abstracta y seguir siendo abstracta. La clase `CircuitoSimple` sigue siendo abstracta porque aún no sabe cómo implementar el método abstracto `Impedancia` que hereda de `Circuito`. C# obliga en estos casos a seguir encabezando la clase con la palabra `abstract`, de este modo se asegura que el programador está consciente de que aún está pendiente de implementar un método abstracto. Sin embargo, la utilidad de esta clase es que aporta a sus clases descendientes el constructor para que lo utilicen las clases que extienden a `CircuitoSimple`.

De `CircuitoSimple` pueden derivar entonces cada uno de los tipos simples *resistencia*, *inductor* y *capacitor*. Note que aunque en el caso de las resistencias el valor de la impedancia es constante para cada circuito (no depende de la frecuencia que se aplique), para aprovechar la utilización polimorfa del método `Impedancia`, dicho método en la clase `Resistencia` también recibe un parámetro para la frecuencia aunque no utilice éste en el cálculo de la impedancia (Listado 10-18). El constructor de las clases `Resistencia`,

`Inductor` y `Capacitor` se limita a invocar el constructor de la clase base (`CircuitoSimple`), pasándole la constante que caracteriza a estos circuitos. Note que en este caso las clases descendientes de `CircuitoSimple` deben indicar un constructor explícitamente, pues la clase base no definió un constructor sin parámetros.

```
class Resistencia : CircuitoSimple {
    public Resistencia(float constante) : base(constante) {}
    public float Impedancia(float frecuencia) {return c;}
    /* La frecuencia no se utiliza pero se incluye como parámetro para
       que el método pueda ser polimorfo con el de Circuito */
}
class Inductor : CircuitoSimple {
    public Inductor(float constante) : base(constante) {}
    public float Impedancia(float frecuencia) { return c*frecuencia; }
}
class Capacitor : CircuitoSimple {
    public Capacitor(float constante) : base(constante) {}
    public float Impedancia(float frecuencia) { return c/frecuencia; }
}
```

Listado 10-18. Clases Resistencia, Inductor y Capacitor

Se introduce en la jerarquía una clase `CircuitoCompuesto` (Listado 10-19) que también hereda de `Circuito`. Los circuitos compuestos tienen en común que están constituidos a su vez por dos o más circuitos. Para simplificar la representación de los circuitos compuestos se ha aprovechado que todo circuito en serie o en paralelo siempre se puede ver como la conexión en serie o en paralelo de dos circuitos, esto se ilustra en la Figura 10-4. Note que la clase `CircuitoCompuesto` también debe ser abstracta, pues sin conocer el tipo de circuito (serie o paralelo) no se puede implementar el método `Impedancia`.

```
public abstract class CircuitoCompuesto : Circuito {
    protected Circuito c1;
    protected Circuito c2;
    protected CircuitoCompuesto(Circuito circ1, Circuito circ2) {
        c1 = circ1; c2 = circ2;
    }
}
```

Listado 10-19. Clase abstracta CircuitoCompuesto

Ahora las dos subclases concretas `Serie` y `Paralelo` (Listado 10-20) heredan de la clase `CircuitoCompuesto`, heredando de esta clase lo común a ambas (el par de variables de instancia `c1` y `c2` y el constructor) solo teniendo que aportar cada una su implementación concreta del método `Impedancia`.

```
1 class Serie : CircuitoCompuesto {
2     public Serie(Circuito circ1, Circuito circ2) : base(circ1, circ2) {}
3     public float Impedancia(float frecuencia) {
4         return c1.Impedancia(frecuencia) + c2.Impedancia(frecuencia);
```

```

5  }
6  }
7  class Paralelo : CircuitoCompuesto {
8      public Paralelo(Circuito circ1, Circuito circ2) : base(circ1, circ2) {}
9      public float Impedancia(float frecuencia) {
10         return 1/(1/c1.Impedancia(frecuencia) + 1/c2.Impedancia(frecuencia));
11     }
12 }

```

Listado 10-20. Clases Serie y Paralelo

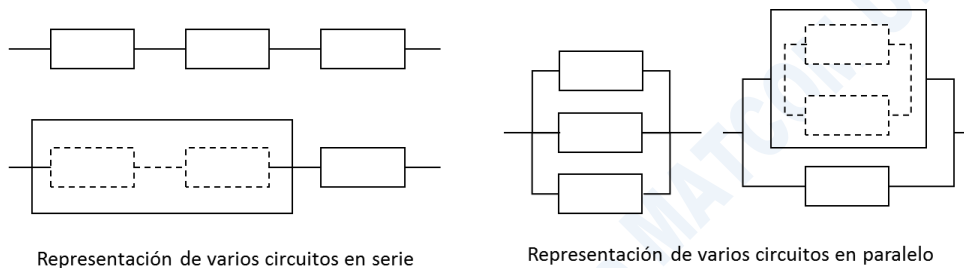


Figura 10-4. Conexión de varios circuitos en serie y paralelo

En estas dos clases está presente otra vez la "magia del polimorfismo", en ambas el método `Impedancia` se apoya a su vez en que cada uno de sus circuitos constituyentes `c1` y `c2` sabe "polimórficamente" calcular su impedancia (líneas 4 y 10 del Listado 10-20). Note que en este caso también las clases `Serie` y `Paralelo` se limitan a invocar el constructor de la clase base, pasándole los dos circuitos que lo conforman.

La jerarquía completa se muestra en la Figura 10-5, note como cada nivel de la jerarquía aporta alguna definición que es común a sus niveles inferiores. Es interesante notar en este ejemplo como una clase descendiente en la jerarquía (`CircuitoCompuesto`) utiliza a su vez una clase superior de su misma jerarquía (`Circuito`); esto permite formar *estructuras recursivas*, como el de este caso en el cual un circuito puede estar constituido a su vez por otros circuitos. Las estructuras recursivas son muy útiles por la flexibilidad que aportan, en el Capítulo 13 se verán otros ejemplos de este tipo de estructuras.



