Creación y destrucción de objetos

Contenido

Notas para el instructor

Este módulo proporciona a los estudiantes la teoría y la sintaxis necesarias para crear y destruir objetos en una aplicación C#.

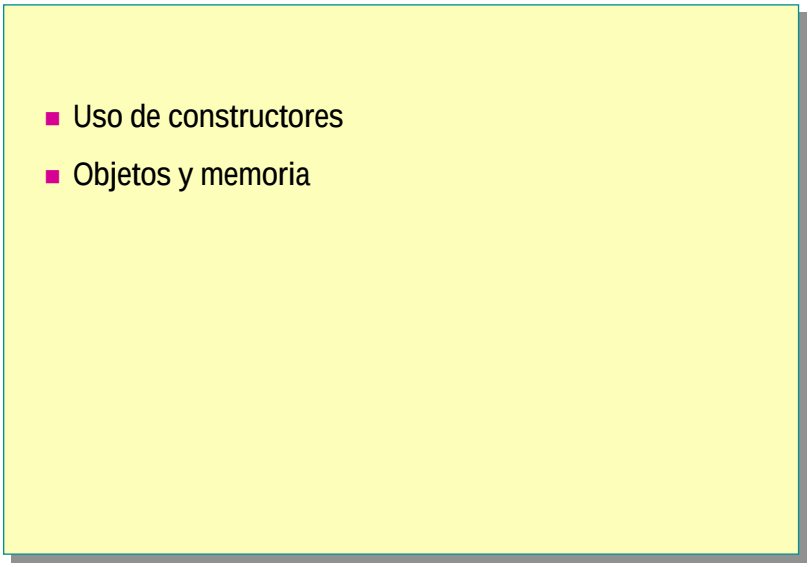
Al final de este módulo, los estudiantes serán capaces de:

- Crear objetos usando el operador **new**.
- Usar constructores para inicializar objetos.
- Describir el tiempo de vida de un objeto y qué ocurre cuando se destruye.
- Crear destructores.
- Heredar de la interfaz **IDisposable** y usar el método **Dispose**.

Descripción general

Objetivo del tema
Ofrecer una introducción a los contenidos y objetivos del módulo.

Explicación previa
En este módulo aprenderemos a controlar el proceso de creación y destrucción de objetos.



En este módulo veremos qué ocurre cuando se crea un objeto y cómo se usan constructores y destructores para inicializar y destruir objetos. También discutiremos qué ocurre cuando se destruye un objeto y cómo la recolección de basura consume memoria.

Al final de este módulo, usted será capaz de:

- Usar constructores para inicializar objetos.
- Crear constructores sobrecargados que pueden aceptar parámetros variables.
- Describir el tiempo de vida de un objeto y qué ocurre cuando se destruye.
- Crear destructores.
- Utilizar el método **Dispose**.

Uso de constructores

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta sección discutiremos los constructores y su uso para inicializar objetos.

- Creación de objetos
- Uso del constructor por defecto
- Sustitución del constructor por defecto
- Sobrecarga de constructores

Los constructores son métodos especiales que se utilizan para inicializar objetos cuando se crean. Aunque no se escriba ningún constructor, existe uno por defecto que se usa cuando se crea un objeto a partir de un tipo referencia.

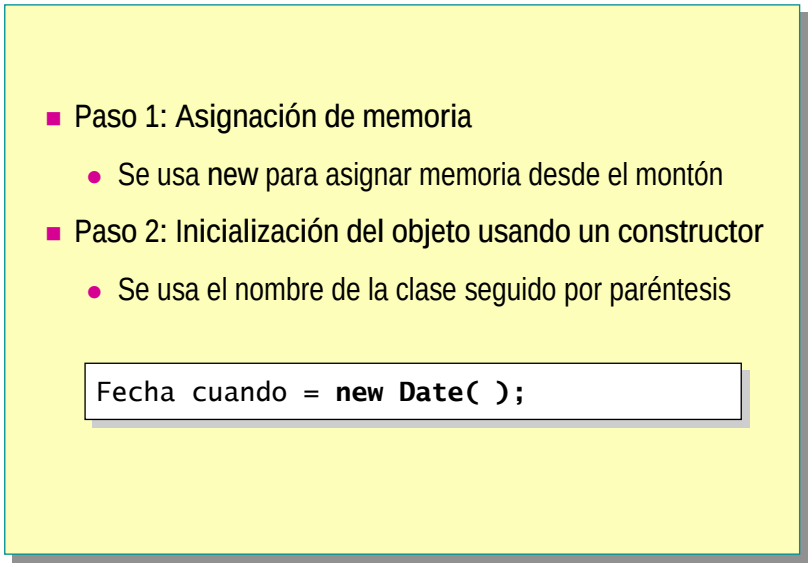
Al final de esta lección, usted será capaz de:

