UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN I

Unidad Temática N°3:
Herencia de Clases

Material Teórico
1° Año – 1° Cuatrimestre

Índice

JERARQUÍA DE CLASES

Las personas tendemos a asimilar nuevos conceptos basándonos en lo que ya conocemos. La herencia es una herramienta que permite definir nuevas clases en base a otras clases. La herencia es una relación entre clases, en la que una clase comparte la estructura y/o el comportamiento definidos en una (herencia simple) o más clases (herencia múltiple). La clase de la que otras heredan se denomina superclase, o clase padre. La clase que hereda de una o más clases se denomina subclase o clase hija.

La herencia permite que se puedan definir nuevas clases basadas de unas ya existentes a fin de reutilizar el código, generando así **una jerarquía de clases dentro de una aplicación**. Si una clase deriva de otra, esta hereda sus atributos y métodos y puede añadir nuevos atributos, métodos o redefinir los heredados.

No hay nada mejor en programación que poder usar el mismo código una y otra vez para hacer nuestro desarrollo más rápido y eficiente. El concepto de herencia ofrece mucho juego. Gracias a esto, lograremos un código mucho más limpio, estructurado y con menos líneas de código, lo que lo hace más legible.

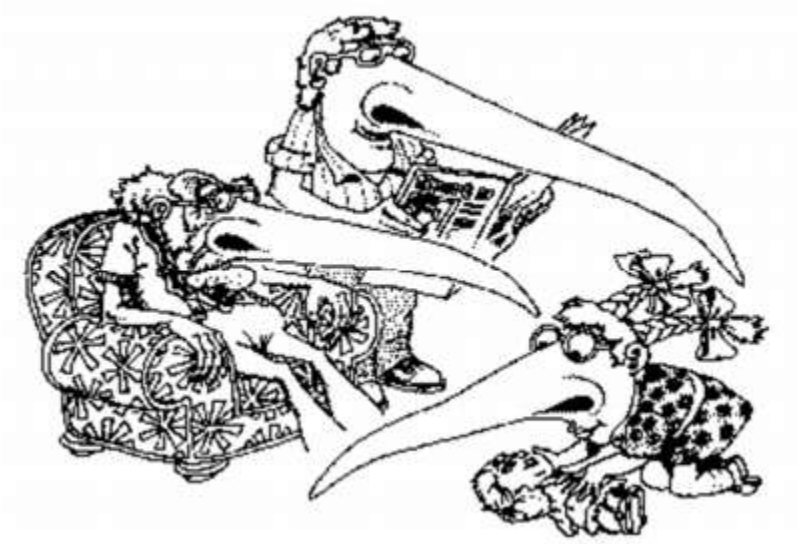


Imagen 1: Elaboración propia

Algunas de las clases tendrán instancias, es decir se crearán objetos a partir de ellas, y otras no. Las clases que tienen instancias se llaman concretas y las clases sin instancias son abstractas. Una clase abstracta se crea con la idea de que sus subclases añadan elementos a su estructura y comportamiento, usualmente completando la implementación de sus métodos (habitualmente) incompletos, con la intención de favorecer la reutilización

Herencia en C#

La herencia es uno de los conceptos más importantes de la Programación Orientada a Objetos (POO) y uno de los fundamentos sobre los cuales está construido C#. Herencia se refiere a la capacidad de crear clases que heredan ciertos aspectos de otras clases primarias. Todo el framework de .NET se basa en el concepto de Herencia y por esto en .NET "todo es un objeto". Incluso un simple número es una instancia de una clase que se hereda de la clase System.Object, aunque .NET haga un poco de trampa acá para facilitar la vida de los usuarios y por esto se puede asignar un número directamente, sin tener que crear una nueva instancia de cada número que se vaya a usar.

La sintaxis:

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

}
```

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

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

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

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

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

El siguiente ejemplo pone de manifiesto la herencia entre las clases Persona y Trabajador:

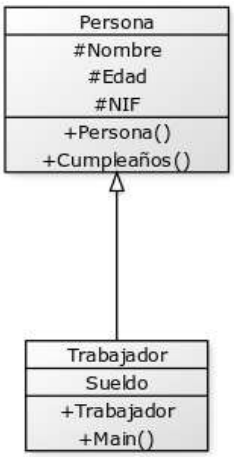


Gráfico 1: Elaboración propia

```
using System;

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

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

  // Constructor de Persona
  public Persona(string nombre, int edad, string nif)
  {
    Nombre = nombre;
    Edad = edad;        // notar que no se llaman igual
    NIF = nif;
  }
}