Figura 10-5. Jerarquía de circuitos electrónicos

El siguiente fragmento de código calcula la impedancia, para una frecuencia 10, de un circuito en paralelo formado por un circuito en serie (de una resistencia de constante 4 y un inductor de constante 3) y un capacitor de constante 2⁹. Realmente para C# todo podría escribirse en una sola línea, se han introducido las sangrías y los cambios de línea para dar mayor legibilidad.

```
Circuito c = new Paralelo(new Serie(  
    new Resistencia(4),  
    new Inductor(3)),  
    new Capacitor(2));  
Console.WriteLine("Impedancia igual a " + c.Impedancia(10));
```

Un lector acostumbrado de la escuela de la programación imperativa tradicional podrá asombrarse que en ningún momento se ha escrito código para guardar explícitamente información sobre el tipo de cada circuito. Sólo hay que limitarse a utilizar el constructor deseado, será el *framework* de ejecución de .NET (CLR) quien internamente conocerá cuál es el tipo de cada circuito constituyente y aplicará en consecuencia el método **Impedancia** correspondiente. La Figura 10-6 ilustra el proceso de construcción del circuito anterior; para mayor claridad, se ha incluido el nombre de cada tipo junto con cada circuito.

⁹ Disculpe, lector, si es Ud. un conocedor de Física; estos valores numéricos se han tomado libremente.

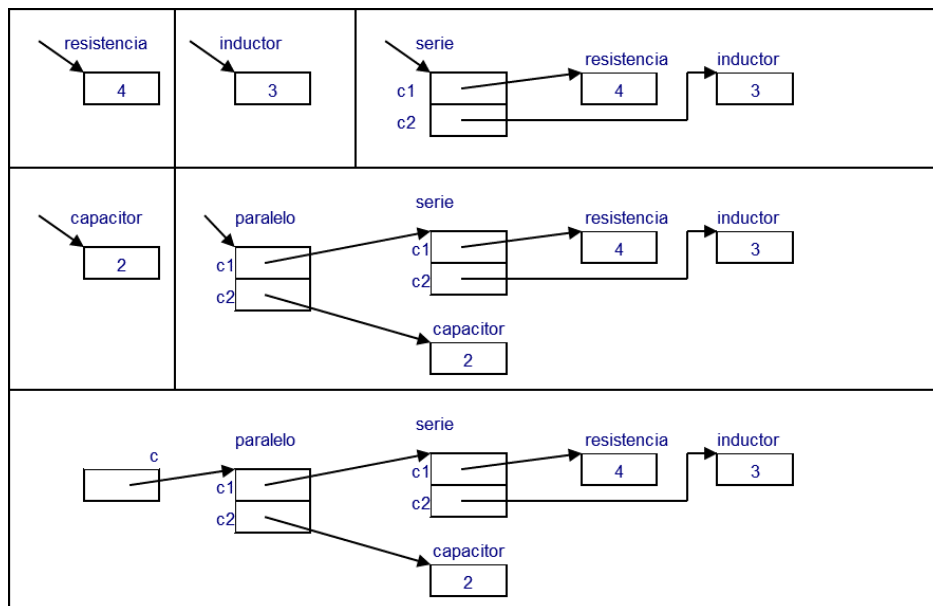


Figura 10-6. Pasos en la creación del circuito

10.8 Operación de conversión de tipo (*casting*)

El principio de sustitución permite asignar un objeto instancia de una subclase a una variable del tipo de una superclase de dicha subclase. C# garantiza una total seguridad en esta operación porque a la variable del tipo de la superclase sólo le permite aplicar métodos que existen en la superclase. En ocasiones es necesario hacer lo contrario, es decir, que un objeto que sea instancia de una subclase esté asociado a una variable del tipo de la superclase, pero que en otro contexto del código se quiera asignar a una variable del tipo de la subclase y por consiguiente poder acceder a las funcionalidades más específicas que proporciona la subclase.

Por ejemplo, si se hace `Cuenta c = new CuentaTransferencia(...)`, por el principio de sustitución se tendría en una variable `c` de tipo `Cuenta` un objeto instancia de una subclase de `Cuenta`. Por tanto, se pueden usar a través de `c` todos aquellos métodos de `Cuenta` (estén o no redefinidos por `CuentaTransferencia`) pero no podemos usar métodos únicos de `CuentaTransferencia`. Es decir, se puede hacer `c.Deposita(100)` o `c.Saldo` pero si se escribe `c.Transfiere(...)` será reportado como error de compilación. C# permite volver a considerar la cuenta asociada a `c` con todos los derechos de una `CuentaTransferencia`, pero exige que esto se indique explícitamente. Es decir, hay que indicarle al compilador de C# que considere que el objeto asociado a `c` es realmente una `CuentaTransferencia`. Esto se hace mediante una operación conocida como **conversión de tipo** o **cast**. Para este ejemplo, habría que escribir entonces `((CuentaTransferencia)c).Transfiere(...)` para poder invocar el método `Transfiere` sobre el objeto al que hace referencia la variable `c`.

Esta operación de conversión tiene la forma

(nombre de tipo)entidad

donde *entidad* es una variable, un parámetro, una propiedad o la aplicación de un método no **void** (que retorne un objeto).

De este modo el compilador puede interpretar entonces la variable *c* como si fuese de tipo **CuentaTransferencia** y permitir aplicar métodos o propiedades definidos en **CuentaTransferencia** sin que esto sea un error. Ahora bien, como C# no puede garantizar que durante la ejecución el objeto asociado a *c* sea realmente del tipo **CuentaTransferencia** (con el que se pretende ahora interpretarlo), entonces genera código para verificar esto en ejecución, y de no cumplirse se lanza una excepción **InvalidCastException**.

```
Cuenta c = new CuentaTransferencia("Juan", 100);
CuentaCredito cuentaCredito = (CuentaCredito)c; // InvalidCastException
```

Ciertamente esta operación de casting no significa que en ejecución ocurrirá alguna "conversión física" de un "formato" a otro, significa que el compilador generará código para comprobar si el tipo del objeto que estará conectado a la entidad es realmente del tipo con el que se ha querido interpretar. Es decir, no hay un coste adicional de tiempo por hacer una conversión sino sólo por comprobar cuál es el tipo del objeto.

Hay situaciones en las que desde la compilación se puede detectar que una tal conversión nunca será posible, de modo que en esos casos el compilador puede ser más preventivo y reportar error de compilación en lugar de una excepción en tiempo de ejecución. Por ejemplo, el intento de hacer `((Rectangulo)c).Perimetro` será reportado error por el compilador porque si la variable *c* es de tipo **Cuenta** de ningún modo puede ser posible que en *c* pueda haber un objeto de tipo **Rectangulo** ya que la clase **Rectangulo** no es descendiente de **Cuenta**.

Abusar de esta operación puede ser reflejo de malos hábitos de programación orientada a objetos, que pretende promover el desarrollo de software robusto y confiable, en el que preferentemente los errores no deben aparecer como excepciones de ejecución.