- Usar constructores por defecto.
- Usar constructores para controlar lo que ocurre cuando se crea un objeto.

Creación de objetos

Objetivo del tema
Describir el proceso de creación de un objeto.

Explicación previa
En C#, la única forma de crear un objeto es mediante el uso de la palabra reservada new para asignar memoria.



El proceso de creación de un objeto en C# se divide en dos pasos:

1. Usar la palabra reservada **new** para adquirir y asignar memoria para el objeto.
2. Escribir un constructor para convertir la memoria adquirida por **new** en un objeto.

Aunque este proceso consta de dos pasos, ambos deben estar incluidos en una sola expresión. Por ejemplo, si **Fecha** es el nombre de una clase, para asignar memoria e inicializar el objeto **cuando** se usa la siguiente sintaxis:

```
Date when = new Date( );
```

Paso 1: Asignación de memoria

El primer paso en la creación de un objeto consiste en asignarle memoria. Todos los objetos, sin excepción, se crean con el operador **new**, ya sea de forma explícita en el código o dejando que lo haga el compilador.

La siguiente tabla contiene ejemplos de código y lo que representan.

Ejemplo de código	Representa
string s = "Hola";	string s = **new** string(new char[ ]{'H','o','l','a'});
int[ ] array = {1,2,3,4};	int[ ] array = **new** int[4]{1,2,3,4};

Paso 2: Inicialización del objeto usando un constructor

El segundo paso en la creación de un objeto consiste en llamar a un constructor. Un constructor convierte en un objeto la memoria asignada por **new**. Hay dos tipos de constructores: constructores de instancia, que inicializan objetos, y constructores estáticos, que son los que inicializan clases.

Colaboración entre new y constructores de instancia

Es importante comprender el papel que juega en la creación de objetos la estrecha colaboración entre **new** y los constructores de instancia. Lo único que **new** hace es adquirir memoria binaria sin inicializar, mientras que el solo propósito de un constructor de instancia es inicializar la memoria y convertirla en un objeto que se pueda utilizar. En particular, **new** no participa de ninguna manera en la inicialización y los constructores de instancia no realizan ninguna función en la adquisición de memoria.

Aunque **new** y los constructores de instancia realizan tareas independientes, un programador no puede emplearlos por separado. De esta forma, C# contribuye a garantizar que la memoria está siempre configurada para un valor válido antes de que se lea (a esto se le llama *asignación definida*).

Nota para programadores de C++ En C++ es posible asignar memoria sin inicializarla (llamando directamente al operador **new**) e inicializar memoria asignada previamente (usando **new** de posición). Esta separación no está permitida en C#.

Uso del constructor por defecto

Objetivo del tema
Explicar lo que ocurre si no se escribe un constructor.

Explicación previa
Si el programador no escribe un constructor, el compilador se encargará de hacerlo.

- Características de un constructor por defecto
  - Acceso público
  - Mismo nombre que la clase
  - No tiene tipo de retorno (ni siquiera **void**)
  - No recibe ningún argumento
  - Inicializa todos los campos a **cero**, **false** o **null**
- Sintaxis del constructor

```
class Date { public Date( ) { ... } }
```

Si se crea un objeto sin escribir ningún constructor, el compilador de C# utilizará un constructor por defecto. Consideremos el ejemplo siguiente:

```
class Fecha
{
    private int ssaa, mm, dd;
}

class Test
{
    static void Main( )
    {
        Fecha cuando = new Fecha( );
        ...
    }
}
```

La instrucción en **Test.Main** crea un objeto **Fecha** llamado **cuando** usando **new** (que asigna memoria del montón) y llamando a un método especial que tiene el mismo nombre que la clase (el constructor de instancia). Sin embargo, la clase **Fecha** no declara ningún constructor de instancia (no declara ningún método). Por defecto, el compilador generará automáticamente un constructor de instancia.