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

  public Trabajador(string nombre, int edad, string nif, int sueldo)
         : base(nombre, edad, nif)
  {// Inicializamos cada Trabajador en base al constructor de Persona
    Sueldo = sueldo;  // notar que no se llaman igual
  }

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

Llamadas por defecto al constructor base

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

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

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

}
```

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

Restricciones

Cuando una clase hereda de otra, la clase hija hereda los atributos y los métodos de la clase padre, pero existen ciertas restricciones en esta herencia:

- Los atributos y los métodos con modo de acceso private no se heredan. Se heredan los atributos y los métodos con modo de acceso public y protected.
- No se hereda un atributo de una clase padre si en la clase hija se define un atributo con el mismo nombre que el atributo de la clase padre.
- No se hereda un método si éste es sobrecargado.
- No se heredan los constructores de la clase base o madre. Por lo tanto en el constructor de la clase derivada, en la primera línea del mismo, hay que invocar al constructor de la clase base con la palabra **base** y la cantidad de parámetros que requiera el constructor, esto es así porque primero se debe crear la clase base y por último la clase derivada.

Para referenciar a los miembros de la clase base se utiliza la palabra reservada **base**, generalmente ésta se usa si los miembros de la clase base se llaman de la misma forma que los de la clase derivada, si no, no hace falta usar la palabra reservada.

La clase Object

Aparte de los tipos de los que puedan heredar mediante herencia única, todos los tipos del sistema de tipos .NET heredan implícitamente de **Object** o de un tipo

derivado de este. La funcionalidad común de **Object** está disponible para cualquier tipo incluso para los tipos primitivos.

Para ver lo que significa la herencia implícita, vamos a definir una nueva clase, *SimpleClass*, que es simplemente una definición de clase vacía:

```
public class SimpleClass
{

}
```

Aunque no se ha definido ningún miembro en la clase SimpleClass, si inspeccionamos la clase se vería que en realidad tiene nueve miembros. Uno de ellos es un constructor sin parámetros (o predeterminado) que el compilador de C# proporciona de manera automática para el tipo SimpleClass. Los ocho restantes son miembros de Object, el tipo del que heredan implícitamente a la larga todas las clases e interfaces del sistema de tipo .NET.

La herencia implícita desde la clase Object permite que estos métodos estén disponibles para la clase SimpleClass:

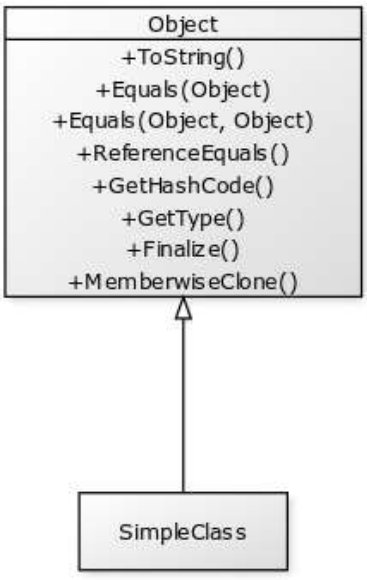


Gráfico 2: Elaboración propia

- El método público **ToString**, que convierte un objeto SimpleClass en su representación de cadena, devuelve el nombre de tipo completo. En este caso, el método **ToString** devuelve la cadena "SimpleClass".
- Tres métodos de prueba de igualdad de dos objetos: el método de instancia pública **Equals**(Object), el método público estático **Equals(**Object**,** Object**)** y el método público estático **ReferenceEquals(**Object**,** Object**).** De forma predeterminada, estos métodos prueban la igualdad de referencia; es decir, para que sean iguales, dos variables de objeto deben hacer referencia al mismo objeto.

- El método público **GetHashCode**, que calcula un valor que permite que una instancia del tipo se use en colecciones con hash.
- El método público **GetType**, que devuelve un objeto Type que representa el tipo SimpleClass.
- El método protegido **Finalize**, que está diseñado para liberar recursos no administrados antes de que el recolector de elementos no utilizados reclame la memoria de un objeto.
- El método protegido **MemberwiseClone**, que crea un clon superficial del objeto actual.

Clases abstractas

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

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

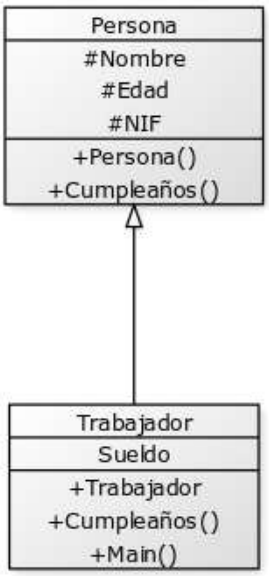
Las clases A y B del ejemplo son abstractas, y como puede verse es posible combinar en cualquier orden el modificador abstract con modificadores de acceso.

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

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

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

Ejemplo: retomaremos el caso anterior donde Trabajador hereda de Persona pero Persona será una clase abstracta que tendrá el método abstracto Cumpleaños() y redefiniremos una nueva versión de ese método para los objetos Trabajador.