10.9 La superclase raíz Object

C# ofrece una clase **Object**¹⁰ que se considera la raíz suprema de toda jerarquía de clases. Toda clase en C# hereda implícitamente de la superclase **Object** (Listado 10-21). De modo que cualquier clase es descendiente directa o indirecta de **Object**, independientemente de que pueda heredar explícitamente de otra clase.

Esto tiene varios beneficios. Por mediación de esta clase se ofrecen funcionalidades comunes para todas las clases. Queda entonces a discreción de cada clase decidir si redefine o no dichos métodos. Además, el principio de sustitución hace que sea posible asignar objetos de cualquier tipo a una variable de tipo **Object**, lo cual en ocasiones puede resultar conveniente.

```
public class Object {
```

¹⁰ En C# se puede usar de manera equivalente la palabra reservada **object** (con minúsculas), que es un alias de la clase **Object**.


```

public virtual bool Equals(object obj);
public virtual int GetHashCode();
public Type GetType();
public virtual string ToString();
}

```

Listado 10-21. Métodos de la clase `Object`

10.9.1 El método `Equals`

Uno de los métodos interesantes definidos en `Object` es el método `Equals`, que se utiliza para comparar dos objetos y determinar si son iguales

```
public virtual bool Equals(Object obj) { return this == obj; }
```

Esta definición le puede parecer trivial al lector. No tiene mucho sentido hacer `x.Equals(y)` si con ello se obtiene el mismo resultado que con `x == y`, es decir, es evidente que si los dos objetos a comparar tienen igual referencia es porque son el mismo objeto. La utilidad de este método es que al estar en la raíz de todas las clases y ser virtual entonces cualquier método que reciba un parámetro de tipo `object` aplicaría el criterio de igualdad del objeto real que se le pase como parámetro. Esto ocurre por ejemplo en los métodos de las clases de estructuras de datos ([Capítulo NNN](#)) cuando se quiere buscar un objeto dentro de la estructura.

El Listado 10-22 muestra una clase `Fecha` que ha redefinido el método `Equals` para que dos objetos `Fecha` sean iguales si tienen el mismo día, el mismo mes y el mismo año.

```

1 class Fecha {
2     int d, m, a;
3     ...
4     public override bool Equals(object x) {
5         Fecha f = (Fecha)x;
6         return (d == f.d) && mes == (m == f.m) && (a == f.a);
7     }
8 }

```

Listado 10-22. Método `Equals` de la clase `Fecha`

Note como es necesario utilizar la operación de *cast* o conversión (línea 5) que se explicó anteriormente. El problema es que el parámetro del método `Equals` definido en la clase `Object` es de tipo `Object` y para redefinir `Equals` en la clase `Fecha` el parámetro tiene que seguir siendo de tipo `Object` ya que C# exige que la redefinición de un método mantenga la misma signatura (es decir, el mismo nombre del método correspondiente en la superclase, los mismos tipos de los parámetros y el mismo tipo de retorno). Para saber si dos fechas son iguales se necesita comparar las componentes (`d`, `m` y `a`) y para esto hay que garantizarle al compilador que el otro objeto a comparar es también una fecha y que podemos referirnos entonces a las componentes `d`, `m` y `a`. En caso de pasar como parámetro al `Equals` un objeto que no sea una `Fecha`, por ejemplo de tipo `Cuenta` o una `Figura`, se devolverá `false` (líneas 5 y 6).

Al hacer la operación de casting de la línea 5 del código del Listado 10-22 se disparará excepción en el caso de que el objeto que se pase como parámetro no sea realmente de tipo `Fecha`.

Sin embargo, puede considerarse que dos objetos de diferente tipo siempre son distintos. Esto es lo que asume .NET predeterminadamente en la implementación del `Equals` en la clase `Object`.

10.9.2 Casting y los operadores `is` y `as`

¿Cómo prever antes de efectuar un `cast` si la conversión será válida? Para ello C# brinda el operador `is`, que permite preguntar en ejecución por el tipo real de un determinado objeto. Entonces el método `Equals` de la clase `Fecha` quedaría como se muestra en el Listado 10-23

```
class Fecha {
    int d, m, a;
    ...
    public override bool Equals(object x) {
        if (x is Fecha) {
            Fecha f = (Fecha)x;
            return (d == f.d) && mes == (m == f.m) && (a == f.a);
        }
        else return false;
    }
}
```

Listado 10-23. Verificar el tipo antes de hacer el cast usando `is`

Un efecto similar podría lograrse con el operador `as`, como se muestra en el Listado 10-24. El resultado de hacer `x as Fecha` es para el compilador de tipo `Fecha` si el valor de `x` es realmente un objeto de tipo `Fecha`, pero devuelve `null` si el valor de `x` no es de tipo `Fecha`. Por esta razón, el operador `as` solo puede usarse para convertir tipos que se traten por referencia. Por ejemplo, tratar de aplicar el operador `as` para convertir un objeto a `int` produciría un error de compilación, pues en el caso de que el objeto no fuera un `int`, no podría devolverse `null`.

```
class Fecha {
    int d, m, a;
    ...
    public override bool Equals(object x) {
        Fecha f = x as Fecha;
        if (f!=null)
            return (d == f.d) && mes == (m == f.m) && (a == f.a);
        else return false;
    }
}
```

Listado 10-24 Haciendo cast con el operador `as`

Casos como el de este ejemplo `Fecha` es un patrón que se aplica con frecuencia: considerar que dos objetos son iguales si son instancias de la misma clase y si cada una de sus componentes son iguales entre sí. Esto se conoce como **igualdad bit a bit**. Esto es lo común cuando todas las componentes son a su vez de un tipo simple.

Si alguna componente no es de tipo simple, sino que es a su vez un objeto, entonces una interpretación posible de `Equals` es aplicar recursivamente `Equals` para la componente en cuestión con la componente respectiva del otro objeto (cuidando de no caer en un ciclo). Si el tipo de la componente no redefinió `Equals`, esta nueva aplicación de `Equals` será la que ya hemos visto que se hereda de `Object` (es decir la comparación de igualdad por identidad `"=="`), pero si el tipo de la componente tiene su propia definición de `Equals` deberá ser esta la que se aplique. Este comportamiento es posible gracias al polimorfismo.

