# PROGRAMACIÓN MULTIMEDIA

# Y

# DISPOSITIVOS MÓVILES

## UT1. Programación Orientada a Objetos en C#

**Departamento de Informática y Comunicaciones**

**CFGS Desarrollo de Aplicaciones Multiplataforma**

**Segundo Curso**

**IES Virrey Morcillo**

**Villarrobledo (Albacete)**

# Índice


1. Programación Orientada a Objetos en C# ..... 3
2. Encapsulación ..... 3
   2.1. Miembros de una clase ..... 3
   2.2. Accesibilidad de los miembros de una clase ..... 4
3. Clases ..... 4
   3.1. Campos ..... 5
   3.2. Constantes ..... 6
   3.3. Propiedades ..... 6
      3.3.1. Propiedades con campos de respaldo ..... 7
      3.3.2. Definiciones de cuerpos de expresión ..... 8
      3.3.3. Propiedades autoimplementadas ..... 9
   3.4. Métodos ..... 9
      3.4.1. Tipos de parámetros ..... 11
      3.4.2. Cuerpo del método y variables locales ..... 12
      3.4.3. Valores devueltos ..... 12
   3.5. Constructores ..... 13
   3.6. Destructores ..... 14
4. Objetos o instancias de clases ..... 15
5. Herencia ..... 16
   5.1. Métodos abstractos y virtuales ..... 17
   5.2. Clases base abstractas ..... 17
   5.3. Interfaces ..... 17
6. Polimorfismo ..... 18
   6.1. Miembros virtuales ..... 20
   6.2. Sobrecarga de métodos ..... 22
7. Conjunto de parámetros de tipo ..... 22
8. Clases y métodos parciales ..... 23
9. Tipos anónimos ..... 24

# 1. Programación Orientada a Objetos en C#

El **lenguaje C#** proporciona compatibilidad completa para la programación orientada a objetos incluida la encapsulación, la herencia y el polimorfismo.

La encapsulación significa que un grupo de propiedades, métodos y otros miembros relacionados se tratan como una sola unidad u objeto.

La herencia describe la posibilidad de crear nuevas clases basadas en una clase existente.

El polimorfismo significa que puede tener múltiples clases que se pueden usar de manera intercambiable, aunque cada clase implementa las mismas propiedades o los mismos métodos de maneras diferentes.

# 2. Encapsulación

A veces se hace referencia a la encapsulación como el primer pilar o principio de la programación orientada a objetos. Según el **principio de encapsulación**, una clase o una estructura pueden especificar hasta qué punto se puede acceder a sus miembros para codificar fuera de la clase o la estructura.

No se prevé el uso de los métodos y las variables fuera de la clase, o el ensamblado puede ocultarse para limitar el potencial de los errores de codificación o de los ataques malintencionados.

## 2.1. Miembros de una clase

Todos los métodos, campos, constantes, propiedades y eventos deben declararse dentro de un tipo, se les denomina **miembros del tipo**.

En el lenguaje C# no hay métodos ni variables globales como en otros lenguajes, incluso el punto de entrada del programa, el método Main, debe declararse dentro de una clase.

Los miembros de una clase son miembros estáticos o miembros de instancia. Los **miembros estáticos** pertenecen a clases y los **miembros de instancia** pertenecen a objetos o instancias de clases.

A continuación, se muestran los tipos de miembros que puede contener una clase:

- **Constantes**: valores constantes asociados a la clase.
- **Campos**: variables de la clase.
- **Métodos**: cálculos y acciones que pueden realizarse mediante la clase.
- **Propiedades**: acciones asociadas a la lectura y escritura de propiedades con nombre de la clase.
- **Indizadores**: acciones asociadas a la indexación de instancias de la clase como una matriz.
- **Eventos**: notificaciones que puede generar la clase.
- **Operadores**: conversiones y operadores de expresión admitidos por la clase.
- **Constructores**: acciones necesarias para inicializar instancias de la clase o la clase propiamente dicha.

- **Destructores**: acciones que deben realizarse antes de que las instancias de la clase se descarten de forma permanente.
- **Tipos**: tipos anidados declarados por la clase.

## 2.2. Accesibilidad de los miembros de una clase

Cada miembro de una clase tiene asociada una accesibilidad, que controla las regiones del texto del programa que pueden tener acceso al miembro.

Existen seis formas de accesibilidad posibles, se resumen a continuación:

- **public**: acceso no limitado.
- **protected**: acceso limitado a esta clase o a las clases derivadas de esta clase.
- **internal**: acceso limitado al ensamblado actual (.exe, .dll, etc.).
- **protected internal**: acceso limitado a la clase contenedora, las clases derivadas de la clase contenedora, o bien las clases dentro del mismo ensamblado.
- **private**: acceso limitado a esta clase.
- **private protected**: acceso limitado a la clase contenedora o las clases derivadas del tipo contenedor con el mismo ensamblado.

# 3. Clases

Las clases son los tipos más fundamentales de C#. Una **clase** es una estructura de datos que combina estados (campos) y acciones (métodos y otros miembros de función) en una sola unidad. Una clase proporciona una definición para instancias creadas dinámicamente de la clase, también conocidas como objetos.