```
abstract class Persona
{
  protected string Nombre;
  protected int Edad;
  protected string DNI;

  public abstract void Cumpleaños();

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

class Trabajador: Persona
{
  int Sueldo;

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

    Console.WriteLine (t.Cumpleaños());

  }
}
```

Nótese cómo se ha añadido el modificador abstract en la declaración de Cumpleaños() en la clase Persona, que también es abstract, para habilitar la posibilidad de que dicho método pueda ser redefinido en clases hijas de Persona y cómo se ha añadido override en la redefinición del mismo método dentro de la clase Trabajador para indicar una funcionalidad del método heredado de la clase base.

POLIMORFISMO

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

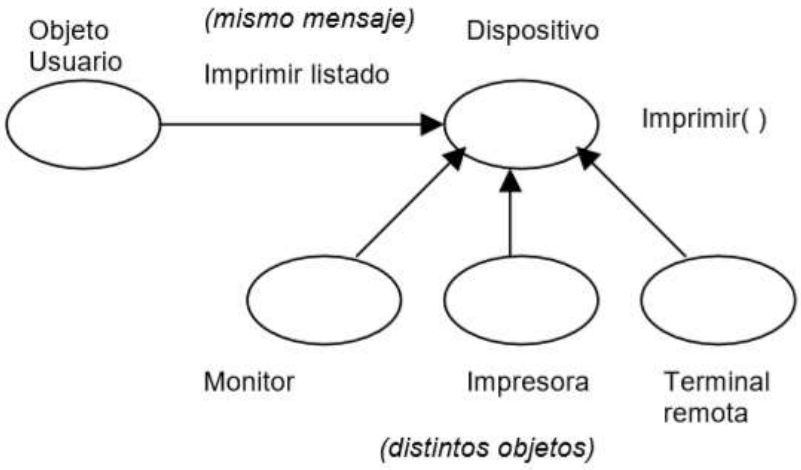


Gráfico 3: Elaboración propia

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

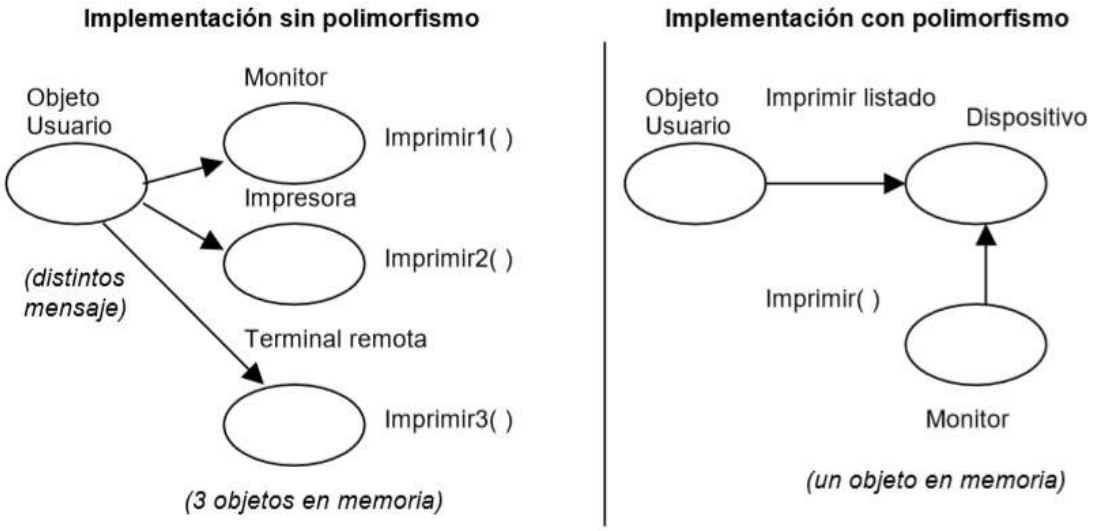


Gráfico 4: Elaboración propia

Otras definiciones

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

Un ejemplo: supongamos que en un programa debemos trabajar con figuras geométricas y construimos algunos objetos que son capaces de representar un rectángulo, un triángulo y un circulo. Si nos interesa que todos puedan dibujarse en la pantalla, podemos dotar a cada uno de un método llamado “dibujar” que se encargue de realizar esta tarea y entonces vamos a poder dibujar tanto un circulo, como un cuadrado o un triángulo simplemente enviándole al mensaje “dibujar” al objeto que queremos mostrar en la pantalla. En otras palabras, vamos a poder hacer X.dibujar() sin importar si X es un cuadrado un circulo o un triángulo. Los “circulo”, como los “triangulo” y los “cuadrado” son polimórficos respecto del mensaje dibujar. Es importante notar que no es lo mismo dibujar una figura que otra, seguramente, cada figura implementa ese método de modo diferente.

Otro ejemplo: supongamos que en un sistema de un banco tenemos objetos que representan cuentas corrientes y cajas de ahorro. Si bien son diferentes, en ambas pueden hacerse depósitos y extracciones, entonces podemos hacer que cada

objeto implemente un método “extraer” y uno “depositar” para que puedan responder a esos mismos mensajes. Sin duda, van a estar implementados de modo diferente porque no se hacen los mismos pasos para extraer o depositar dinero de una cuenta corriente que de una caja de ahorro, pero ambos objetos van a poder responder e esos mensajes y entonces serán polimorfitos entre sí respecto del conjunto de mensajes formado por “depositar” y “extraer”. La gran ventaja es que ahora podemos hacer X.depositar() y confiar en que se realizara un depósito en la cuenta que corresponde sin que tengamos que saber a priori si X es una cuenta corriente o una caja de ahorro.

Métodos virtuales

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

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

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

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

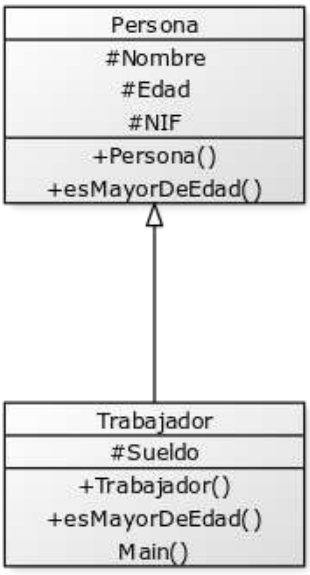
Nótese que esta posibilidad de cambiar el código de un método en su clase hija sólo se da si en la clase padre el método fue definido como **virtual**. En caso contrario, el compilador considerará un error intentar redefinirlo.

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

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

Para aclarar mejor el concepto de método virtual, vamos a retomar el ejemplo anterior, agregando la definición del método esMayorDeEdad() en los objetos Persona que muestra un mensaje si la edad supera los 18 años y redefiniremos ese método para los objetos Trabajador de modo que el mensaje sea mostrado si la edad supera los 21 años.

El código de este ejemplo es el que se muestra a continuación:




```
class Persona
{
  protected string Nombre;
  protected int Edad;
  protected string DNI;

  public virtual void esMayorDeEdad()
  {
    if (Edad>18)
    Console.WriteLine("Persona mayor de edad.");
  }
  // Constructor de Persona
  public Persona (string nombre, int edad, string dni)
  {
    Nombre = nombre;
    Edad = edad;
    DNI = dni;
  }
}

class Trabajador: Persona
{
  int Sueldo;

  public Trabajador(string nombre, int edad, string dni, int sueldo)
         : base(nombre, edad, dni)
  {   // Inicializamos cada Trabajador en base al constructor de Persona
    Sueldo = sueldo;
  }
  public override void esMayorDeEdad()
  {
    if (Edad>21)
    Console.WriteLine("Trabajador mayor de edad.");
  }
  public static void Main()
  {
    Persona p = new Persona("María", 20, "77588261-Z");
    Trabajador t = new Trabajador("Juan", 25, "77588260-Z", 100000);

    Console.WriteLine (t.esMayorDeEdad ());
```

```
        Console.WriteLine (p.esMayorDeEdad ());
    }
}
```

Nótese cómo se ha añadido el modificador virtual en la definición de esMayorDeEdad () en la clase Persona para habilitar la posibilidad de que dicho método puede ser redefinido en clase hijas de Persona y cómo se ha añadido override en la redefinición del mismo dentro de la clase Trabajador para indicar que la nueva definición del método es una redefinición del heredado de la clase base.

La salida de este programa confirma que la implementación de esMayorDeEdad() es distinta en cada clase, pues es de la forma:

```
Trabajador mayor de edad.
Persona mayor de edad.
```

Modificadores: sealed y static

Clases y métodos sellados

Cuando se aplica a una clase, el sealed modificador evita que otras clases hereden de ella. En el siguiente ejemplo, la clase B hereda de la clase A, pero ninguna clase puede heredar de la clase B.