Sin embargo, no debemos engañarnos, no todos los casos son tan triviales como éste de `Fecha`. Que dos objetos "sean iguales componente a componente" puede que sea una condición suficiente para decir que los objetos "son iguales" pero no es una condición necesaria. Considere por ejemplo la jerarquía de circuitos. En una primera impresión podemos pensar que dos circuitos son iguales si sus variables de instancia son iguales. Esta interpretación vale para los circuitos simples pero no vale para los circuitos compuestos porque no importa el orden en que se conectan dos circuitos el efecto del circuito resultante es el mismo. Es decir los circuitos

```
Serie cSerie1 = new Serie(new Capacitor(3), new Inductor(5));
```

y

```
Serie cSerie2 = new Serie(new Inductor(5), new Capacitor(3));
```

no son iguales componente a componente pero realmente se comportarían de igual forma como circuitos. Se sugiere al lector como ejercicio reimplementar la jerarquía de circuitos introduciendo correctamente el método `Equals` en los niveles que corresponda.

10.9.3 El método `ToString`

Otro método muy útil que conviene que esté presente en todo objeto y por eso está definido en el tipo `Object` raíz de toda jerarquía, es el método `ToString`. Este método retorna una "representación textual" del objeto en forma de un `string`.

Lo interesante de este método es que posibilita la conversión implícita a `string` en los contextos que convenga. Por ejemplo la operación `"+"` en el siguiente código es legal y es interpretada como concatenación de cadenas.

```
string s = "Salu";
int k = 2;
s = s + 2;
Console.WriteLine(s); // Escribe Salu2
```

C# asume por defecto que el método `ToString` de una `clase`, si no está redefinido, devuelve una cadena formada por el espacio de nombres (*namespace*) donde está definida la clase y el nombre de la clase de la cual el objeto es una instancia. Por ejemplo, si la clase `Fecha` está definida en el espacio de nombres `Programacion`, entonces al aplicar

la implementación predeterminada de `ToString` sobre un objeto de tipo `Fecha`, se obtendría la cadena `"Programacion.Fecha"`.

Una clase puede redefinir el método `ToString` y aportar entonces la representación en forma de cadena que desea para sus instancias. Por ejemplo, la clase `Fecha` puede redefinir el método `ToString` para formar una cadena con el formato *día/mes/año* Listado 10-25

```
class Fecha {  
    int dia, mes, anno;  
    ...  
    public override string ToString() { return dia + "/" + mes + "/" + anno; }  
}
```

Listado 10-25. Método `ToString` de la clase `Fecha`

Entonces, al ejecutar el siguiente fragmento de código

```
Fecha f = new Fecha (12, 5, 2008); // 12 de mayo de 2008  
Console.WriteLine(f);  
se muestra el resultado
```

```
12/5/2008
```

El método `WriteLine` tiene varias sobrecargas, una de las cuales recibe un parámetro de tipo `Object`. Como todos los tipos heredan implícitamente de `Object`, por el principio de sustitución el método `WriteLine` acepta en el código anterior una instancia de `Fecha`. Internamente la implementación del método `WriteLine` invoca el método `ToString` del objeto para obtener su representación en forma de cadena (lo que en este ejemplo significa que "polimórficamente" ejecuta la versión del método que se redefinió en la clase `Fecha`).

Modifique el lector este método `ToString` para que el día y el mes siempre aparezcan en el formato de dos dígitos. Haga otra implementación del método para que el mes aparezca por su nombre en lugar de numéricamente.

La clase `Object` tiene también un método `GetHashCode` que guarda relación con el método `Equals` y que es utilizado en el Capítulo 12 en la implementación de la estructura de datos conocida como diccionario.

10.10 Limitar el polimorfismo. El modificador `sealed`

A veces puede ser deseable impedir que se pueda heredar de una cierta clase o impedir que con la herencia se pueda redefinir el comportamiento de un método. En el caso de C# esto se expresa con el modificador `sealed`.

Una clase puede definirse como `sealed`, y esto evita que se pueda heredar de ella.

```
sealed class C {  
    ...  
}
```

Si se quiere que un método definido como `virtual` en una clase ancestro no se pueda redefinir al heredar, entonces éste se debe marcar como `sealed`.

```
class A {
    public virtual void f() { ... }
}
class B : A {
    ...
    public sealed override void f() { ... }
}
```

De acuerdo con el código anterior, las clases que hereden de `B` no podrán redefinir el método `f`. Note que en C#, si no se marca un método como `virtual`, de manera predeterminada se asume `sealed` ya que para hacer `override` tiene que haber sido definido `virtual` en el ancestro. De modo que solo tiene sentido declarar un método como `sealed` cuando es `override`.

Un método o una clase completa se pueden especificar `sealed` por una de dos razones (o por ambas):

- *Diseño y seguridad*

Se está convencido que el comportamiento del método o de la clase ya está completo y no tiene por qué modificarse. Es decir, en este caso se quiere impedir el polimorfismo y el principio de sustitución, porque se quiere estar seguro que si se tiene `x.f()`, donde `x` es de tipo `B`, esto siempre se comportará de igual modo bien sea porque no podrá haber ningún objeto subtipo de `B` ya que `B` es `sealed`, o bien porque ninguna subclase podría dar una nueva definición del método `f` al ser éste `sealed`.

Por estas razones varias de las clases que el propio C# oferta en su biblioteca de clases son `sealed`, porque los diseñadores de dicha biblioteca no quieren que se les pueda cambiar su comportamiento. El ejemplo más significativo de clase `sealed` es la clase `String`. Imagine lo confuso que podría llegar a ser de entender una aplicación si comportamientos tan establecidos como el de `String` pudieran modificarse.

Obviamente, si una clase o método es `abstract` no puede ser a la vez marcado como `sealed`.

- *Eficiencia (solo para lectores avanzados)*

La otra razón es la eficiencia en tiempo de ejecución. Aunque no tenga conocimientos de compiladores, el lector puede imaginar que no hay ningún misticismo detrás del polimorfismo; si una llamada `x.f()` puede tener varias interpretaciones, según el objeto asociado a `x` y según si el método `f` ha sido redefinido, entonces "alguna averiguación" tendrá que hacer el *runtime* de .NET para poder determinar esto cuando la aplicación esté ejecutando. La ventaja del polimorfismo es que no es el programador el responsable de programar tal "averiguación", ganando por tanto en comodidad, flexibilidad y seguridad. Al especificar que un método (o toda una clase) `f` es `sealed` entonces .NET podrá

hacer una ejecución más eficiente de una llamada como `x.f()` porque no tendrá que hacer averiguaciones adicionales al ser un único método `f` el que podría ser ejecutado.

Con esta explicación no se pretende estimular que se obvie el polimorfismo para buscar eficiencia. Por lo general la razón de utilizar `sealed` es de diseño y seguridad, y si esto además ayuda a mejorar la eficiencia pues bienvenido. Tal es el caso de lo que puede hacer el compilador de C# con las aplicaciones que trabajen con `String`.