Un tipo que se define como una clase, es un **tipo de referencia**. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor null hasta que se cree expresamente una instancia de la clase mediante el operador new o se le asigne un objeto de un tipo compatible que se ha creado en otro lugar.

Cuando se crea el objeto, se asigna suficiente memoria en el montón (heap) administrado para ese objeto específico y la variable solo contiene una referencia a la ubicación de dicho objeto. Los tipos del montón administrado producen sobrecarga cuando se asignan y cuando los reclama la función de administración de memoria automática de CLR, conocida como recolección de elementos no utilizados.

Las clases se crean mediante **declaraciones de clase**. Una declaración de clase se inicia con un encabezado que especifica los atributos y modificadores de la clase, el nombre de la clase, la clase base, si se indica, y las interfaces implementadas por la clase. Al encabezado le sigue el cuerpo de la clase, que consta de una lista de declaraciones de miembros escritas entre los delimitadores { y }.

La siguiente es una declaración de una clase simple denominada Point:

```
public class Point {

   public int x, y;

   public Point(int x, int y) {
          this.x = x;
          this.y = y;
```

```
    }

}
```

## 3.1. Campos

Los **campos** almacenan habitualmente los datos que deben ser accesibles para más de un método de clase y que deben almacenarse durante más tiempo de lo que dura un único método. Un campo es, en definitiva, una variable que está asociada con una clase o a una instancia de una clase.

Una clase puede tener campos de instancia, campos estáticos o ambos.

Los **campos de instancia** son específicos de una instancia de un tipo. Si tiene una clase T, con un campo de instancia F, puede crear dos objetos de tipo T y modificar el valor de F en cada objeto sin afectar el valor del otro objeto.

Por el contrario, los **campos estáticos** pertenecen a la propia clase y se comparte entre todas las instancias de esa clase. Los cambios realizados desde la instancia A serán visibles inmediatamente para las instancias B y C, si tienen acceso al campo. Un campo declarado con el modificador **static** define un campo estático.

Los campos se declaran en el bloque de clase especificando el nivel de acceso del campo, seguido por el tipo del campo, seguido por el nombre del campo.

En el ejemplo siguiente, cada instancia de la clase Color tiene una copia independiente de los campos de instancia r, g y b, pero solo hay una copia de los campos estáticos Black, White, Red, Green y Blue:

```
public class Color {

    public static readonly Color Black = new Color(0, 0, 0);

    public static readonly Color White = new Color(255, 255, 255);

    public static readonly Color Red = new Color(255, 0, 0);

    public static readonly Color Green = new Color(0, 255, 0);

    public static readonly Color Blue = new Color(0, 0, 255);

    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

}
```

Como se muestra en el ejemplo anterior, los campos de solo lectura se pueden declarar con un modificador readonly. La asignación a un campo readonly solo se puede producir como parte de la declaración del campo o en un constructor de la misma clase.

Por lo general, solo se deben usar campos para las variables que tienen accesibilidad privada o protegida. Los datos que la clase expone al código de cliente deben proporcionarse a través de métodos, propiedades e indizadores.

## 3.2. Constantes

Las **constantes** son valores inmutables que se conocen en tiempo de compilación y que no cambian durante la ejecución del programa.

Las constantes se declaran con el modificador **const** y se deben inicializar al declararse.

Sólo los tipos integrados de C#, excluido System.Object, pueden declararse como constantes:

| | |
|---|---|
| bool | System.Boolean |
| byte | System.Byte |
| sbyte | System.SByte |
| char | System.Char |
| decimal | System.Decimal |
| double | System.Double |
| float | System.Single |
| int | System.Int32 |

| | |
|---|---|
| uint | System.UInt32 |
| long | System.Int64 |
| ulong | System.UInt64 |
| object | System.Object |
| short | System.Int16 |
| ushort | System.UInt16 |
| string | System.String |

Los tipos definidos por el usuario, incluidas las clases, las estructuras y las matrices, no pueden ser constantes.

El lenguaje C# no admite métodos, propiedades ni eventos constantes.

El tipo enum permite definir constantes con nombre para los tipos integrados enteros, por ejemplo: int, uint, long, etc.

En este ejemplo la constante MESES siempre es 12 y ni siquiera la propia clase la puede cambiar. De hecho, cuando el compilador detecta un identificador de constante en el código fuente de C#, sustituye directamente el valor literal en el código de lenguaje intermedio que genera:

```
public const int MESES = 12;
```

## 3.3. Propiedades

Las **propiedades** son miembros que proporcionan un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados **descriptores de acceso**. Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos.

Las propiedades son una extensión natural de los campos. Ambos son miembros con nombre con tipos asociados y la sintaxis para acceder a los campos y las propiedades es la misma. Sin embargo, a diferencia de los campos, las propiedades no denotan ubicaciones de almacenamiento. Las propiedades tienen descriptores de acceso que especifican las instrucciones que se ejecutan cuando se leen o escriben sus valores.

Las propiedades permiten que una clase exponga una manera pública de obtener y establecer valores, a la vez que se oculta el código de implementación o verificación.

Para devolver el valor de la propiedad se usa un descriptor de acceso de propiedad **get**, mientras que para asignar un nuevo valor se emplea un descriptor de acceso de propiedad **set**. Estos descriptores de acceso pueden tener diferentes niveles de acceso.