```
class A {}
sealed class B : A {}
```

También puede usar el sealed modificador en un método o propiedad que anula un método o propiedad virtual en una clase base. Esto le permite permitir que las clases se deriven de su clase y evitar que anulen propiedades o métodos virtuales específicos.

Nota: Para determinar si sellar una clase, método o propiedad, generalmente debe considerar los siguientes dos puntos:

- Los beneficios potenciales que pueden obtener las clases derivadas a través de la capacidad de personalizar su clase.
- El potencial de que las clases derivadas puedan modificar sus clases de tal manera que ya no funcionen correctamente o como se esperaba.

Clases estáticas

Una clase **estática** es básicamente lo mismo que una clase no estática, pero hay una diferencia: no se puede crear una instancia de una clase estática. En otras palabras, no puede utilizar el operador new para crear una variable del tipo de clase. Debido a que no hay una variable de instancia, puede acceder a los miembros de una clase estática utilizando el nombre de la clase en sí.

Por ejemplo:

```
public static class ConvertidorTemperatura
{
}
```

Una clase estática se puede usar como un contenedor conveniente para conjuntos de métodos que solo operan en parámetros de entrada y no tienen que obtener o establecer ningún campo de instancia interno. Por ejemplo, en la biblioteca de clases .NET, la clase estática **System.Math** contiene métodos que realizan operaciones matemáticas, sin ningún requisito de almacenar o recuperar datos que son exclusivos de una instancia particular de la clase **Math**

La siguiente lista proporciona las características principales de una clase estática:

- Contiene solo miembros estáticos.
- No se puede crear una instancia.
- Está sellada.
- No puede contener constructores de instancias .

Miembros de clases estáticas

Una clase no estática puede contener métodos, campos, propiedades o eventos estáticos. El miembro estático es invocable en una clase incluso cuando no se ha creado ninguna instancia de la clase. Siempre se accede al miembro estático por el nombre de la clase, no por el nombre de la instancia. Solo existe una copia de un miembro estático, independientemente de cuántas instancias de la clase se creen. Los métodos y propiedades estáticos no pueden acceder a campos y eventos no estáticos en su tipo contenedor, y no pueden acceder a una variable de instancia de ningún objeto a menos que se pase explícitamente en un parámetro de método.

Es más típico declarar una clase no estática con algunos miembros estáticos que declarar una clase completa como estática. Dos usos comunes de los campos estáticos son llevar un recuento de la cantidad de objetos que se han instanciado o almacenar un valor que debe compartirse entre todas las instancias.

Los métodos estáticos se pueden sobrecargar, pero no anular, porque pertenecen a la clase y no a ninguna instancia de la clase.

Aunque un campo no se puede declarar como **static const**, un campo constante es esencialmente estático en su comportamiento. Pertenece al tipo, no a las instancias del tipo. Por lo tanto, **const** se puede acceder a los campos usando la

misma **ClassName.MemberName** notación que se usa para los campos estáticos. No se requiere ninguna instancia de objeto.

C# no admite variables locales estáticas (es decir, variables que se declaran en el ámbito del método).

Ejemplo: Supongamos que necesitamos una clase de utilidad que nos permita convertir temperaturas entre ºC y ºF.

