



Diapositiva 1




Día 3:

Módulo 3: Elementos básicos de la programación orientada a objetos

Conceptos de la OOP




Diapositiva 2




Objetivos del día

- Al final de este día, usted podrá:
 - Comprender los conceptos básicos de la Programación orientada a objetos
 - Explicar lo que es una clase y los diferentes tipos de clases que se pueden crear con C#
 - Comprender el concepto de la Encapsulación y cómo realizarla con C#
 - Definir lo que es un Objeto en la OOP
 - Explicar cómo usar Objetos en C#



Diapositiva 3


Contenidos del día

- 1. Conceptos de la OOP
 - 2. Clases en C#
 - 3. Encapsulación en C#
 - 4. Objetos en C#
- 

Diapositiva 4

1. Conceptos de la OOP

Preguntas de la sección

- Preguntas resueltas
 - ¿Qué es una Clase?
 - ¿Qué es un Objeto?
 - ¿Cómo se relacionan las clases y los objetos?
 - ¿Cómo se deben diseñar los objetos?
 - ¿Qué es la Encapsulación?
 - ¿Qué es un Sistema orientado a objetos?
¿Cómo se crean?
 - ¿Cuáles son las ventajas principales de la Programación orientada a objetos?
- 

Diapositiva 5

Clases

- Varias definiciones de “clase”
 - Una representación abstracta de algo real
 - Es lo básico para una clasificación
 - Un perro, un auto, una factura
 - Un borrador para crear objetos
 - Define los atributos y la conducta del objeto

```
class Perro :  
{  
    decimal height;  
    decimal peso;  
    string color;  
    void Walk int steps}  
    void Bark {}  
    void Eat {}  
}
```

Existen varias definiciones de **class**. Una clase (class) es un término que se utiliza no solamente para programación.

Una representación abstracta de algo real

Los elementos básicos para una clasificación es una clase. Construir clases es un acto de una clasificación, y es algo que todos hacen. Por ejemplo, aunque todo perro **real** es diferente al resto, existen comportamientos (todos ladran y caminan) y características (patas, color del pelo entre otros) similares. La gente utiliza el término **perro** para clasificar a ese tipo de animales.

Podemos pensar en una clase llamada perro para crear una representación abstracta del tipo de animales.

Un proyecto para crear objetos

Esta definición es más técnica que la anterior.

Una **class** es esencialmente un proyecto, a partir del cual puede crear objetos.

Una clase define las características de un objeto, incluyendo las propiedades que definen los tipos de datos que ese objeto puede contener y los métodos que describen el comportamiento del objeto. Estas características determinan la manera en que otros objetos pueden acceder y trabajar con los datos que se incluyen en el objeto.

Por ejemplo, si desea construir objetos que representen perros, pueden definir una clase **perro** con ciertos comportamientos, tales como caminar, ladrar y comer, y propiedades específicas tales como altura, peso y color. Una vez que ha definido la clase perro, puede crear objetos con base en esa clase. Es importante observar que todos los objetos perro creados basados en la clase perro compartirán los mismos comportamientos, pero tendrán sus propios valores específicos para el mismo conjunto de propiedades.

El siguiente ejemplo representa la definición de la clase perro. Tome en consideración que ésta no es sintaxis estricta de C#; simplemente es un ejemplo de la definición de clase.

```
class Perro :  
{  
    decimal    height;  
    decimal    height;  
    string color;  
    void        Walk(int steps)  
    void        Bark()  
    void        Eat()  
}
```

Diapositiva 6

Objetos

- Un objeto es la instancia de una clase
 - Una clase es la representación abstracta de algo
 - Un objeto es un ejemplo que se puede realizar de la cosa que representa la clase
 - Clase -> perro
 - Objeto -> mi perro se llama Bart
- Cada objeto muestra:
 - Identidad: cada objeto es distinguible
 - Comportamiento: los objetos realizan tareas
 - Estado: los objetos almacenan información

```
Perro miPerro = new  
perro()
```

Un objeto es una **instancia** de una clase. Una clase es una representación abstracta de algo, mientras que un objeto es un ejemplo utilizable de la cosa que representa la clase.

Siguiendo con el ejemplo del perro, podemos representar a los perros definiendo una clase llamada perro. Después puede crear una nueva instancia de esa clase perro para tener un ejemplo utilizable de un perro: su propio perro o la mascota de su hijo.

Para poder crear un objeto de la clase perro, debe crear una nueva instancia basada en esa clase. Por ejemplo:
`perro miPerro = new perro{}`

Cada objeto es un elemento único de una clase en la que se basa. Si una clase es como un plano, entonces un objeto es lo que se crea a partir del plano. La clase es la definición de un elemento; el objeto **es** el elemento. El plano/proyecto de su casa es como la clase; la casa en la que vive es un objeto.

Todo objeto presenta tres características. Identidad, comportamiento y estado. Esta es una forma útil de pensar y entender los objetos.

Identidad

La identidad es una característica que distingue a un objeto de todos los demás objetos en la misma clase. Por ejemplo, imaginemos dos vecinos que tienen el mismo auto, del mismo modelo y el mismo color. A pesar de las similitudes obvias, está garantizado que los números de registro son únicos y son la reflexión externa de que los autos tienen una identidad. La ley determina que es necesario distinguir a un objeto auto de otro.

Comportamiento


El comportamiento es lo que hace que un objeto sea útil. El objeto existe para poder proporcionar un comportamiento. Este comportamiento de un objeto es lo que es accesible. El comportamiento de un objeto también es lo que determina con mayor fuerza su clasificación. Los objetos de la misma clase comparten el mismo comportamiento. Un auto es un auto debido a que puede conducirlo; una pluma es una pluma porque puede escribir con ella.

Estado

El estado se refiere a los trabajos internos de un objeto que le permiten proporcionar su comportamiento. Un objeto bien diseñado mantiene su estado inaccesible. A usted no le importa cómo un objeto realiza lo que realiza; simplemente le importa que lo realice. Dos objetos pueden contener por coincidencia el mismo estado, sin embargo pueden ser dos objetos diferentes. Por ejemplo, dos perros idénticos contienen exactamente la misma altura, el mismo peso y el mismo color, pero son dos animales distintos.

Diapositiva 7

Diseñar objetos

- El diseño de objetos no es algo simple
 - Usar la Abstracción para enfocarse en los detalles importantes
 - Decidir lo que es importante y lo que no
 - Enfocarse en lo que es importante y depender sólo de ello
 - Ignorar lo que no es importante y no depender de ello
 - Basarse en el principio de Dependencia mínima
 - Utilizar la Encapsulación para aplicar la Abstracción
- 

El diseño de objetos no es algo trivial.

Debido a que las clases son representaciones abstractas de las entidades del mundo real, se puede ver abrumado con la gran cantidad de atributos y comportamientos que tienen estas entidades. Y, si este es el caso, el diseño de sus clases (y los objetos que utilizará más tarde) tendrá un gran conjunto de métodos y propiedades que nunca utilizará.