Una propiedad se declara como un campo, salvo que la declaración finaliza con un descriptor de acceso **get** y un descriptor de acceso **set** escrito entre los delimitadores { y } en lugar de finalizar en un punto y coma.

La palabra clave **value** se usa para definir el valor que va a asignar el descriptor de acceso set.

Las propiedades pueden ser:

- ✓ de **lectura y escritura**: en ambos casos tienen un descriptor de acceso get y set.
- ✓ de **solo lectura**: tienen un descriptor de acceso get, pero no set.
- ✓ de **solo escritura**: tienen un descriptor de acceso set, pero no get.

### 3.3.1. Propiedades con campos de respaldo

Un **patrón básico** para implementar una propiedad conlleva el uso de un campo de respaldo privado para establecer y recuperar el valor de la propiedad. El descriptor de acceso **get** devuelve el valor del campo privado y el descriptor de acceso **set** puede realizar alguna validación de datos antes de asignar un valor al campo privado. Ambos descriptores de acceso además pueden realizar algún cálculo o conversión en los datos antes de que se almacenen o se devuelvan.

En este ejemplo, la clase TimePeriod representa un intervalo de tiempo. Internamente, la clase almacena el intervalo de tiempo en segundos en un campo privado denominado _seconds. Una propiedad de lectura y escritura denominada Hours permite al cliente especificar el intervalo de tiempo en horas. Los descriptores de acceso get y set realizan la conversión necesaria entre horas y segundos. Además, el descriptor de acceso set valida los datos e inicia una excepción ArgumentOutOfRangeException si el número de horas no es válido:

```
using System;

class TimePeriod {
    private double _seconds;

    public double Hours {
        get { return _seconds / 3600; }
        set {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(
                      $"{nameof(value)} must be between 0 and 24.");

            _seconds = value * 3600;
        }
    }
}

class Program {
    static void Main() {
        TimePeriod t = new TimePeriod();
        // The property assignment causes the 'set' accessor to be called.
        t.Hours = 24;
```

```
        // Retrieving the property causes the 'get' accessor to be called.
        Console.WriteLine($"Time in hours: {t.Hours}");
    }
}
// The example displays the following output:
//    Time in hours: 24
```

## 3.3.2. Definiciones de cuerpos de expresión

Las **definiciones de cuerpos de expresión** constan del símbolo **=>** seguido de la expresión que se va a asignar a la propiedad o a recuperar de ella. A partir de C# 6, las propiedades de solo lectura pueden implementar el descriptor de acceso get como miembro con forma de expresión. En este caso, no se usan ni la palabra clave del descriptor de acceso get ni la palabra clave return.

En el ejemplo siguiente se implementa la propiedad de solo lectura Name como miembro con forma de expresión:

```
using System;

public class Person {
    private string _firstName;
    private string _lastName;

    public Person(string first, string last) {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}

public class Example {
    public static void Main() {
        var person = new Person("Isabelle", "Butts");
        Console.WriteLine(person.Name);
    }
}
// The example displays the following output:
//       Isabelle Butts
```

A partir de C# 7.0, los descriptores de acceso get y set se pueden implementar como miembros con forma de expresión. En este caso, las palabras clave get y set deben estar presentes. En el ejemplo siguiente se muestra el uso de definiciones de cuerpos de expresión para ambos descriptores de acceso. Observe que no se usa la palabra clave return con el descriptor de acceso get:

```
using System;

public class SaleItem {
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost) {
        _name = name;
        _cost = cost;
    }
```

```
    public string Name {
        get => _name;
        set => _name = value;
    }

    public decimal Price {
        get => _cost;
        set => _cost = value;
    }
}

class Program {
    static void Main(string[] args) {
        var item = new SaleItem("Shoes", 19.95m);
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

### 3.3.3. Propiedades autoimplementadas

En algunos casos, los descriptores de acceso de propiedad get y set simplemente asignan un valor a un campo de respaldo o recuperan un valor de él sin incluir ninguna lógica adicional. Mediante las **propiedades implementadas automáticamente**, se puede simplificar el código y conseguir que el compilador de C# le proporcione el campo de respaldo de forma transparente.

Si una propiedad tiene un descriptor de acceso get y set, ambos deben ser implementados automáticamente. Una propiedad implementada automáticamente se define mediante las palabras clave get y set sin proporcionar ninguna implementación. El ejemplo siguiente repite el anterior, salvo que Name y Price son propiedades implementadas automáticamente:

```
using System;

public class SaleItem {
    public string Name { get; set; }
    public decimal Price { get; set; }
}

class Program {
    static void Main(string[] args) {
        var item = new SaleItem { Name = "Shoes", Price = 19.95m };
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

## 3.4. Métodos

Los **métodos** son miembros que implementan un cálculo o una acción que puede realizar un objeto o una clase. Un método es un bloque de código que contiene una serie de instrucciones. Un programa hace que se ejecuten las instrucciones al llamar al método y especificando los argumentos de método necesarios. En C#, todas las instrucciones ejecutadas se realizan en el contexto de un

método. El método Main es el punto de entrada para cada aplicación de C# y se llama mediante Common Language Runtime (CLR) cuando se inicia el programa.