En todo caso, tal eficiencia se debería dejar a los compiladores. Aunque un método no se haya especificado `sealed`, si el compilador detecta que en el proyecto no hay, ni pudiese haber, un tipo que redefina al método, entonces también podría obviar generar el código para hacer la tal averiguación.

10.11 Structs

La memoria para representar los objetos instancias de un tipo definido por una clase se reserva en un área llamada *heap*. Se dice que las clases constituyen "tipos por referencia", por la forma en que se manejan los objetos almacenados en la memoria. C# introduce el concepto de `struct` como alternativa para definir un tipo en lugar de usar una clase. Los structs¹¹ se comportan en muchos casos como las clases, siendo su mayor diferencia la forma de almacenamiento en la memoria. Contrariamente a los objetos creados a partir de clases, los structs se consideran "tipos por valor" y por tanto los objetos creados a partir de ellos se almacenan en un área de la memoria denominada pila. Como se menciona en el Capítulo 3, los tipos simples en C# son "tipos por valor". Los structs pueden considerarse entonces como una alternativa a las clases para crear "tipos por valor" personalizados.

Un struct se define de forma similar a una clase, usando la palabra reservada `struct` en lugar de `class`. El Listado 10-26 muestra el código de un struct `Color` para representar colores mediante las componentes RGB¹².

```
struct Color {
    int r, v, a;

    public Color(int rojo, int verde, int azul) {
        r = rojo; v = verde; a = azul;
    }

    public int Rojo { get { return r; } set { r = value; } }
    public int Verde { get { return v; } set { v = value; } }
    public int Azul { get { return a; } set { a = value; } }
```

¹¹ Aunque puede traducirse por estructura, para evitar confusión con otros uso de la palabra estructura en este libro se ha decidido dejar el término en inglés.

¹² RGB, por sus siglas en inglés: Red, Green, Blue.

```

public override bool Equals(object obj)
{
    if (obj is Color) {
        Color c = (Color)obj;
        return c.r == r && c.v == v && c.a == a;
    }
    return false;
}

public override string ToString() {
    return string.Format("R:{0}, V:{1}, A:{2}", r, v, a);
}
}

```

Listado 10-26. Struct Color

Debido a la forma en que se almacenan en la memoria los objetos de un tipo `struct`, su uso suele ser en algunos casos más eficiente que al tratar con objetos creados a partir de tipos por referencia. Esto no significa que al diseñar un tipo deba preferirse usar `struct` en lugar de clase. Generalmente se suelen definir como `struct` aquellos tipos no demasiado complejos, con pocas componentes y una funcionalidad que no necesitará ser redefinida.

Otra diferencia notable entre clases y structs es que los últimos no soportan herencia, por lo que no pueden ser abstractos ni sus métodos o propiedades ser `virtual`. Por tanto no existe polimorfismo ni se aplica el principio de sustitución cuando se trabaja con structs. Este es el costo a pagar por la eficiencia que proporcionan los structs. No obstante un struct puede implementar todas las interfaces que quiera.

Aunque los structs no soportan herencia, sí forman parte de la jerarquía descendiente de la raíz `Object`, y por tanto heredan sus métodos, que sí pueden por lo tanto ser redefinidos según convenga. En el Listado 10-26 se muestra una implementación del método `Equals` heredado de la clase `Object`, que devuelve `true` si las componentes de ambos objetos tienen los mismos valores, y `false` en otro caso. Asimismo, se ha redefinido el método `ToString` para devolver también la información de un color como una representación textual indicando los valores de cada componente.

```

Color color1 = new Color(10, 20, 30);
Color color2 = color1;
color1.Rojo = 50;

```

```

Console.WriteLine(color1);
Console.WriteLine(color2);

```

Listado 10-27. Usando el struct Color

De acuerdo con la definición del struct `Color`, si se ejecuta el código del Listado 10-27, se escribirá en la ventana de consola

```

R:50, V:20, A:30
R:10, V:20, A:30

```

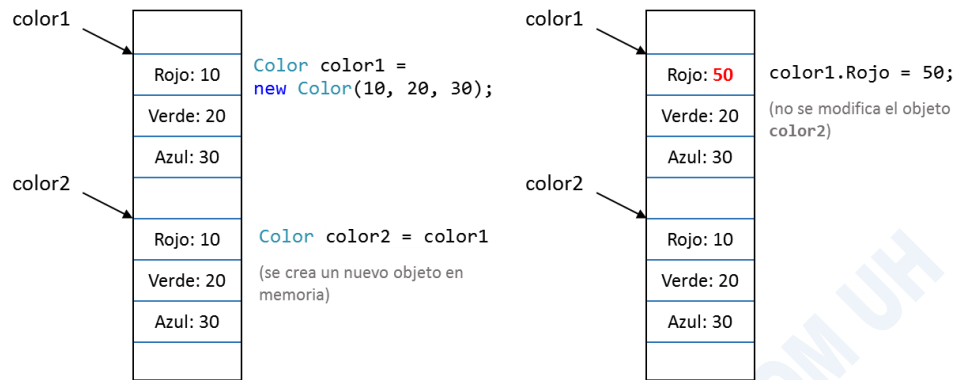


Figura 10-7. Comportamiento de los structs en memoria

Note que este comportamiento (Figura 10-7) se debe a que el tipo `Color` se maneja por valor y no por referencia. Es decir, que al hacer `Color color2 = color1`, se crea en la memoria un nuevo objeto independiente de `color1`, con los mismos valores para cada una de sus componentes. Luego, al hacer `color1.Rojo = 50`, no se afecta la componente Rojo de `color2`.



Los tipos `Point` y `Color` del espacio de nombres `System.Drawing` y el tipo `DateTime` del espacio de nombres `System`, son structs y por tanto se tratan como tipos "por valor".

10.11.1 Boxing y Unboxing

El lenguaje C# define tipos por valor y tipos por referencia para optimizar la ejecución de los programas que los usan. Sin embargo, en algunas situaciones puede necesitarse asignar un tipo por valor a una variable declarada como un tipo por referencia. El ejemplo más general es el de poder asignar "cualquier cosa" a una variable de tipo `Object`.

A una variable de tipo `Object` puede asignársele un objeto de otro tipo definido por una clase, ya sea definida en .NET o definida por terceros. En este caso se pone en práctica el principio de sustitución, y el compilador detecta como válida esta asignación, pues toda clase hereda implícitamente de `Object`. Es por eso que el método `WriteLine` de la clase `Console` puede recibir como parámetro cualquier objeto (incluyendo los de los tipos básicos que caen en la misma categoría de tipo por valor que los structs) sin importar su tipo, pues una de sus sobrecargas recibe `Object`.