Para enfocarnos únicamente en los detalles importantes de las entidades que desea representar (los atributos y comportamientos que realmente utilizará), puede utilizar una Abstracción

Abstracción es una táctica de quitarle a una idea o a un objeto todos los acompañamientos innecesarios hasta que los deja en una forma esencial y mínima. Una buena abstracción elimina todos los detalles poco importantes y le permite enfocarse y concentrarse en los detalles importantes.

La abstracción es un principio de software importante. Una clase bien diseñada expone un conjunto mínimo de métodos cuidadosamente considerados que proporcionan el comportamiento esencial de una clase en una forma fácil de usar. Crear buenas abstracciones de software no es fácil. Encontrar buenas

abstracciones generalmente requiere de un entendimiento muy claro del problema y su contexto, gran claridad de pensamiento y una gran experiencia.

Dependencia mínima

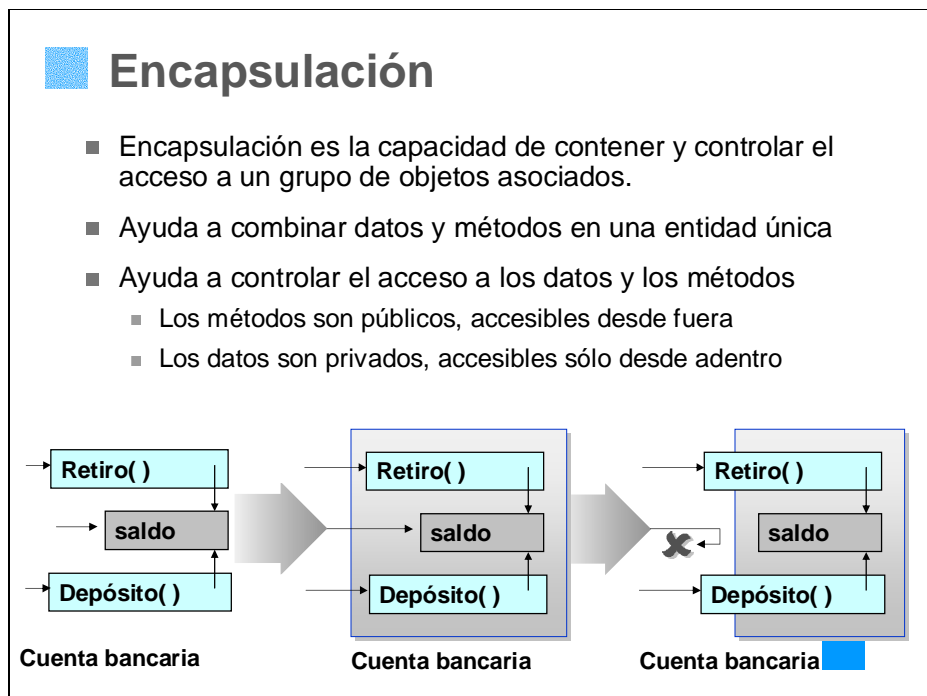
Las mejores abstracciones de software hacen que las cosas complejas sean simples. Logran esto al ocultar implacablemente los aspectos no esenciales de una clase. Estos aspectos no esenciales, una vez que han sido debidamente ocultados, no se pueden ver, usar o utilizar como dependencia.

Es este principio de dependencia mínima lo que hace que la abstracción sea tan importante. El cambio es normal en el desarrollo de software. Lo mejor que puede hacer es minimizar el impacto de un cambio cuando este sucede. Y cuanto menos dependa de algo menos se verá afectado cuando cambia.

Encapsulación

Los lenguajes orientados a objetos proporcionan Encapsulación. La encapsulación se puede utilizar para aplicar el concepto de Abstracción. La encapsulación se detallará en la siguiente sección.

Diapositiva 8



Existen dos aspectos importantes para la encapsulación:

Combinar los datos y los métodos en una entidad única

Controlar la capacidad de acceso de los miembros de la entidad

Combinar los datos y los métodos en una entidad única

La encapsulación significa que un grupo de propiedades, métodos y otros miembros relacionados se traten como una sola entidad u objeto. En la imedadn, los datos (campo balance o saldo) y las funciones (métodos Retiro y Depósito) forman una sola entidad llamada Cuenta Bancaria.

Una vez que se combinan los datos y las funciones en una entidad única, la entidad misma forma un límite cerrado, creando naturalmente una parte interna y una externa.

Controlar la capacidad de acceso de los miembros de la entidad

Los objetos pueden controlar la manera en que los campos cambian y en que se ejecutan los métodos. Por ejemplo, un objeto puede validar los valores antes de permitir cambios a la propiedad. En la gráfica del centro, "Retiro", "Depósito" y "saldo" han sido agrupados dentro de la entidad CuentaBancaria. Sin embargo, existe algo equivocado en este modelo de cuenta bancaria: los datos del "balance". Si este fuera el caso del mundo real, podría incrementar su saldo sin hacer ningún depósito.

Puede resolver este problema al utilizar la encapsulación. Una vez que los datos y las funciones se combinan en una sola entidad, la entidad misma forma un cerco cerrado. Puede usar este límite para controlar selectivamente la capacidad de acceso de las entidades: algunas se podrán acceder solamente desde dentro; otras tanto desde dentro como desde fuera. Los miembros que siempre son accesibles son *públicos*, y los que sólo se pueden acceder desde dentro son *privados*.

Para hacer el modelo de una cuenta bancaria más próximo a una cuenta bancaria real, puede hacer que los métodos "Retiro" y "Depósito" sean públicos y que "saldo" sea privado. Ahora, la única manera de incrementar un saldo de la cuenta desde el exterior es Depósitoar dinero en la cuenta. Observe que "Depósito" puede acceder a "saldo" debido a "Depósito" está dentro.

La técnica de declarar detalles internos de una clase como Privados para evitar que sean utilizados desde el exterior se llama ocultar datos.

Ocultar datos también facilita cambiar la implementación en un momento posterior. Por ejemplo, versiones posteriores de la clase CuentaBancaria podrían cambiar el tipo de datos del campo balance sin peligro de dañar la aplicación que depende que este campo tenga un tipo de dato específico.

¿La encapsulación es importante? Sí.

Existen dos razones para encapsular:

Controlar el uso

Minimizar el impacto del cambio.

La encapsulación permite control

La primera razón para encapsular es controlar el uso. Cuando conduce un auto, únicamente piensa en el acto de conducir, no en los elementos internos de un auto. Cuando retira dinero de una cuenta no piensa cómo se representa la cuenta. Puede utilizar la encapsulación y los métodos de comportamiento para diseñar objetos de software de manera que únicamente se utilizan de la forma que usted pretende.