Los **métodos se declaran** en una clase especificando el nivel de acceso, como public o private, modificadores opcionales como abstract o sealed, el valor de retorno, el nombre del método y los parámetros del método. Todas estas partes forman la **firma o prototipo** del método.

Los parámetros de método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere parámetros. El tipo de valor devuelto de un método es void si no se devuelve un valor.

En el siguiente ejemplo se observa una clase abstracta que contiene cuatro métodos:

```
abstract class Motorcycle {
    // Anyone can call this.
    public void StartEngine() {
        /* Method statements here */
    }

    // Only derived classes can call this.
    protected void AddGas(int gallons) {
        /* Method statements here */
    }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) {
        /* Method statements here */
        return 1;
    }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Los **métodos se llaman** de forma similar a como se accede a un campo. Después del nombre del objeto, se agrega un punto, el nombre del método y entre paréntesis se enumeran los argumentos separados por comas.

Los métodos de la clase Motorcycle declarada anteriormente se pueden llamar como en el ejemplo siguiente:

```
class TestMotorcycle : Motorcycle {

    public override double GetTopSpeed() {
        return 108.4;
    }

    static void Main() {
        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

### 3.4.1. Tipos de parámetros

Hay tres tipos de parámetros:

- ✓ parámetros de valor,
- ✓ parámetros de referencia y
- ✓ parámetros de salida.

Un **parámetro de valor** se usa para pasar argumentos de entrada. Un parámetro de valor corresponde a una variable local que obtiene su valor inicial del argumento que se ha pasado para el parámetro. Las modificaciones en un parámetro de valor no afectan el argumento que se pasa para el parámetro.

De forma predeterminada, cuando un tipo de valor se pasa a un método, se pasa una copia en lugar del propio objeto. Por lo tanto, los cambios realizados en el argumento no tienen ningún efecto en la copia original del método de llamada.

Un **parámetro de referencia** se utiliza para pasar como argumento una referencia al objeto, es decir, su dirección. Así, el método no recibe el objeto concreto, recibe un argumento que indica la ubicación del objeto. Si cambia un miembro del objeto mediante esta referencia, el cambio se refleja en el argumento del método de llamada, incluso si pasa el objeto por valor. Un parámetro de referencia se declara con el modificador **ref**.

En el ejemplo siguiente se muestra el uso de parámetros de referencia:

```
class Example {

    static void Intercambio(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    public static void Example() {
        int i = 1, j = 2;

        Intercambio (ref i, ref j);
        Console.WriteLine($"{i} {j}");
        // Muestra "2 1"
    }

}
```

Un **parámetro de salida** se usa para pasar argumentos mediante una referencia. Es similar a un parámetro de referencia, excepto que no necesita que asigne un valor explícitamente al argumento proporcionado por el autor de la llamada. Un parámetro de salida se declara con el modificador **out**. En el siguiente ejemplo se muestra el uso de los parámetros out con la sintaxis que se ha presentado en C# 7:

```
class OutExample {

    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }
```

```
    public static void OutUsage() {
        Divide(10, 3, out int res, out int rem);
        Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
    }

}
```

### 3.4.2. Cuerpo del método y variables locales

En el **cuerpo de un método** se especifican las instrucciones que se ejecutarán cuando se invoca el método en cuestión. Dentro de éste se pueden declarar variables que son específicas de la invocación del método. Estas variables se denominan **variables locales**. Una declaración de una variable local especifica un nombre de tipo, un nombre de variable y, posiblemente, un valor inicial.

En el ejemplo siguiente se declara una variable local i con un valor inicial de cero y una variable local j sin ningún valor inicial.

```
class Squares {

    public static void WriteSquares() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }

}
```

El lenguaje C# requiere que se asigne definitivamente una variable local antes de que se pueda obtener su valor. Por ejemplo, si la declaración de i anterior no incluyera un valor inicial, el compilador notificaría un error con los usos posteriores de i porque i no se asignaría definitivamente en esos puntos del programa.

### 3.4.3. Valores devueltos

Los métodos pueden devolver un valor al autor de llamada. Si el tipo de valor devuelto no es void, el método puede devolver el valor mediante la palabra clave **return**.

Una instrucción con la palabra clave return seguida de una variable, una constante o una expresión que coincide con el tipo de valor devuelto, devolverá este valor al autor de la llamada al método. La palabra clave return también detiene la ejecución del método.

Si el tipo de valor devuelto es void, una instrucción return sin un valor también es útil para detener la ejecución del método. Sin la palabra clave return, el método dejará de ejecutarse cuando alcance el final del bloque de código.

Por ejemplo, estos dos métodos utilizan la palabra clave return para devolver enteros:

```
class SimpleMath {
    public int AddTwoNumbers(int number1, int number2) {
        int suma;
        suma = number1 + number2;
```

```
        return suma;
    }

    public int SquareANumber(int number) {
        return number * number;
    }
}
```

Para utilizar un valor devuelto de un método, el método de llamada se puede usar la llamada al método en cualquier lugar, un valor del mismo tipo sería suficiente. También se puede asignar el valor devuelto a una variable.