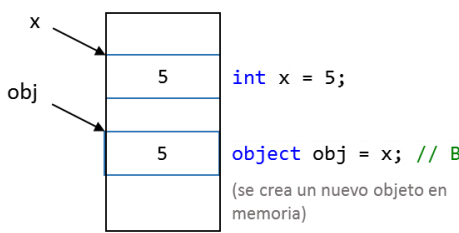
```
Fecha fecha = new Fecha(12, 5, 2008);
Console.WriteLine(fecha);
```

En ocasiones puede resultar necesario asignar un objeto de un tipo tratado por valor a una variable de tipo `Object`, para usarlo en algún contexto polimorfo, como el caso del

Comment [m1]: Pasar este subepígrafe para el capítulo de Interfaces cuando Octavio lo mande

Listado 10-27, donde se pasaron objetos de tipo `Color` al método `WriteLine`. Esto puede hacerse en C# y "por detrás del telón"; lo que ocurre se denomina **boxing**. La acción de *boxing* ocurre cuando se asigna un tipo por valor a un tipo por referencia, como es el caso de `Object`. Siendo así, puede hacerse lo siguiente

```
int x = 5;
object obj = x; // Boxing
```



(se crea un nuevo objeto en memoria)

Figura 10-8. Boxing

Como se ilustra en la Figura 10-8, al asignar la variable `x` a la variable `obj`, se crea una copia del objeto en memoria.

La operación contraria se conoce como **unboxing**, e implica convertir un tipo por referencia en un tipo por valor usando el operador de `cast`.

```
x = (int)obj;
```

En el caso de los tipos básicos quizás no resulte notable la utilidad del *boxing* y *unboxing*, salvo para la uniformidad del `WriteLine`, pero si trasladamos su aplicación al escenario de los structs quizás resulte más claro.

Los structs son tipos que se tratan por valor, por lo que al asignarlos a una variable de tipo `Object` ocurre el boxing. Los structs pueden implementar interfaces, y cuando un objeto creado a partir de un struct se asigna a una variable cuyo tipo es una interfaz, también ocurre **boxing**!

Para ilustrar el boxing-unboxing con structs veamos un ejemplo donde se ha definido una interfaz `ICredencial` y un struct `Credencial` que la implementa (Listado 10-28).

```
interface ICredencial {
    string Titular { get; }
    DateTime FechaExpiracion { get; }
    void Renovar(int dias);
}

public struct Credencial : ICredencial {
    public Credencial(string titular) {
        this.titular = titular;
        this.fechaExpiracion = DateTime.Today.AddDays(30);
    }
    private string titular;
    public string Titular {
        get { return titular; }
    }
}
```

Comment [OH2]: En itálicas en todo el documento?

Comment [OH3]: En itálicas en todo el documento?

Comment [mkm4]: El capítulo de interfaces viene después en todo caso pasar este ejemplo como un epígrafe para allá, o no hablar de esto ¿qué opinas?

Comment [LSS]: Sí, mejor pasar el epígrafe de boxing-unboxing para el final del cap de interfaces, o mejor el de structs completo.

Comment [OH6]: Efectivamente, creo mejor pasar este ejemplo al cap. De interfaces. O introducir las interfaces antes, contraponiéndolas a las clases abstractas?

Comment [mkm7]: Bueno pero termina de revisarlo y arreglarlo con lo que te he puesto ahora para después pasarlo ya sin arrastrar cambios.

```

    }
    private DateTime fechaExpiracion;
    public DateTime FechaExpiracion {
        get { return fechaExpiracion; }
    }
    public void Renovar(int dias) {
        fechaExpiracion = fechaExpiracion.AddDays(dias);
    }
}

```

Listado 10-28. Interface ICredencial y struct Credencial

Suponga la existencia de un método `RenovarCredencial` que recibe como parámetro un `ICredencial` y que invoca el método `Renovar` definido por esta interface.

```
void RenovarCredencial(ICredencial c) { c.Renovar(10); }
```

De acuerdo con esto, se puede hacer entonces

```

Credencial cred = new Credencial("Juan");
RenovarCredencial(cred); // Boxing implícito
Console.WriteLine(cred.FechaExpiracion);

```

¿Qué fecha se escribirá? ¿La fecha original con la que se creó el objeto `cred` o diez días después de la original? Como `Credencial` es un `struct`, sus objetos se tratan "por valor" y al pasarlo como parámetro del método `RenovarCredencial` se hace un boxing con una copia del mismo lo que a su vez hace una copia de la referencia al string `Titular` y una copia de la fecha `FechaExpiracion` ya que el tipo `DateTime` es también `struct`. Por tanto, las modificaciones a la fecha dentro del método `RenovarCredencial` afectan a la copia de la fecha realizada como parte del boxing y no a la fecha original. Es decir, que el código anterior escribirá la fecha con la que originalmente se creó el objeto.

En este caso, el boxing está ocurriendo de forma implícita al pasar el parámetro real del tipo `struct Credencial` al parámetro formal definido de tipo `ICredencial` (Figura 10-9).

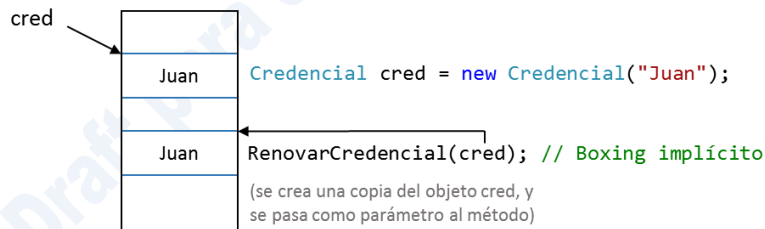


Figura 10-9. Boxing implícito

¿Podría lograrse el efecto de que el método `RenovarCredencial` modifique el objeto original? Sí, y la vía para lograrlo es hacer un uso explícito del concepto de boxing. Con el siguiente código se logra este propósito.

```

ICredencial cred = new Credencial("Juan"); // Boxing explícito
RenovarCredencial(cred);

```

```
Console.WriteLine(cred.FechaExpiracion);
```

La sutil diferencia en el texto del código (se cambió el tipo de la variable `cred` de `Credencial` a `ICredencial`) modifica por completo el resultado de ejecutar el método `RenovarCredencial`. En este caso ocurre un boxing cuando de forma explícita asignamos un tipo por valor (el struct `Credencial`) a una variable de tipo por referencia (la variable `cred` de tipo `ICredencial`) de modo que aunque ha ocurrido un boxing del objeto creado por el `new`, el objeto de tipo `Credencial` al que tenemos acceso es al referido por la variable `cred`. Luego, al método `RenovarCredencial` se le está pasando la referencia al mismo objeto que al que tenemos acceso a través de `cred` y es por eso que la fecha que resulta modificada es la misma (Figura 10-10).