La encapsulación permite cambios


La segunda razón para encapsular se deriva de la primera. Si los detalles de implementación de un objeto son privados, se pueden modificar y los cambios no afectarán directamente a los usuarios del objeto (quienes únicamente puede acceder a los métodos únicos). En la práctica, esto puede ser sumamente útil. Los nombres de los métodos generalmente se estabilizan muchos antes de la implementación de los métodos.

La capacidad de realizar cambios internos se vincula muy de cerca con la Abstracción. Dados dos diseños de una clase, utilice uno para menos métodos públicos.

Si existe una necesidad de decidir si un método debe ser público o privado, hágalo privado. Un método privado puede ser modificado fácilmente y quizás más tarde promoverlo a un método público. Sin embargo, un método público no puede pasar a método privado sin romper el código del cliente.

Diapositiva 9

Ventajas de la programación orientada a objetos

- La programación orientada a objetos ofrece a los desarrolladores una mejor forma de escribir software
 - Las técnicas de la programación orientada a objetos se compara con otras técnicas:
 - Centrada en el proceso vs. centrada en el objeto
 - Revela datos vs. oculta datos
 - Unidad única vs. unidad modular
 - Uso único vs. reutilizable
 - Algoritmo ordenado vs. algoritmo no ordenado
- 

Hacia mediados de los 80, los beneficios de la programación orientada a objetos empezaron a obtener reconocimiento, y el diseño de objetos pareció ser un enfoque sensato para la gente que deseaba utilizar el lenguaje de programación orientada a objetos.

La programación orientada a objetos ofrece a los desarrolladores una mejor manera de escribir software.

Un enfoque orientado a objetos para programar ofrece muchos beneficios sobre un enfoque estructurado.

Centrado en procesos vs. Centrado en objetos

Un enfoque estructurado de programación está centrado en procesos, lo cual significa que toma un problema y se enfoca en una jerarquía de procesos que se deben llevar a cabo de manera secuencial para llegar a una solución.

El análisis orientado a objetos y su diseño se enfoca en los objetos. Los objetos tienen ciertos comportamientos y atributos que determinarán la manera en que interactúan y funcionan. No se intenta proporcionar un orden para las acciones

al momento de diseño debido a que los objetos funcionan basados en la manera en que funcionan otros objetos.

La programación orientada a objetos ayuda a los desarrolladores a crear objetos que reflejen escenarios del mundo real. La mayor parte de las personas encuentran que el enfoque orientado a objetos es un modelo de diseño mucho más natural que otras metodologías. Esto se debe a que se mezcla muy bien con la forma en que la gente interpreta el mundo de manera natural. El entendimiento humano depende en gran medida de la identificación y generalización (objetos y clases), encontrando relaciones entre los grupos, interactuando a través de una interfaz normal de la entidad (comportamientos).

Revela datos vs. oculta datos

El enfoque estructurado empaca los datos y procedimientos, mismos que se revelan o están accesibles al resto del programa. Hay muy poco esfuerzo para realmente ocultar la información de otros procesos. El enfoque estructurado deja esta decisión al implementador.

Las implementaciones orientadas a objetos ocultan datos, lo cual significa que muestran únicamente los comportamientos a los usuarios y ocultan el código subyacente de un objeto. Los comportamientos que el programador expone son únicamente aquellos elementos que el usuario de un objeto puede afectar.

Unidad única vs. Unidad modular

El enfoque estructurado se basa en una unidad única de código, en donde los procesos invocan a otros procesos y dependen entre sí.

El enfoque orientado a objetos permite que los objetos estén auto contenidos. Los objetos existen por sí mismos, con una funcionalidad para invocar comportamientos de otros objetos. Al utilizar un enfoque orientado a objetos, los desarrolladores pueden crear aplicaciones que reflejen objetos del mundo real tales como rectángulos, elipses y triángulos, además de dinero, números de partes, elementos en un inventario.

Uso de una sola vez vs. reutilizable

Un proceso estructurado no puede ser reutilizado dependiendo de la implementación.

En un enfoque orientado a objetos, los objetos por definición son modulares en su construcción. Esto quiere decir que son entidades completas y, por lo tanto, tienden a ser altamente reutilizables.

Considere el ejemplo de adquirir un auto nuevo. El fabricante arma un auto con un modelo base. Si prefiere características adicionales tales como aire acondicionado, ventanas eléctricas y otro equipo adicional, estos elementos se pueden agregar al auto. Al agregar características, amplía las características del modelo base en lugar de construir un auto totalmente nuevo.

Algoritmo ordenado vs. Algoritmo no ordenado

Los enfoques estructurados tienden a resultar en una implementación lineal o de arriba hacia abajo basada en algoritmos.

Las aplicaciones orientadas a objetos se construyen sobre el paradigma de los mensajes o de los eventos en donde los objetos envían mensajes a otros objetos, tales como el sistema operativo Microsoft® Windows®.

Resumen de los beneficios

En resumen, la programación orientada a objetos beneficia a los desarrolladores debido a que:

Los programas son fáciles de diseñar debido a que los objetos reflejan elementos del mundo real

Las aplicaciones son más fáciles para los usuarios debido a que están ocultos los datos innecesarios

Los objetos son unidades auto contenidas

La productividad se incrementa debido a que puede reutilizar el código

Los sistemas son fáciles de mantener y adaptar a las cambiantes necesidades de negocios

Es más fácil crear nuevos tipos de objetos a partir de los existentes

Simplifica los datos complejos

Reduce la complejidad de la transacción

Confiabilidad

Robustez

Capacidad de ampliación

Diapositiva 10

■ 2. Clases en C#

Preguntas de la sección

- Preguntas resueltas
 - ¿Cómo crear una clase simple en C#?
 - ¿Cómo crear objetos con C#?
 - ¿Cuál es la diferencia entre miembros estáticos y de instancia?

Diapositiva 11

■ Cómo definir una clase en C#

- Utilizar la palabra clave **class** antes del nombre de la clase
- Insertar los miembros de la clase entre corchetes
 - Colocar los datos y métodos juntos dentro de una clase

```
class Cliente
{
    public void ActualizarCiudad (string Ciudad)
    { ... }

    public string    nombre;
    public decimal   limiteCredito;
    public uint      ClientelD;
}
```

Para definir una clase, se coloca la palabra clave **class** antes del nombre de su clase, y después se insertan los miembros de la clase (datos y métodos) entre llaves:

class *identificador* {cuerpo}

Si incluye métodos, entonces el código de cada método se deberá incluir entre las llaves.

El siguiente ejemplo define una nueva clase **Cliente**, con tres partes de información relevante asociada: el nombre del cliente, el límite de crédito del cliente y el identificador del cliente, así como un método que permite actualizar la ciudad del cliente (ActualizarCiudad). A pesar de que la clase **Cliente** se define en el ejemplo, no existen a un objeto **Cliente**. Deberán ser creados.

```
class Cliente
{
    public void ActualizarCiudad (string City)
        { ... }
    public string nombre;
    public decimal limiteCredito;
    public uint    ClienteID;
}
```

Una clase es un tipo definido por el usuario en contraposición a un tipo proporcionado por el sistema. Al definir una clase, en realidad crea un nuevo tipo en su aplicación.