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

## 3.5. Constructores

Los **constructores** permiten al programador establecer valores predeterminados, limitar la creación de instancias y escribir código flexible y fácil de leer. Cada vez que se crea una clase, se llama a su constructor. Una clase puede tener varios constructores que toman argumentos diferentes.

Si no proporciona un constructor para la clase, C# creará uno de manera predeterminada que cree instancias del objeto y establezca las variables miembros en los valores predeterminados que se indican:

| | |
|---|---|
| bool | `false` |
| byte | 0 |
| char | '\0' |
| decimal | 0M |
| double | 0.0D |
| enum | Valor generado por la expresión `(E)0`, donde `E` es el identificador de enumeración. |
| float | 0.0F |
| int | 0 |
| long | 0L |
| sbyte | 0 |
| short | 0 |
| struct | El valor generado al establecer todos los campos de tipo de valor en sus valores predeterminados y todos los campos de tipo de referencia en `null`. |
| uint | 0 |
| ulong | 0 |
| ushort | 0 |

Un constructor es un método cuyo nombre es igual que el nombre de su tipo. Su firma del método incluye solo el nombre del método y su lista de parámetros; no incluye un tipo de valor devuelto.

En el ejemplo siguiente se muestra el constructor de una clase denominada Person:

```
public class Person {
    private string last;
    private string first;

    public Person(string lastName, string firstName) {
        last = lastName;
        first = firstName;
    }
}
```

Si un constructor puede implementarse como una instrucción única, puede usarse una definición del cuerpo de expresión.

En el ejemplo siguiente se define una clase Location cuyo constructor tiene un único parámetro de cadena denominado name. La definición del cuerpo de expresión asigna el argumento al campo locationName:

```
public class Location {
    private string locationName;

    public Location(string name) => Name = name;

    public string Name {
        get => locationName;
        set => locationName = value;
    }
}
```

El lenguaje C# admite los siguientes tipos de constructores:

- ✓ Un **constructor de instancia** es un miembro que implementa las acciones necesarias para inicializar una instancia de una clase.
- ✓ Un **constructor estático** es un miembro que implementa las acciones necesarias para inicializar una clase en sí misma cuando se carga por primera vez.

Un constructor se declara como un método sin ningún tipo de valor devuelto y el mismo nombre que la clase contenedora. Si una declaración de constructor incluye un modificador **static**, declara un constructor estático. De lo contrario, declara un constructor de instancia.

En los ejemplos anteriores se han mostrado constructores de instancia, que crean un objeto nuevo.

Una clase también puede tener un constructor estático, que inicializa los miembros estáticos del tipo. Los constructores estáticos no tienen parámetros. Si no proporciona un constructor estático para inicializar los campos estáticos, el compilador de C# proporcionará un constructor estático predeterminado que inicializa los campos estáticos en su valor predeterminado.

En el ejemplo siguiente se usa un constructor estático para inicializar un campo estático:

```
public class Adult : Person {
    private static int minimumAge;

    public Adult(string lastName, string firstName) : base(lastName, firstName) { }

    static Adult() {
        minimumAge = 18;
    }
}
```

## 3.6. Destructores

Los **destructores**, también denominados **finalizadores**, se usan para realizar cualquier limpieza final necesaria cuando el recolector de elementos no utilizados recopila una instancia de clase.

Una clase solo puede tener un finalizador. Los finalizadores no se pueden heredar ni sobrecargar. No se puede llamar a los finalizadores. Se invocan automáticamente. Un finalizador no permite modificadores ni tiene parámetros.

Por ejemplo, el siguiente código muestra una declaración de un finalizador para la clase Car:

```
class Car {
    ~Car() {
        // cleanup statements...
        Console.WriteLine($"The {ToString()} destructor is executing.");
    }
}
```

## 4. Objetos o instancias de clases

Una definición de clase o estructura es como un plano que especifica qué puede hacer el tipo. Un **objeto** es básicamente un bloque de memoria que se ha asignado y configurado de acuerdo con el plano. Un programa puede crear muchos objetos de la misma clase. Los objetos también se denominan **instancias** y pueden almacenarse en una variable con nombre, o en una matriz o colección.

El código de cliente es el código que usa estas variables para llamar a los métodos y acceder a las propiedades públicas del objeto. En un lenguaje orientado a objetos, como C#, un programa típico consta de varios objetos que interactúan dinámicamente.

Puesto que las clases son tipos de referencia, una variable de un objeto de clase contiene una referencia a la dirección del objeto del montón administrado. Si se asigna un segundo objeto del mismo tipo al primer objeto, ambas variables hacen referencia al objeto de esa dirección.

Las **instancias de clases** se crean mediante el operador **new**, que asigna memoria para una nueva instancia, invoca un constructor para inicializar la instancia y devuelve una referencia a la instancia. Las instrucciones siguientes crean dos objetos Point y almacenan las referencias en esos objetos en dos variables p1 y p2:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

La memoria ocupada por un objeto se libera automáticamente cuando el objeto ya no es accesible. No es necesario ni posible desasignar explícitamente objetos en C#.