Características de un constructor de instancia

Conceptualmente, el constructor de instancia generado por el compilador para la clase **Fecha** será parecido al siguiente:

```
class Fecha
{
    public Fecha( )
    {
        ssaa = 0;
        mm = 0;
        dd = 0;
    }
    private int ssaa, mm, dd;
}
```

El constructor presenta las siguientes características:

- Mismo nombre que la clase

  Por definición, un constructor de instancia es un método que tiene el mismo nombre que la clase a la que pertenece. Esta definición es natural e intuitiva y coincide con la sintaxis explicada anteriormente. Por ejemplo:

  ```
  Fecha cuando = new Fecha( );
  ```

- No tiene tipo de retorno

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

- No necesita argumentos

  Es posible declarar constructores que reciben argumentos, pero el constructor por defecto generado por el compilador no recibe ninguno.

- Todos los campos están inicializados a cero

  Esto es importante. El constructor por defecto generado por el compilador inicializa implícitamente todos los campos no estáticos de la siguiente manera:

  - Los campos numéricos (como **int**, **double** y **decimal**) se inicializan a cero.
  - Los campos de tipo **bool** se inicializan a **false**.
  - Los tipos referencia (tratados en un módulo anterior) se inicializan a **null**.
  - Los campos de tipo **struct** se inicializan de forma que contengan valores cero en todos sus elementos.

- Acceso público

  Esto permite crear nuevas instancias del objeto.

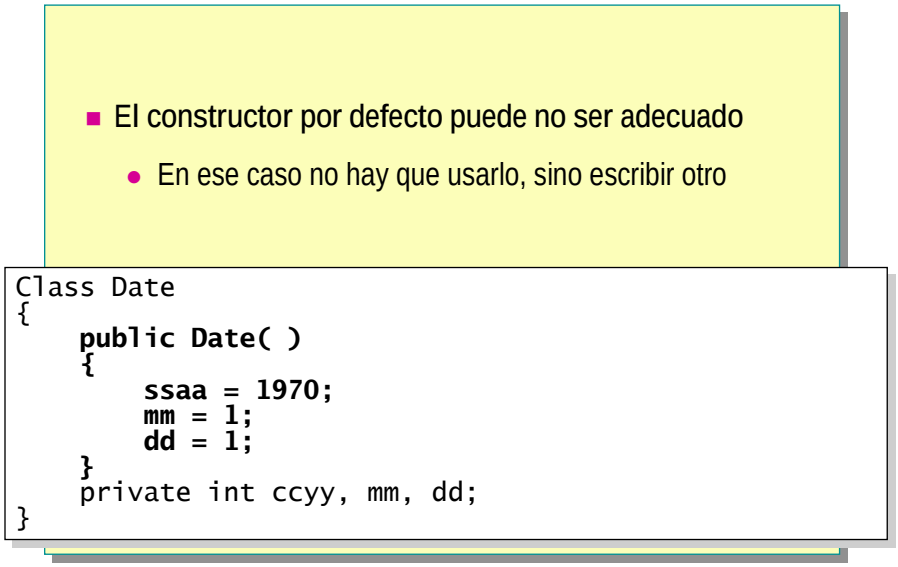
Para su información
Estas inicializaciones predeterminadas contribuyen a que el constructor por defecto generado por el compilador nunca lance una excepción, aunque no lo garantizan del todo. Podría haber una clase base con su propio constructor por defecto, que por supuesto se llamaría implícitamente con la sintaxis `:base( )` (este aspecto no se discutirá en este módulo).

Nota En el Módulo 10, "Herencia en C#", del Curso 2124C, *Programación en C#*, se discuten las bases abstractas. El constructor por defecto generado por el compilador para una clase abstracta tiene acceso protegido.

Sustitución del constructor por defecto

Objetivo del tema
Explicar qué hay que hacer si el constructor por defecto no es adecuado.

Explicación previa
En ocasiones, el constructor por defecto generado por el compilador no será el adecuado. En ese caso es mejor no utilizarlo.