Diapositiva 12

Cómo crear un objeto en C#

- Cómo crear la instancia de una clase como un objeto
 - Utilizar el operador **new** para crear un objeto

```
Cliente nextCliente = new Cliente();
```

- Cómo acceder a los miembros de la clase
 - Utilizar el nombre de la clase en instancia, un **punto** y el nombre del miembro de la clase que necesita

```
aCliente.nombre = "John Doe";
```



Para utilizar una clase que haya definido, deberá primero crear la instancia de un objeto de ese tipo utilizando la palabra clave **new**.

Sintaxis

<object_variable> = **new** <class>

```
nextCliente = new Cliente();
```

También puede crear la instancia de un objeto cuando se declara la variable.
Por ejemplo:

```
Cliente nextCliente = new Cliente();
```

Acceder a las variables de clase

Una vez que ha creado la instancia de un objeto para acceder y utilizar los datos que contiene ese objeto, deberá escribir el nombre de la clase creada en la instancia, seguida por un punto y el nombre del miembro de la clase que desea acceder.

Por ejemplo, puede acceder al miembro **nombre** de la clase **Cliente** y asignarle un valor en su objeto **aCliente**, el nombre de la clase creada en la instancia, como sigue:


```
atCliente.nombre = "John Doe";
```

El siguiente código define una nueva clase llamada **Perro** con un miembro de clase llamado **peso**, y crea una instancia de la clase **Perro** , un objeto llamado **peqPerro** . El valor se asigna al miembro **peso** en la clase **peqPerro** .

```
public class Perro
{
    public int peso;
}

...

Perro peqPerro = new Perro ();
peqPerro .peso = 100;
```

Las clases son tipos de referencia

Cada objeto **Perro** creado en un objeto separado, como aparece en el siguiente código:

```
PerromayorPerro = new Perro ();
Perro menorPerro = new Perro ();
largerPerro .peso = 100;
```

El código anterior no cambia al miembro **peso** del objeto **menorPerro** . Ese valor es cero, el valor predeterminado para un número entero.

El siguiente código cambia el miembro **peso** del objeto **menorPerro** :

```
Perro mayorPerro = new Perro ();
largerPerro.peso = 125;
Perro menorPerro = new Perro ();
menorPerro.peso = 75;
Perro perroRecienPesado = menorPerro ;
perroRecienPesado .peso = 85;
// el peso de menorPerro ahora es 85.
```

En el código anterior, el valor de **menorPerro.peso** es **85**.

Debido a que **Perro** es un tipo de referencia, la asignación a **perroRecienPesado** causa que **menorPerro** y **perroRecienPesado** haga referencia al mismo objeto.

Destrucción de un objeto

Al crear un objeto, en realidad está asignando un espacio en la memoria para ese objeto. Microsoft® .NET Framework proporciona una función de administración automática de la memoria llamada **recolección de residuos** (explicaremos la recolección de residuos en un modulo diferente).

Diapositiva 13

Miembros estáticos versus instancias

- Los miembros estáticos los comparten todas las instancias de una clase
- Los miembros de instancia pertenecen a instancias específicas de una clase, un objeto
 - Cada instancia de una clase contiene una copia separada de todos los miembros de instancia de la clase
- De manera predeterminada, todos los miembros son miembros de instancia
 - Para volverlos estáticos, utilice el modificador “estático” antes del nombre del miembro

```
class ATimer
{
    static int tickCount;
}
```

En algunas ocasiones no vale la pena almacenar información dentro de cada objeto. Por ejemplo, si todas las cuentas de banco siempre comparten la misma tasa de interés, entonces almacenar la tasa dentro de cada objeto de cuenta no sería una buena idea por las siguientes razones:

Es una implementación deficiente del problema según se describe: Todas las cuentas bancarias comparten la misma tasa de interés.

Incrementa innecesariamente el tamaño de cada objeto, utilizando recursos de memoria adicionales, cuando el programa se está ejecutando y el espacio de disco adicional cuando se guarda en el disco.

Hace difícil modificar la tasa de interés. Tendría que cambiar la tasa de interés en cada objeto de cuenta. Si necesita un cambio en la tasa de interés en cada objeto individual, un cambio en la tasa de interés haría que todas las cuentas quedaran inaccesibles, mientras que se lleva a cabo el cambio.

Incrementa el tamaño de la clase. Los datos privados de la tasa de interés requieren métodos públicos. La clase cuenta empieza a perder su cohesión. Ya no realiza una cosa y solo una, pero bien hecha.

Para resolver este problema, no comparta la información que es común entre objetos a nivel objeto. En lugar de describir la tasa de interés, muchas veces a nivel de objeto, describa la tasa de interés una vez en el nivel de clase. Cuando define la tasa de interés a nivel de clase, se vuelve realidad en un dato global.

Sin embargo, los datos globales, por definición, no se almacenan dentro de una clase y, por lo tanto, no se pueden encapsular. Debido a que estos lenguajes de programación orientados a objetos (que son muchos y que incluyen a C#) no permiten datos globales. En cambio, permiten que los datos se describan como estáticos.

Declarar datos estáticos

Los datos estáticos se declaran físicamente dentro de una clase y se benefician de las posibilidades que ofrece la encapsulación de la clase, pero están asociados lógicamente con la clase misma y no con cada objeto. En otras palabras, los datos estáticos se declaran dentro de una clase por conveniencia sintáctica y existen incluso si el programa nunca genera ningún objeto de esa clase.

Los miembros estáticos son miembros (campos y métodos) que se comparten en todas las instancias de la clase.

Los miembros de instancia son miembros (campos y métodos) que pertenecen a una instancia específica de una clase, un objeto.

Un campo declarado con el modificador **static** es un campo **estático**. Un campo **static** identifica exactamente una sola ubicación de almacenamiento. Sin importar cuantas instancias de un tipo se generen, solamente existe una copia de un campo **static**. Un campo **static** surge cuando el programa empieza su ejecución y deja de existir cuando se termina el programa. Se inicia un campo **static** con el valor predeterminado de su tipo. Debido a que los miembros **static** pertenecen a una clase, en lugar de una instancia, se acceden a través de la clase, y no a través de una instancia de la clase.

Un campo declarado sin el modificador **static** es un campo **de instancia**. Cada instancia de una clase contiene una copia separada de todos los campos **de instancia** de esa clase. Todos los campos de instancia se inicializan con el valor predeterminado de ese tipo. Un campo de instancia del tipo de referencia surge cuando se crea una nueva instancia de ese tipo y deja de existir cuando ya no existen referencias a esa instancia y se ejecuta el método **Finalize** (veremos el método Finalize más adelante en este modulo).