Cuando se **comparan dos objetos** para comprobar si son iguales, primero debe determinar si quiere saber si las dos variables representan el mismo objeto en la memoria o si los valores de uno o varios de sus campos son equivalentes:

- ✓ Para determinar si dos instancias de clase hacen referencia a la misma ubicación en la memoria, lo que significa que tienen la misma identidad, se utiliza el método estático **Equals**.
- ✓ Para determinar si los campos de instancia de dos instancias de estructura presentan los mismos valores, se usa el método **ValueType.Equals**.
- ✓ Para determinar si los valores de los campos de dos instancias de clase son iguales, se puede usar el método **Equals** o el operador **==**.

Por ejemplo:

```
public struct Person {
    public string Name;
    public int Age;

    public Person(string name, int age) {
        Name = name;
        Age = age;
    }
}
```

```
Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

## 5. Herencia

La **herencia** permite crear clases nuevas que reutilizan, extienden y modifican el comportamiento que se define en otras clases. La clase cuyos miembros se heredan se denomina **clase base** y la clase que hereda esos miembros se denomina **clase derivada**.

Una clase derivada solo puede tener una clase base directa, pero la herencia es transitiva. Si ClaseC se deriva de ClaseB y ClaseB se deriva de ClaseA, ClaseC hereda los miembros declarados en ClaseB y ClaseA.

Conceptualmente, una clase derivada es una especialización de la clase base. Por ejemplo, si tiene una clase base Animal, podría tener una clase derivada denominada Mammal y otra clase derivada denominada Reptile. Mammal es Animal y Reptile también es Animal, pero cada clase derivada representa especializaciones diferentes de la clase base.

Cuando se define una clase para que derive de otra clase, la clase derivada obtiene implícitamente todos los miembros de la clase base, salvo sus constructores y sus destructores. La clase derivada puede reutilizar el código de la clase base sin tener que volver a implementarlo. Puede agregar más miembros en la clase derivada. De esta manera, la clase derivada amplía la funcionalidad de la clase base.

En el ejemplo siguiente, la clase base de Point3D es Point y la clase base de Point es object:

```
public class Point {
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D : Point {
    public int z;

    public Point3D(int x, int y, int z) : base(x, y) {
        this.z = z;
    }
}
```

Una clase hereda a los miembros de su clase base. La herencia significa que una clase contiene implícitamente todos los miembros de su clase base, excepto la instancia y los constructores estáticos, y los finalizadores de la clase base. Una clase derivada puede agregar nuevos miembros a aquellos de

los que hereda, pero no puede quitar la definición de un miembro heredado. En el ejemplo anterior, Point3D hereda los campos x e y de Point y cada instancia de Point3D contiene tres campos: x, y y z.

Existe una conversión implícita de un tipo de clase a cualquiera de sus tipos de clase base. Por lo tanto, una variable de un tipo de clase puede hacer referencia a una instancia de esa clase o a una instancia de cualquier clase derivada.

Por ejemplo, dadas las declaraciones de clase anteriores, una variable de tipo Point puede hacer referencia a una instancia de Point o Point3D:

```
Point p = new Point(10, 20);
Point p3d = new Point3D(10, 20, 30);
```

## 5.1. Métodos abstractos y virtuales

Cuando una clase base declara un método como **virtual**, una clase derivada puede reemplazar el método con su propia implementación.

Si una clase base declara un miembro como **abstracto**, ese método se debe reemplazar en todas las clases no abstractas que hereden directamente de dicha clase.

Si una clase derivada es abstracta, hereda los miembros abstractos sin implementarlos. Los miembros abstractos y virtuales son la base del polimorfismo, que es la segunda característica principal de la programación orientada a objetos.

## 5.2. Clases base abstractas

Se puede declarar una clase como **abstracta** para impedir la creación directa de instancias mediante la palabra clave new. Si se hace, la clase solo se puede usar si se deriva de ella una clase nueva.

Una clase abstracta puede contener una o más firmas o prototipos de método que, a su vez, se declaran como abstractas. Estas firmas especifican los parámetros y el valor devuelto, pero no tienen ninguna implementación, cuerpo del método.

Una clase abstracta no tiene que contener miembros abstractos, pero si lo hace, la clase debe declararse como abstracta.

Las clases derivadas que no son abstractas deben proporcionar la implementación para todos los métodos abstractos de una clase base abstracta.

## 5.3. Interfaces

Una **interfaz** es un tipo de referencia similar a una clase base abstracta formada únicamente por miembros abstractos.

Cuando una clase implementa una interfaz, debe proporcionar una implementación para todos los miembros de la interfaz. Una clase puede implementar varias interfaces, aunque solo puede derivar de una única clase base directa.

Las interfaces se usan para definir funciones específicas para clases que no tienen necesariamente una relación "es un/una".

# 6. Polimorfismo

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después de la encapsulación y la herencia. **Polimorfismo** es una palabra griega que significa "con muchas formas" y tiene dos aspectos diferentes:

- ✓ En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos y colecciones o matrices. Cuando esto ocurre, el tipo declarado del objeto ya no es idéntico a su tipo en tiempo de ejecución.
- ✓ Las clases base pueden definir e implementar métodos virtuales, y las clases derivadas pueden invalidarlos, lo que significa que pueden proporcionar su propia definición e implementación. En tiempo de ejecución, cuando el código de cliente llama al método, CLR busca el tipo en tiempo de ejecución del objeto e invoca esa invalidación del método virtual. Por lo tanto, en el código fuente puede llamar a un método en una clase base y hacer que se ejecute una versión del método de la clase derivada.

Los **métodos virtuales** permiten trabajar con grupos de objetos relacionados de manera uniforme. Por ejemplo, supongamos que tiene una aplicación de dibujo que permite a un usuario crear varios tipos de formas en una superficie de dibujo. En tiempo de compilación, no sabe qué tipos específicos de formas creará el usuario. Sin embargo, la aplicación tiene que realizar el seguimiento de los distintos tipos de formas que se crean, y tiene que actualizarlos en respuesta a las acciones del mouse del usuario.

Para solucionar este problema en dos pasos básicos, se puede usar el polimorfismo:

1. Crear una jerarquía de clases en la que cada clase de forma específica deriva de una clase base común.
2. Usar un método virtual para invocar el método apropiado en una clase derivada mediante una sola llamada al método de la clase base.

El siguiente ejemplo muestra todo lo expuesto anteriormente:

```
using System;
using System.Collections.Generic;

public class Shape {
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw() {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape {
    public override void Draw() {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
```

```
}

class Rectangle : Shape {
    public override void Draw() {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}


class Triangle : Shape {
    public override void Draw() {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

class Program {
    static void Main(string[] args) {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used whereever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        var shapes = new List<Shape>
        {
            new Rectangle(),
            new Triangle(),
            new Circle()
        };

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (var shape in shapes) {
            shape.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

Primero, se crea una clase base llamada Shape y clases derivadas como Rectangle, Circle y Triangle. Se da a la clase Shape un método virtual llamado Draw y se invalida en cada clase derivada para dibujar la forma determinada que la clase representa. Se crea un objeto List<Shape> y se agrega Circle, Triangle y Rectangle a él. Para actualizar la superficie de dibujo, se usa un bucle foreach para iterar por la lista y llamar al método Draw en cada objeto Shape de la lista. Aunque cada objeto de la lista tenga un tipo declarado de Shape, se invocará el tipo en tiempo de ejecución, la versión invalidada del método en cada clase derivada.

## 6.1. Miembros virtuales

Cuando una clase derivada hereda de una clase base, obtiene todos los métodos, campos, propiedades y eventos de la clase base. El diseñador de la clase derivada tiene las siguientes opciones:

- Invalidar los miembros virtuales de la clase base.
- Heredar el método de la clase base más próximo sin invalidarlo.
- Definir una nueva implementación no virtual de esos miembros que oculte las implementaciones de la clase base.

Una clase derivada puede invalidar un miembro de la clase base si este se declara como virtual o abstracto. El miembro derivado debe usar la palabra clave override para indicar explícitamente que el propósito del método es participar en una invocación virtual.

El siguiente fragmento de código muestra un ejemplo:

```
public class BaseClass {
    public virtual void DoWork() { }
    public virtual int WorkProperty {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass {
    public override void DoWork() { }
    public override int WorkProperty {
        get { return 0; }
    }
}
```

Los campos no pueden ser virtuales; solo los métodos, propiedades, eventos e indizadores pueden ser virtuales. Cuando una clase derivada invalida un miembro virtual, se llama a ese miembro aunque se acceda a una instancia de esa clase como una instancia de la clase base. El siguiente fragmento de código muestra un ejemplo:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Los métodos y propiedades virtuales permiten a las clases derivadas extender una clase base sin necesidad de usar la implementación de clase base de un método.

Cuando una declaración de método de instancia incluye un modificador virtual, se dice que el método es un **método virtual**. Cuando no existe un modificador **virtual**, se dice que el método es un método no virtual.

Cuando se invoca un método virtual, el tipo en tiempo de ejecución de la instancia para la que tiene lugar esa invocación determina la implementación del método real que se invocará. En una invocación de método no virtual, el tipo en tiempo de compilación de la instancia es el factor determinante.

Por ejemplo, cualquier clase que herede este método puede reemplazarlo:

```
public virtual double Area() {
    return x * y;
}
```

Un método de reemplazo de una clase derivada puede modificar la implementación de un miembro virtual.

Un método virtual puede ser un **método de reemplazado** en una clase derivada. Cuando una declaración de método de instancia incluye un modificador **override**, el método reemplaza un método virtual heredado con la misma signatura o prototipo. Mientras que una declaración de método virtual introduce un método nuevo, una declaración de método de reemplazo especializa un método virtual heredado existente proporcionando una nueva implementación de ese método.

En este ejemplo, la clase Square debe proporcionar una implementación de invalidación de Area porque Area se hereda de la clase abstracta Shapes:

```
abstract class Shapes {
    abstract public int Area();
}

class Square : Shapes {
    int side = 0;

    public Square(int n) {
        side = n;
    }

    // Es necesario el método Area para evitar
    // un error en tiempo de compilación.
    public override int Area() {
        return side * side;
    }

    static void Main() {
        Square sq = new Square(12);
        Console.WriteLine("Área del cuadrado = {0}", sq.Area());
    }

}
// Salida: Área del cuadrado = 144
```

Un método override proporciona una nueva implementación de un miembro que se hereda de una clase base. El método invalidado por una declaración override se conoce como método base invalidado. El método base invalidado debe tener la misma firma que el método override.