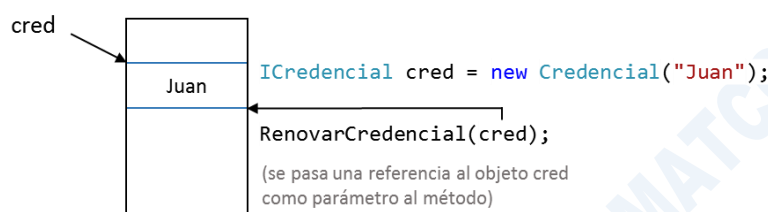


Figura 10-10. Boxing explícito

En resumen, los structs pueden resultar apropiados cuando se trate de tipos relativamente sencillos, con una funcionalidad bien definida, que no necesitará ser extendida o adaptada mediante herencia, y que por razones de eficiencia no convenga modelarse con clases.



En algunas situaciones en las que se requiere interoperar con otras plataformas o lenguajes, puede resultar una exigencia trabajar con structs en lugar de clases. Un ejemplo de ello es cuando se necesita invocar desde C# algunas funciones definidas en bibliotecas externas y en otros lenguajes, como es el caso del API de Windows definido en C++.

10.12 Métodos Extensores

Los métodos extensores son un recurso introducido a partir de C# 3.0 para permitir la incorporación de nuevas funcionalidades en tipos existentes. Hasta ahora hemos dicho que con la herencia se puede ampliar la funcionalidad de un tipo definiendo una nueva clase que herede de la clase base, y añada el nuevo método con la funcionalidad que se quiere incluir. Sin embargo, una clase puede ser `sealed` y por tanto no se puede heredar de ella o se quiere lograr el efecto de "añadirle" métodos sin tener que escribir toda una nueva clase. Esto puede lograrse con los métodos extensores que se asocian a un determinado tipo mediante el primer parámetro del método.

Un **método extensor** es un método estático definido dentro de una clase estática y cuyo primer parámetro va precedido de la especificación `this`. El tipo de este primer parámetro es el tipo que se quiere "ampliar". Mejor veamos esto con un ejemplo: el

Listado 10-29 le añade el método `Inverse` a la clase `string` para a partir de una cadena devolver la cadena reverso.

```
static class StringExtensions {  
    public static string Inverse(this string s) {  
        StringBuilder sb = new StringBuilder();  
        foreach (var c in s)  
            sb.Insert(0, c);  
        return sb.ToString();  
    }  
}
```

Listado 10-29. Método extensor `Inverse` para `string`

Este método puede usarse entonces sobre cualquier variable de tipo `string`, usando la *notación punto* (dot notation) de la POO, como si fuera uno de los métodos definidos en la propia clase `string`. Como sabemos, una de las bondades de la *notación punto* es el auxilio que nos da una herramienta como el Intellisense (Figura 10-11)

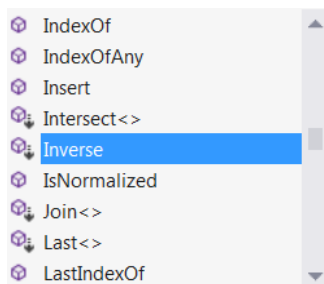


Figura 10-11 Intellisense Mostrando el `Inverse`

```
string s = "roma";  
Console.WriteLine(s.Inverse()); // amor
```

Para ello basta con incluir una referencia al ensamblado donde se haya definido la clase `StringExtension` y la respectiva directiva `using` indicando el espacio de nombres en el que encuentra esta la clase.

10.12.1 Extendiendo el tipo `Point`

Suponga que se quiere añadir una clase `Poligono` a la jerarquía de figuras utilizada en este capítulo. Un polígono se puede representar por una secuencia de puntos (vértices) en el plano. Para ello utilizaremos el tipo `Point` del espacio de nombres `System.Drawing`, mediante el cual se puede representar un punto en el plano a través de sus propiedades `x` y `y`. Para el cálculo del perímetro hay que calcular la distancia que existe entre dos puntos consecutivos del polígono e ir las sumando, pero el tipo `Point` no proporciona ningún método para tal fin. Se podría entonces definir un método extensor `Distance`

(Listado 10-30) de modo que si `p1` y `p2` son objetos de tipo `Point`, pudiera escribirse `p1.Distance(p2)`, y esto devolvería la distancia entre los puntos `p1` y `p2`.

```
static class PointExtension {
    public static double Distance(this Point p, Point otroPunto) {
        return Math.Sqrt(Math.Pow(p.X-otroPunto.X, 2) +
            Math.Pow(p.Y-otroPunto.Y, 2));
    }
}
```

Listado 10-30. Método extensor `Distance` para la clase `Point`

De esta manera la implementación de la propiedad `Perimetro` quedaría simple y elegante en la clase `Poligono` (Listado 10-31), y la "extensión" del tipo `Point` con un método `Distance` puede ser reutilizada por otros.

```
class Poligono {
    Point[] vertices;

    public double Perimetro {
        get {
            double perimetro = 0;
            for (int i = 0; i < vertices.Length - 1; i++) {
                perimetro += vertices[i].Distance(vertices[i + 1]);
            }
            perimetro += vertices[0].Distance(vertices[vertices.Length - 1]);
            return perimetro;
        }
    }
}
```

Listado 10-31. Perímetro del polígono basado en el método extensor `Distance` de `Point`

10.12.2 Métodos extensores y el polimorfismo

Como se ha ilustrado, existen situaciones donde puede resultar útil definir un método extensor. Sin embargo, no se deje engañar el lector por la forma en que C# permite la llamada de los métodos extensores usando la notación de objetos. Aun cuando "da la impresión" de que el método se ha definido en la propia clase, no es más que azúcar sintáctica para llamar a un método `static`, que puede invocarse de la forma habitual con el mismo resultado, como se muestra en el Listado 10-32

```
string s = "roma";
Console.WriteLine(StringExtension.Inverse(s));

Point p1 = new Point(5, -3);
Point p2 = new Point(-8, 2);
double d = PointExtension.Distance(p1, p2);
```