Los campos estáticos son útiles cuando tiene información que es parte de una clase, pero que no es específica una instancia de una clase. Los campos normales existen independientemente para cada instancia de una clase. Cambiar el valor de un campo asociado con cualquiera de las instancias no afecta el valor de los campos o propiedades de otras instancias de la clase.

Los métodos también pueden ser estáticos. Cuando un modificador de acceso estático se aplica a un método, el método es accesible únicamente a través de la clase y no en una instancia del objeto. Los únicos miembros de una clase que puede acceder a un método estático son los datos estáticos y otros métodos estáticos. Debido a que los métodos son parte de una clase, puede invocarlos sin crear una instancia del objeto. En C#, no puede acceder a un método estático dentro de una instancia.

Ejemplos de miembros estáticos

El siguiente ejemplo crea un campo de instancia, un campo estático y un método estático para demostrar la manera en que operan los miembros estáticos en el código:

```
public class AClass
{
    public string InstanceValue;
    public static string StaticValue;

    public static void StaticMethod()
    {
        Console.WriteLine("Este es un método estático.");
    }
}
```

```

public class Class1
{
    static void Main( )
    {
        AClass Static1 = new AClass();
        AClass Static2 = new AClass();
        Static1.InstanceValue = "Valor de instancia 1";      //
válido
        Static2.InstanceValue = " Valor de instancia 1 ";
// válido

        Static1.StaticValue = "Valor estático 1";          // no
válido
        Static2.StaticValue = " Valor estático 2";          // no
válido

        AClass.StaticValue = "Este es un valor estático";
// válido
        AClass.StaticMethod();
// válido
    }
}

```

Al ejecutar el procedimiento Main, se crean dos instancias de la clase, y se modifica el campo de instancia InstanceValue para ambas instancias. Cuando se modifica el campo de instancia en la segunda instancia de la clase, no se modifica el valor asignado al campo de instancia en la primera instancia de la clase (debido a que pertenece a diferentes instancias).


No puede acceder al campo estático StaticValue desde cualquiera de las instancias de la clase.

Este código no se compilará.

Diapositiva 14


3. Encapsulación en C#

Preguntas de la sección

- Preguntas resueltas
 - ¿Cómo aplica la encapsulación utilizando C#?
 - ¿Cómo restringe el acceso a los métodos?
 - ¿Qué son las propiedades? ¿Cómo las utiliza?
 - ¿Cuántos niveles de acceso están disponibles?
- 

Diapositiva 15

Encapsulación en C#

- Existen dos aspectos importantes de la encapsulación:
 - Combinar datos y funciones en una entidad única
 - Utilizar clases para definir las entidades y los objetos para crearlos
 - Controlar la accesibilidad de los miembros de la entidad
 - Utilizar modificadores de accesibilidad para definir el alcance de los miembros
 - Los métodos requeridos se deben marcar como públicos y se debe acceder a ellos desde afuera
 - El campo se debe marcar como privado y se debe acceder a él desde dentro
 - Utilizar propiedades para acceder a campos
- 

Existen dos aspectos importantes para la encapsulación:

Combinar los datos y las funciones en una entidad única

Controlar la capacidad de acceso de los miembros de la entidad

Combinar los datos y las funciones en una entidad única

Como vimos antes, las clases proporcionan la abstracción necesaria para crear entidades, y los objetos se pueden utilizar para crear instancias únicas.

Controlar la capacidad de acceso de los miembros de la entidad

Una vez que se combinan los datos y las funciones en una entidad única, la entidad misma forma un límite cerrado, creando naturalmente una parte interna y una externa. Puede usar este límite para controlar selectivamente la capacidad de acceso de las entidades: algunas se podrán acceder solamente desde dentro; otras tanto desde dentro como desde fuera. Los miembros que siempre son accesibles son **públicos**, y los que sólo se pueden acceder desde dentro son **privados**. Esto se conoce como el alcance de los miembros.

C#, al igual que muchos otros lenguajes de programación orientados a objetos, ofrece libertad completa al elegir si los miembros deben ser accesibles o no. Puede, si lo desea, crear datos públicos. Sin embargo, se recomienda que los datos siempre estén marcados como privados y únicamente los métodos requeridos estén marcados como públicos. Si necesita proporcionar acceso a los campos, puede utilizar las propiedades, como lo veremos más tarde.

Diapositiva 16

Declarar métodos

- Declarar un método como público
 - Sólo métodos a los que se debe de acceder desde afuera
- Declarar un método como privado
 - Todos los demás métodos

```
clase CuentaBancaria
{
    public bool Retiro() { ... }
    public bool Depósito() { ... }

    private int CalcularInterés() { ... }
    private bool EstablecerSucursal() { ... }
    ... }
    // add fields...
}
```



Al diseñar una clase debe considerar la manera en que los usuarios de esa clase accederán a sus miembros. Cuantos más miembros públicos ofrezca, más acceso ofrecerá.

Esto es válido para los métodos y para los campos.

Al declarar un método como **public**, el método estará accesible desde el exterior. Únicamente los métodos que desee tener accesibles desde el exterior deben marcarse como públicos.

Cuando declara un método como **private**, el método no estará accesible desde el exterior. Únicamente se puede acceder desde el interior de la clase, tanto para los métodos públicos como privados. La mayoría de los métodos en una clase se deben marcar como privados.

Una gran cantidad de diseño está relacionada a la decisión de colocar una función en el interior o el exterior. Cuantas más funciones coloque en el interior (y mantenga su capacidad de uso), mejor.

Ejemplo

En el siguiente ejemplo los métodos Retiro y Depósito serán accesibles desde el exterior. Cada objeto que pueda acceder a un objeto del tipo CuentaBancaria, podrá invocar Retiro y Depósito. Por otro lado, los métodos CalcularInterés y EstablecerSucursal (establecer la sucursal del banco) no se podrán acceder desde el exterior. El método CalcularInterés sólo se podrá acceder desde los métodos Retiro, Depósito y AsignarIdDeCuenta y AsignarIdDeCuenta sólo se podrá acceder desde los métodos Retiro, Depósito y CalcularInterés.

```
class CuentaBancaria
{
    public bool Retiro() ... }
    public bool Depósito() ... }

    private int CalcularInterés() ... }
    private bool AsignarIdDeCuenta() ... }
    // add fields...
}
```


Diapositiva 17

Declarar propiedades

- ¿Qué son las propiedades?
 - Miembros de clase que proporcionan acceso a los elementos de un objeto o clase
- ¿Cómo declarar una propiedad?
 - Usar métodos de acceso para definir el código con la finalidad de obtener y establecer el valor de propiedad

```
public int myIntegerProperty {  
    get {  
        // Código para obtener el valor de la propiedad  
    }  
    set {  
        // Código para asignar el valor a la propiedad  
    }  
}
```

La capacidad de acceso también se puede definir por campos (no sólo por métodos). A pesar de que puede controlar el acceso a los campos de clase utilizando los modificadores de acceso, una forma más poderosa de administrar el acceso es a través del uso de **propiedades**. Al utilizar las propiedades, puede administrar el acceso que otros objetos tienen a los datos en su clase.

propiedades son miembros de clase que proporcionan acceso a los elementos de un objeto o clase.