No se puede invalidar un método estático o no virtual. El método base invalidado debe ser virtual, abstract o override.

## 6.2. Sobrecarga de métodos

La **sobrecarga de métodos**, también llamada polimorfismo ad-hoc, permite que varios métodos de la misma clase tengan el mismo nombre mientras tengan signaturas únicas. Al compilar una invocación de un método sobrecargado, el compilador usa la resolución de sobrecarga para determinar el método concreto que de invocará.

La resolución de sobrecarga busca el método que mejor coincida con los argumentos o informa de un error si no se puede encontrar ninguna mejor coincidencia.

En el ejemplo siguiente se muestra la resolución de sobrecarga en vigor. El comentario para cada invocación del método UsageExample muestra qué método se invoca realmente:

```
class OverloadingExample {

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    public static void UsageExample() {
        F();            // Llama a F()
        F(1);           // Llama a F(int)
        F(1.0);         // Llama a F(double)
        F("abc");       // Llama a F<string>(string)
        F((double)1);   // Llama a F(double)
        F((object)1);   // Llama a F(object)
        F<int>(1);      // Llama a F<int>(int)
        F(1, 1);        // Llama a F(double, double)
    }
}
```

## 7. Conjunto de parámetros de tipo

Una definición de clase puede especificar un conjunto de **parámetros de tipo** poniendo tras el nombre de clase una lista de nombres de parámetro de tipo entre corchetes angulares. Los parámetros

de tipo pueden usarse luego en el cuerpo de las declaraciones de clase para definir a los miembros de la clase. En el ejemplo siguiente, los parámetros de tipo de Pair son TFirst y TSecond:

```
public class Pair<TFirst, TSecond> {
    public TFirst First;
    public TSecond Second;
}
```

Un tipo de clase que se declara para tomar parámetros de tipo se conoce como **tipo de clase genérica**. Los tipos struct, interfaz y delegado también pueden ser genéricos. Cuando se usa la clase genérica, se deben proporcionar argumentos de tipo para cada uno de los parámetros de tipo:

```
Pair<int, string> pair = new Pair<int, string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst es de tipo int
string s = pair.Second; // TSecond es de tipo string
```

Un tipo genérico con argumentos de tipo proporcionado, como Pair<int,string> anteriormente, se conoce como **tipo construido**.

## 8. Clases y métodos parciales

Es posible dividir la definición de una clase, una estructura, una interfaz o un método en dos o más archivos de código fuente. Cada archivo de código fuente contiene una sección de la definición de tipo o método, y todos los elementos se combinan cuando se compila la aplicación. Para ello se consideran las **clases parciales**.

Es recomendable dividir una definición de clase en varias situaciones:

- ✓ Cuando se trabaja con proyectos grandes, el hecho de repartir una clase entre archivos independientes permite que varios programadores trabajen en ella al mismo tiempo.
- ✓ Cuando se trabaja con código fuente generado automáticamente, se puede agregar código a la clase sin tener que volver a crear el archivo de código fuente. Visual Studio usa este enfoque al crear formularios Windows Forms, código de contenedor de servicio Web, etc.

Para dividir una definición de clase, use el modificador de palabra clave **partial**, como se muestra aquí:

```
public partial class Employee {
    public void DoWork() {
    }
}

public partial class Employee {
    public void GoToLunch() {
    }
}
```

La palabra clave partial indica que se pueden definir en el espacio de nombres otros elementos de la clase, la estructura o la interfaz. Todos los elementos deben usar la palabra clave partial. Todos los elementos deben estar disponibles en tiempo de compilación para formar el tipo final. Todos los elementos deben tener la misma accesibilidad, como public, private, etc.

Si algún elemento se declara abstracto, todo el tipo se considera abstracto. Si algún elemento se declara sellado, todo el tipo se considera sellado. Si algún elemento declara un tipo base, todo el tipo hereda esa clase.

Todos los elementos que especifiquen una clase base deben coincidir, pero los elementos que omitan una clase base heredan igualmente el tipo base. Los elementos pueden especificar diferentes interfaces base, y el tipo final implementa todas las interfaces enumeradas por todas las declaraciones parciales. Todas las clases, estructuras o miembros de interfaz declarados en una definición parcial están disponibles para todos los demás elementos. El tipo final es la combinación de todos los elementos en tiempo de compilación.

En el ejemplo siguiente, los campos y el constructor de la clase, Coords, se declaran en una definición de clase parcial y el miembro PrintCoords se declara en otra definición de clase parcial:

```
public partial class Coords {
    private int x;
    private int y;

    public Coords(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords {
    public void PrintCoords() {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }

}

class TestCoords {
    static void Main() {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

## 9. Tipos anónimos

Los **tipos anónimos** son una manera cómoda de encapsular un conjunto de propiedades de sólo lectura en un único objeto sin tener que definir primero un tipo explícitamente. El compilador genera el nombre del tipo y no está disponible en el nivel de código fuente. El compilador deduce el tipo de cada propiedad. Para crear tipos anónimos, se usa el operador new con un inicializador de objeto.

En el ejemplo siguiente se muestra un tipo anónimo que se inicializa con dos propiedades llamadas Amount y Message:

```
var v = new {Amount = 108, Message = "Hello"};

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```