Listado 10-32. Uso de los métodos extensores en notación clásica

Suponga que existe ya toda una jerarquía de `Cuenta` y se quiere dotar a todas las clases de la jerarquía de la capacidad de hacer transferencias sin tener que reprogramar la clase `Cuenta` y por tanto recompilar todas las clases de la jerarquía. Se puede definir para esto un método extensor (Listado 10-33)

```
public static void Transfiere(this Cuenta c, float cantidad,
                             Cuenta cuentaDestino) {
    if (cantidad <= 0)
        throw new Exception("Cantidad a transferir debe ser mayor que cero");
    else if (c.Saldo >= cantidad) {
        cuentaDestino.Deposita(cantidad);
        c.Extrae(cantidad);
        Console.WriteLine("Transferencia de {0} a {1}", c.Titular,
                           cuentaDestino.Titular);
    }
    else throw new Exception("No hay saldo suficiente para hacer el
    traspaso");
}
```

Listado 10-33. Método extensor `Transfiere` para la clase `Cuenta`

Para diferenciarlo de la implementación del método `Transfiere` de la clase `CuentaTransferencia`, vamos a hacer que el método extensor `Transfiere` escriba en la consola un mensaje indicado de quién es la transferencia y hacia quién se dirige.

De acuerdo con esta definición, sería correcto hacer entonces

1. `Cuenta cuenta = new Cuenta("Juan", 65);`
2. `CuentaCredito cuentaCredito = new CuentaCredito("Pedro", 100);`
3. `Cuenta cuentaTransferencia = new CuentaTransferencia("José", 200);`
4. `cuenta.Transfiere(50, cuentaCredito);`
5. `cuentaTransferencia.Transfiere(50, cuentaCredito);`

Al ejecutar la línea 4 se invoca el método extensor, como es de esperar, pues la clase `Cuenta` no tiene definido un método `Transfiere`. ¿Qué método se invoca al ejecutar la línea 5? Podría esperarse que se ejecute el método `Transfiere` definido en la clase `CuentaTransferencia`, pues la variable `cuentaTransferencia`, aunque definida de tipo `Cuenta`, hace referencia a un objeto de tipo `CuentaTransferencia`, y se esperaría que entrase en funcionamiento el polimorfismo. Sin embargo, no es esto lo que ocurre y se invoca en este caso también el método extensor. Si analizamos el ejemplo con más detenimiento se verá que este comportamiento tiene sentido, pues de hecho antes de definir el método extensor, el código de la línea 5 hubiera sido reportado como error de compilación, pues no existía una definición del método `Transfiere` en la clase `Cuenta`.

Para invocar el método `Transfiere` definido en `CuentaTransferencia` se podría utilizar el operador de conversión antes de invocar el método, del mismo modo que habría que hacerlo si no existiese el método extensor.

```
((CuentaTransferencia)cuentaTransferencia).Transfiere(50, cuentaCredito);
```

Otra utilidad de los métodos extensores es la de lograr la apariencia de añadir métodos a un tipo interface. Las interfaces se estudian en el Capítulo 11.

10.13 Practique la herencia

1. Modifique la jerarquía de figuras para incluir clases para representar triángulos, cuadrados y polígonos en general.
2. Implemente los métodos `Equals` y `ToString` para las clases de la jerarquía de figuras.
3. Implemente los métodos `Equals` y `ToString` para las clases de la jerarquía de circuitos. Tenga en cuenta que la igualdad de circuitos no puede basarse solamente en la igualdad componente a componente.
4. Implemente un método que reciba un `Circuito` y devuelva la cantidad de circuitos simples que lo componen.
5. Mejore la definición del método `ToString` para fechas de modo que el día y el mes aparezcan siempre con dos dígitos. Redefina también `ToString` para que el mes aparezca con el nombre en lugar de en dígitos.
6. Introduzca en la jerarquía de cuentas la clase `CuentaAhorro`. Una cuenta de ahorro debe incluir la tasa de interés que se aplica sobre el saldo y un método para calcular los intereses.
7. Modifique la clase `CuentaCredito` para permitir indicar, al instanciar cada objeto, el porcentaje de interés que se aplica al extraer y el crédito máximo que se puede conceder. Modifique el método `Extrae` para que aplique el interés solo sobre el crédito (la cantidad negativa). Es decir, que si la cuenta tiene saldo 100, y se extraen 200, el interés debe aplicarse solo sobre los 100 que se otorgan como crédito.
8. Intente rediseñar la jerarquía de cuentas, para poder tener un tipo que incluya la funcionalidad de `CuentaCredito` y `CuentaTransferencia`.
9. Implemente el método `ToString` en cada clase de la jerarquía de cuentas.
10. Implemente un método que reciba una colección de cuentas y las ordene de menor a mayor según el saldo.
11. De manera similar a la jerarquía de cuentas, diseñe e implemente una jerarquía de tarjetas. Note que cada tarjeta deberá tener asociada una `Cuenta`.
12. Diseñe e implemente una clase `CajeroAutomatico` que permita operar con objetos de la jerarquía de tarjetas.
13. Diseñe e implemente una jerarquía de productos, donde se tienen productos con un precio y productos con descuento. El precio de los productos con descuento es su precio base menos el descuento. Evite tener que modificar el método `Vende` de la clase `Tienda` (Listado 10-9) para poder vender productos de ambos tipos.

14. Añada a la clase `Tienda` una sobrecarga al método `Vende` que reciba una colección de productos y permita pagarlos todos a la vez. Intente reusar al máximo las operaciones definidas en la clase `Tienda`.
15. Implemente una clase `Biblioteca` que permita gestionar préstamos y devoluciones de libros. Los libros pueden ser impresos o electrónicos. El préstamo de libros impresos está limitado por la cantidad de ejemplares en existencia, mientras que no existen limitaciones para prestar libros electrónicos. Diseñe e implemente una jerarquía de libros que permita abstraerse del tipo de libro en la implementación de la clase `Biblioteca`.
16. Organice en una jerarquía *conjuntos* (sus elementos no se repiten) y *contenedores* (sus elementos pueden estar repetidos). Incluya operaciones para quitar una y para quitar todas las ocurrencias de un elemento. Trate de estratificar al máximo las características comunes.
17. Diseñe e implemente una jerarquía que permita evaluar expresiones aritméticas (suma, resta, multiplicación y división). Note que las expresiones se componen de operandos y operadores.
18. Implemente métodos extensores que considere de utilidad para el tipo `string` y el tipo `Point`.