Las propiedades son una extensión de los campos y se acceden utilizando la misma sintaxis. A diferencia de los campos, las propiedades no designan ubicaciones de almacenamiento. En lugar de eso, las propiedades tienen elementos de acceso que leen, escriben o calculan sus valores.

Sintaxis

La sintaxis para definir una propiedad consiste en un modificador de acceso tal como **public** o **protected**, seguido por el tipo, el nombre de la propiedad, las palabras clave **get** y **set**, y el código de la propiedad para cada uno dentro las llaves, como aparece en el siguiente código:

```

public int myIntegerProperty {
    get {
        // Código para obtener el valor de la propiedad
    }
    set {
        // Código para asignar el valor a la propiedad
    }
}

```

Las instrucciones **get** y **set** se llaman **accessors**.

El elemento de acceso **get** puede devolver un tipo que es el mismo que el tipo de propiedad o uno que puede estar implícitamente convertido en el tipo de propiedad. El elemento de acceso **set** es equivalente a un método que tiene un parámetro implícito llamado **value**.

Al utilizar las propiedades se aplica la encapsulación. Ningún campo se podrá acceder desde el exterior, y tendrá la capacidad de validar el acceso a los campos al agregar su propio código personalizado dentro de los elementos de acceso set y get.

Ejemplo

Escribe una aplicación para dar seguimiento a la cantidad de alimento que consumen los perros de manera que pueda utilizar este valor para predecir sus compras futuras de alimentos. Decide representar este valor de consumo como DailyFood (alimento diario), como aparece en el siguiente código:

```

class Perro :
    // Not a good idea!
    public decimal DailyFood;
}

class Track {
    static void Main(string[] args) {
        Perro miPerro = new Perro ();
        miPerro .DailyFood = 15;
    }
}

```

Este código permite que acceda directamente al valor `DailyFood` de `miPerro` y lo altere. Esta es una falla de diseño debido a que el programador no puede asegurar que el cambio sea permitido o que el valor sea correcto. **¡La clase `Perro` no proporciona encapsulación!**

Uso de propiedades

Usar propiedades es la mejor manera de declarar `DailyFood`, como se muestra en el siguiente código:

```
class Elephant    {
    private decimal dailyConsumptionRate;
    public decimal DailyFood {
        get {
            return dailyConsumptionRate;
        }
        set {
            if ( value > dailyConsumptionRate + 20 )
            { // add more money to food budget    }
            else{
                dailyConsumptionRate = value;
            }
        }
    }
}

class Track {
    static void Main(string[] args) {
        Perro miPerro = new Perro ();
        miPerro .DailyFood = 15;
    }
}
```

En este ejemplo, los usuarios del objeto `Perro` puede acceder al método `DailyFood` en la misma manera en que accederían a la variable del miembro público en la clase. El implementador de la clase puede separar la interfaz que ofrece **`DailyFood`** de la variable del miembro que se utiliza internamente en la clase, **`dailyConsumptionRate`** (cantidad diaria de consumo), para predecir las compras de alimentos para animales. En implementaciones futuras, el

programador puede cambiar **dailyConsumptionRate** por otro tipo, pero los usuarios de la clase **Perro** no tendrán que modificar su código.

Observe que el elemento de acceso **set** utiliza la palabra clave **value** para recuperar el valor nuevo.

Diapositiva 18

Alcance

- Alcance: la región del código desde la que se puede referenciar un elemento del programa
- Los modificadores de acceso permiten definir el alcance de los miembros de la clase
- Modificadores de acceso disponibles:

Capacidad de acceso	Significado
Public	El acceso no está restringido
protected	El acceso está limitado a la clase de contenido o a los tipos derivados de la clase de contenido
internal	El acceso está limitado al proyecto actual
internal protected	El acceso está limitado al proyecto actual o a los tipos derivados de la clase de contenido
private	El acceso está limitado al tipo de contenido

Definición del alcance

El alcance se refiere a la región del código desde la cual se puede referenciar un elemento del programa. En el siguiente ejemplo, el miembro **peso** de la clase **Perro** se puede acceder únicamente desde dentro de la clase **Perro** . Por lo tanto, el **scope** del miembro **peso** es la clase **Perro** .

Al utilizar los modificadores de acceso, puede definir el alcance de los miembros de la clase en sus aplicaciones. Es importante entender la manera en que los modificadores de acceso funcionan debido a que afectan su capacidad para utilizar una clase y sus miembros.

Cuando se permite el acceso a un miembro, se dice que es accesible. De otra manera, es inaccesible. Utilice los modificadores de acceso, **public**, **protected**, **internal** o **private** para especificar una de las capacidades de acceso que se muestran en la diapositiva para los diferentes miembros.

Se permite un solo modificador de acceso para un miembro o tipo, a excepción de la combinación **protected internal**.

Los modificadores de acceso no están permitidos en los espacios de nombre. Los espacios de nombre no tienen restricciones de acceso (veremos espacios de nombre posteriormente en este capítulo).

Reglas

Se aplican las siguientes reglas:

Los espacios de nombre siempre son públicos (implícitamente).

Las clases siempre son públicas (implícitamente).

Los miembros de clase son privadas por predeterminación.

Sólo un modificador se puede declarar en un miembro de clase. Aunque *protected internal* son dos palabras, es sólo un modificador de acceso.

El alcance de un miembro nunca es mayor al del tipo que lo contiene.

Recomendaciones.


La capacidad de acceso de los miembros de su clase determina el conjunto de comportamientos que ve el usuario de su clase. Si define un miembro de clase como privado, el usuario de esa clase no puede ver o utilizar ese miembro. Debe hacer públicos únicamente aquellos objetos que los usuarios de su clase necesiten ver. Al limitar el conjunto de acciones que su clase hace pública reduce la complejidad de su clase desde el punto de vista del usuario, y es más fácil para usted documentar y mantener su clase.

```
class ClassMain {
    public class Perro {
        public int edad;
        private int    peso;
    }
    static void Main(string[] args) {
        Perro miPerro = new Perro ();
        miPerro.edad = 3;
        // la siguiente línea da error en la compilación
        miPerro.peso = 75;
    }
}
```

Diapositiva 19


4. Objetos en C#

Preguntas de la sección

- Preguntas resueltas
 - ¿Cómo se crean los objetos?
 - ¿Cómo se maneja la memoria?
 - ¿Qué es un constructor?
 - ¿Qué tipo de constructores tenemos?
 - ¿Qué es un destructor?
 - ¿Cómo se destruyen los objetos?
- 

Diapositiva 20

Crear y destruir objetos

- Crear objetos
 - Utilizar la palabra clave **new** para adquirir y asignar memoria
 - Llamar al constructor para convertir la memoria binaria adquirida por **new** dentro de un objeto
 - Destruir objetos
 - Des inicializar el objeto, convertir el objeto de nuevo a memoria binaria. En C#, esto se realiza mediante el destructor.
 - Desasignar la memoria binaria, esto es, regresarla a la pila de memoria.
- 

Crear objetos

Crear un objeto C# para un tipo de referencia es un proceso de dos pasos como sigue:

Utilizar la palabra clave **new** para adquirir y asignar memoria

Llamar al constructor para convertir la memoria cruda adquirida por **new** dentro de un objeto

A pesar de que hay dos pasos en este proceso, deberá realizar ambos en una expresión. Por ejemplo, si **Perro** es el nombre de la clase, utilice la siguiente sintaxis para asignar memoria e inicializar el objeto **miPerro** :

```
Perro miPerro = new Perro ();
```

Paso 1: Asignar memoria

El primer paso en crear un objeto es asignar memoria para el objeto. Todos los objetos se crean utilizando el operador **new**. No hay excepción a esta regla. Puede hacerlo de manera explícita en su código, o el compilador lo hará por usted.