```
public static class ConvertidorTemperatura
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        double celsius = Double.Parse(temperatureCelsius);
        double fahrenheit = (celsius * 9 / 5) + 32;
        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        double fahrenheit = Double.Parse(temperatureFahrenheit);
        double celsius = (fahrenheit - 32) * 5 / 9;
        return celsius;
    }
}
```

Manejo de excepciones

En C#, los errores en tiempo de ejecución se propagan a través del programa mediante un mecanismo denominado excepciones. Las excepciones las inicia el código que encuentra un error y las detecta el código que puede corregir dicho error. El entorno de ejecución .NET o el código de un programa pueden producir excepciones. Una vez iniciada, una excepción se propaga hasta la pila de llamadas hasta que encuentra una instrucción **catch** para la excepción.

Las excepciones están representadas por clases derivadas de **Exception**. Esta clase identifica el tipo de excepción y contiene propiedades que tienen los detalles sobre la excepción. Iniciar una excepción implica crear una instancia de una clase derivada de excepción, configurar opcionalmente las propiedades de la excepción y luego producir el objeto con la palabra clave **throw.**

Bloque try/catch/finally

Cuando se inicia una excepción, el entorno runtime comprueba la instrucción actual para ver si se encuentra dentro de un bloque **try**. Si es así, se comprueban los bloques catch asociados al bloque **try** para ver si pueden detectar la excepción. Los bloques **Catch** suelen especificar tipos de excepción; si el tipo del bloque catch es el mismo de la excepción, o una clase base de la excepción, el bloque **catch** puede controlar el método.

Por ejemplo:

```
try
{
    TestThrow();
}
catch (Exception ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

Una instrucción **try** puede contener más de un bloque **catch**. Se ejecuta la primera instrucción **catch** que pueda controlar la excepción; las instrucciones **catch** siguientes se omiten, aunque sean compatibles. Los bloques **catch** deben ordenarse siempre de más específico (o más derivado) a menos específico.

Para que el bloque **catch** se ejecute, el entorno runtime busca bloques **finally**. Los bloques **Finally** permiten al programador limpiar cualquier estado ambiguo que pudiera haber quedado tras la anulación de un bloque **try** o liberar los recursos externos (como identificadores de gráficos, conexiones de base de datos o flujos de archivos) sin tener que esperar a que el recolector de elementos no utilizados en el entorno de ejecución finalice los objetos.

Problema modelo

Supongamos que necesitamos desarrollar un formulario que permita convertir un valor de temperatura ingresado en una caja de texto. Para ello vamos a reutilizar la clase `ConvertidorTemperatura` mencionada anteriormente.

Con una interfaz gráfica como se muestra a continuación:

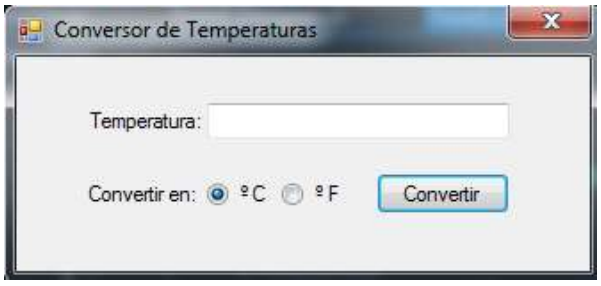


Imagen 2: Elaboración propia

```
namespace ConvertidorTemperatura
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            rbCelsius.Checked = true;
        }

        private void btnConvertir_Click(object sender, EventArgs e)
        {
            try
            {
                float temp = float.Parse(txtTemperatura.Text);
```

```
                    if (rbCelsius.Checked)
                    {
                        MessageBox.Show("Valor expresado en ºF: " +
TemperaturaConverter.CelsiusToFahrenheit(txtTemperatura.Text), "Convertidor");
                    }
                    else {
                        MessageBox.Show("Valor expresado en ºC: " +
TemperaturaConverter.FahrenheitToCelsius(txtTemperatura.Text), "Convertidor");

                    }
                }
                catch (FormatException fe)
                {
                    MessageBox.Show("Valor de temperatura incorrecto!", "Error");
                    txtTemperatura.Text = String.Empty;
                    txtTemperatura.Focus();
                }
                catch(Exception ex) {
                    MessageBox.Show("Error desconocido!. Intente nuevamente");
                }
            }
        }
    }
```

BIBLIOGRAFÍA

Bishop, P. (1992) Fundamentos de Informática. Anaya.

Brookshear, G. (1994) Computer Sciense: An Overview. Benjamin/Cummings.

De Miguel, P. (1994) Fundamentos de los Computadores. Paraninfo.

Joyanes, L. (1990) Problemas de Metodología de la Programación. McGraw Hill.

Joyanes, L. (1993) Fundamentos de Programación: Algoritmos y Estructura de Datos. McGraw Hill.

Norton, P. (1995) Introducción a la Computación. McGraw Hill.

Prieto, P. Lloris A. y Torres J.C. (1989) Introducción a la Informática. McGraw Hill.

Tucker, A. Bradley, W. Cupper, R y Garnick, D (1994) Fundamentos de Informática (Lógica, resolución de problemas, programas y computadoras). McGraw Hill.