Paso 2: Inicializar el objeto utilizando un constructor

El segundo paso al crear un objeto es llamar a un constructor. Un constructor convierte la memoria asignada por **new** en un objeto. Existen dos tipos de constructores: constructores de instancias y constructores estáticos. Los constructores de instancias son constructores que inicializan los objetos. Los constructores estáticos son constructores que inicializan las clases.

Destruir los objetos

Destruir un objeto C# es también un proceso de dos pasos:

Desinicializar el objeto. Esto convierte nuevamente al objeto en una memoria cruda. En C# esto se realiza mediante el destructor. Esto es lo inverso a la inicialización realizada por el constructor. Puede controlar lo que sucede en este paso al escribir su propio destructor o método de finalización.

La memoria cruda también se des asigna; esto es, se regresa nuevamente a la pila de memoria. Esto es lo inverso de la asignación realizada por **new**. No puede cambiar el comportamiento en este paso en forma alguna.

Paso 1: Desinicializar el objeto

El tiempo de vida de un objeto no está vinculado con el alcance en el cual se crea. Los objetos se inicializan en la pila de memoria asignada a través del operador **new**. Por ejemplo, en el siguiente código, la variable de referencia **eg** se declara dentro de la instrucción **for**. Esto significa que **eg** sale del alcance final de la instrucción **for** y es una variable local. No obstante, **eg** se inicializa

con un objeto **new Example()**, y este objeto no sale fuera del alcance con *eg*. Recuerde, una variable de referencia y el objeto a la cual hace referencia son cosas diferentes.

```
class Ejemplo
```

```
{
```

```
    void Metodo(int limite)
```

```
    {
```

```
        for (int i = 0; i < limite; i++) {
```

```
            Ejemplo eg = new Ejemplo( );
```

```
            ...
```

```
        }
```

```
        // eg está fuera del alcance
```

```
        // ¿Existe eg? No.
```

```
        // ¿Puede existir el objeto aún? Si, está en memoria, pero no puede ser alcanzado (la variable eg está fuera de alcance en este punto).
```

```
    }
```

```
}
```

Esto significa que los objetos, por lo general, tienen las siguientes características:

Destrucción no determinista

Un objeto se crea cuando lo crea, pero a diferencia de un valor, no se destruye al final del alcance en el cual se ha creado. La creación de un objeto es determinista, pero la destrucción de un objeto no lo es. No puede controlar exactamente cuándo un objeto será destruido.

Tiempos de vida más largos

Debido a que la vida de un objeto no está vinculada con el método que genera, un objeto puede existir mucho más allá de una sola indicación del método .

Pero, de nuevo, la variable de referencias utilizada para referenciar el objeto no lo hará.

Cuando se destruye finalmente un objeto, se convierte de nuevo a memoria bruta. En C#, no hay manera de destruir explícitamente los objetos.

Paso 2: Des asignar memoria

En este segundo paso, la memoria bruta se devuelve a la pila para ser reciclada. **El recolector de residuos** automatiza completamente el segundo paso de este proceso para usted. La recolección de residuos es un proceso

automático no determinista que asegura que los objetos que no se alcancen sean destruidos y que la memoria bruta del objeto este libre al devolverla a la pila para ser reciclada.

Diapositiva 21

Utilizar constructores

- Constructor de instancia:
 - Inicializa la memoria y la convierte en un objeto que está listo para usarse
- Si no se especifica un constructor:
 - El compilador genera un constructor predeterminado
 - Accesibilidad pública
 - Mismo nombre que el de la clase
 - Sin tipo de retorno, ni siquiera **null**
 - No espera argumentos
 - Inicializa todos los campos en **cero, falso o nulo**
- Si se especifica un instructor
 - Se invoca cuando se inicializa el objeto
 - Se puede sobrecargar

Cómo colaboran los constructores de new y de instancias

Es importante observar cuan de cerca colaboran los constructores **new** y de instancias para crear objetos. El único objeto de **new** es adquirir memoria bruta no inicializada. El único objeto de un constructor de instancias es inicializar la memoria y convertirla en un objeto preparado para usarse. Especialmente, **new** no está involucrado con la inicialización en forma alguna y los constructores de instancias no están involucrados en adquirir memoria en forma alguna.

Al crear un objeto, el compilador C# proporciona un constructor predeterminado si no desea escribirlo usted mismo. Considere el siguiente ejemplo:

```
class Date
{
    private int yy, mm, dd;
}
class Test
{
    static void Main( )
```

```

{
    Date todayVar = new Date( );
    ...
}

```

La instrucción dentro de **Test.Main** crea un objeto **Date** llamado **todayVar** utilizando **new** (que asigna memoria desde la pila) e invoca un método inicial que tiene el mismo nombre que la clase (constructor de instancias). Sin embargo, la clase **Date** no declara un constructor de instancias. De manera predeterminada, el compilador genera automáticamente un constructor de instancias predeterminado.

Características del constructor predeterminado

En su concepto, el constructor de instancias que genera para el compilador de la clase **Date** se ve como el siguiente ejemplo:

```

class Date
{
    public Date( )
    {
        yy = 0;
        mm = 0;
        dd = 0;
    }
    private int yy, mm, dd;
}

```

El constructor tiene las siguientes características:

Mismo nombre que el nombre de clase

Por definición, un constructor de instancias es un método que tiene el mismo nombre que su clase. Es la definición natural e intuitiva y se correlaciona con la sintaxis que ya ha visto. A continuación hay un ejemplo:

```
Date todayVar = new Date( );
```

Sin tipo de retorno

Esta es la segunda característica que define a un constructor. Un constructor nunca tiene un tipo de retorno ni siquiera **void**.

No se requieren argumentos

Es posible declarar constructores que tomen argumentos. Sin embargo, el constructor predeterminado generado por el compilador no espera argumentos.

Todos los campos se inicializan con cero

Esto es importante. El constructor predeterminado generado por el compilador inicializa de manera implícita todos los campos no estáticos como sigue:

Campos numéricos (tales como **int**, **double** y **decimal**) se inicializan a cero.

Los campos del tipo **bool** se inicializan en **false**.

Tipos de referencia (cubiertos en un módulo anterior) se inicializan en **null**.

Los campos del tipo **struct** se inicializan para contener valores de cero en todos sus elementos.

Accesibilidad pública

Esto permite que se generen nuevas instancias del objeto.

Escribir su propios constructor predeterminado

Existen diferentes casos en los cuales puede no ser adecuado el constructor predeterminado generado por el compilador:

El acceso público en ocasiones no es adecuado.

La inicialización a cero a veces no es adecuada.

El código invisible es difícil de mantener. No puede ver el código del constructor predeterminado.

Si el constructor predeterminado generado por el compilador no es adecuado deberá escribir su propio constructor. El lenguaje C# le ayuda a hacerlo.

Si escribe su propio constructor, el compilador C# no generará un constructor predeterminado y será el suyo el que se utilice.

Los constructores son tipos especiales de métodos. Al igual que puede sobrecargar los métodos, también puede sobrecargar los constructores.

¿Qué es sobrecargar?

Sobrecargar es un término técnico para declarar dos o más métodos en el mismo alcance con el mismo nombre. El siguiente código muestra un ejemplo:

```
class Overload
{
    public void Method( ) ... }
    public void Method(int x) ... }
}
class Use
{
```

```

static void Main( )
{
    Overload o = new Overload( );
    o.Method( );
    o.Method(42);
}
}

```

En este ejemplo de código, dos métodos llamados **Method** se declaran en el alcance de la clase **Overload**, y ambos se invocan en **Use.Main**. No existe ambigüedad, debido a que el número y tipos de argumentos determinan que tipo de método se invoca.

Diapositiva 22

Constructores privados y estáticos

- Los constructores pueden tener diferentes modificadores de accesibilidad
- Constructores privados
 - Utilizarlos para evitar que los objetos se creen para una clase específica
- Constructores estáticos
 - Utilizarlos para estar seguro de que una clase siempre se inicializa antes de usarse

Constructores privados

Imaginemos que desea crear una clase para proporcionar cierta funcionalidad, pero no desea que los usuarios de esa clase creen objetos con base en esa clase. ¿Cómo lo lograría? Puede lograrlo utilizando un método estático únicamente dentro de la clase. En este caso, no hay necesidad de crear instancias en la clase (crear un objeto) para invocar su método.

Sin embargo queda un pequeño problema. El compilador generará un constructor predeterminado con acceso público, permitiéndole crear objetos.

Estos objetos pueden no tener ningún objetivo debido a que la clase contiene únicamente métodos estáticos.

Puede evitar que se generen estos objetos al declarar un constructor privado para la clase. Al declarar un constructor en la clase, evita que el compilador genere un constructor predeterminado, y también declara el constructor privado, los objetos ya no se podrán crear tampoco.

Constructores estáticos

Igual que un constructor de instancias garantiza que un objeto se encuentre en un estado inicial bien definido antes de ser utilizado, un constructor estático garantiza que la clase se encuentra en un estado inicial bien definido antes de ser utilizada.

Cargar clases en el tiempo de ejecución

C# es un lenguaje dinámico. Cuando el motor de ejecución común de los lenguajes se ejecuta en un programa basado en Microsoft® .NET, con frecuencia encuentra código que utiliza una clase que aún no se ha cargado. En estas situaciones, la ejecución se suspende momentáneamente, la clase se carga de manera dinámica y después continua la ejecución.

Inicializar clases en el tiempo de carga

C# asegura que una clase siempre se inicialice antes de que se utilice en el código en alguna forma. Esta garantía se logra al utilizar constructores estáticos.

Puede declarar un constructor estático como lo haría con un constructor de instancias, pero el prefijo debe ser la palabra clave **static**, como sigue:

```
class Example
{
    static Example( ) ... }
}
```

Una vez que el cargador de clase carga una clase que será utilizada pronto, pero antes de que continúe la ejecución normal, ejecuta el constructor estático para esa clase. Debido a este proceso, tiene garantizado que las clases siempre se inicialicen antes de ser utilizadas.

Diapositiva 23

Destruktores y terminadores

- Las acciones finales de los diferentes objetos serán diferentes
 - No se pueden determinar por recolección de residuos.
 - Los objetos en .NET Framework tienen un método de **Finalizar**.
 - Si está presente, la recolección de residuos llamará al destructor antes de reclamar la memoria binaria.
 - En C#, implementar un destructor para escribir el código de limpieza. No puede llamar o sobrescribir **Object.Finalize**
- Un destructor es el mecanismo de limpieza

```
class SourceFile
{
    ~SourceFile( ) { ... }
}
```

Ya hemos visto que destruir un objeto es un proceso de dos pasos. En el primer paso, el objeto se convierte de nuevo a memoria bruta. En el segundo, la memoria bruta regresa a la pila para ser reciclada. El proceso de recolección de residuos automatiza completamente el segundo paso de este proceso. Sin embargo, las acciones requeridas para finalizar el que un objeto específico regrese a la memoria bruta para limpiarlo dependerá del objeto específico. Esto significa que el recopilador de residuos no puede automatizar el primer paso. Si existe alguna instrucción específica que desee que ejecute un objeto cuando se recupera de la recolección de residuos y justo antes de que se reclame la memoria, necesita escribir estas instrucciones en el destructor.

Finalización

Cuando el recolector de residuos destruye un objeto no alcanzable, verificará si la clase del objeto cuenta con su propio destructor o método **Finalize**. Si la clase cuenta con un destructor o método **Finalize**, invocará el método antes de reciclar la memoria a la pila. Las instrucciones que escribe en el destructor o método **Finalize** serán específicas para la clase.

Destructores

Puede escribir un destructor para implementar la limpieza de un objeto. En C#, el método **Finalize** no está disponible directamente, y no puede invocar o

cancelar el método **Finalize**. Debe colocar código para que sea ejecutado durante la finalización dentro de un destructor.

El siguiente ejemplo muestra la sintaxis del destructor C# para una clase llamada **SourceFile**:

```
~ SourceFile( ) {  
    // Perform cleanup  
}
```

Debe utilizar el símbolo ~ para denotar el método Destructor.

Un destructor no tiene:

Un modificador de acceso.

Usted no llama al destructor; el recolector de residuos lo hace.

Un tipo de retorno.

El propósito del destructor es no devolver un valor, sino realizar las acciones requeridas de limpieza.

Parámetros.

De nuevo, no invoca al destructor, de manera que no puede pasarle ningún argumento. Observe que esto significa que el destructor no puede ser sobrecargado.

Diapositiva 24

Resumen de recursos

- Libros de referencia sobre la Programación orientada a los objetos

<http://support.microsoft.com/default.aspx?scid=kb%3ben-us%3b138569>