En ocasiones no es conveniente emplear el constructor por defecto generado por el compilador. En esos casos se puede escribir otro constructor que contenga únicamente el código necesario para inicializar campos con valores distintos de cero. Todos los campos que no estén inicializados en este constructor conservarán su inicialización predeterminada a cero.

¿Qué ocurre si el constructor por defecto no es adecuado?

Hay varias situaciones en las que el constructor por defecto generado por el compilador puede no ser apropiado:

- No siempre se quiere que el acceso sea público.

  El patrón Factory Method utiliza un constructor que no es público. Este patrón, que se discute en el libro *Design Patterns: Elements of Reusable Object-Oriented Software*, de E. Gamma, R. Helm, R. Johnson y J. Vlissides, se tratará en un módulo posterior

  Las funciones procedurales (como **Cos** y **Sin**) utilizan con frecuencia constructores privados.

  El patrón Singleton usa generalmente un constructor privado. Este patrón también aparece en el libro *Design Patterns: Elements of Reusable Object-Oriented Software* y se discutirá en un tema posterior de esta sección.

- La inicialización a cero no siempre resulta conveniente.

  Consideremos el constructor por defecto generado por el compilador para la clase **Fecha** siguiente:

```
class Date
{
    private int ccyy, mm, dd;
}
```

El constructor por defecto inicializará el campo de año (*ssaa*) a cero, y lo mismo hará cono los campos de mes (*mm*) y de día (*dd*), lo que no resulta conveniente si se desea que el valor por defecto de la fecha sea distinto.

- El código invisible s más difícil de mantener.

  El código del constructor por defecto no se puede ver, lo que a veces puede resultar un problema. Por ejemplo, durante la depuración no es posible ejecutar paso a paso el código invisible. Por otra parte, si se utiliza la inicialización predeterminada a cero los desarrolladores que deban mantener el código no podrán saber si ha sido una elección deliberada o no.

Cómo escribir nuestro propio constructor

Si el constructor por defecto generado por el compilador no es adecuado, lo mejor es que escribamos nuestro propio constructor. El lenguaje C# nos ayuda a hacerlo.

Podemos escribir un constructor que contenga únicamente el código necesario para inicializar campos con valores distintos de cero. Todos los campos que no estén inicializados en este constructor conservarán su inicialización predeterminada a cero. El siguiente código muestra un ejemplo:

```
class DefaultInit
{
    public int a, b;
    public DefaultInit ( )
    {
        a = 42;
        // b conserva la inicialización predeterminada a cero
    }
}
class Test
{
    static void Main( )
    {
        DefaultInit di = new DefaultInit( );
        Console.WriteLine(di.a); // Writes 42
        Console.WriteLine(di.b); // Writes zero
    }
}
```

Para evitar errores, en los constructores propios es mejor no ir más allá de inicializaciones sencillas. La única forma adecuada de señalar un error de inicialización en un constructor consiste en lanzar una excepción.

Nota Esto también es válido para operadores. Los operadores se discuten en el Módulo 12, “Operadores, delegados y eventos” del Curso 2124C, *Programación en C#*.

El objeto se podrá usar si la inicialización tiene éxito; de lo contrario, es como si no existiera.

Sobrecarga de constructores

Objetivo del tema
Presentar la idea de que es posible sobrecargar constructores para disponer de más de una forma de inicializar objetos de una clase concreta.

Explicación previa
Los métodos pueden estar sobrecargados y los constructores también, puesto que los constructores no son más que un tipo especial de métodos.

- Los constructores son métodos y pueden estar sobrecargados
  - Mismo ámbito, mismo nombre, distintos parámetros
  - Permite inicializar objetos de distintas maneras
- AVISO
  - Si se escribe un constructor para una clase, el compilador no creará un constructor por defecto

```
Class Date
{
    public Date( ) { ... }
    public Date(int anno, int mes, int dia) { ... }
    ...
}
```

Los constructores son un tipo especial de métodos y pueden estar sobrecargados del mismo modo que los métodos.

¿Qué es la sobrecarga?

Sobrecarga es el término técnico para la declaración de dos o más métodos en el mismo ámbito y 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( )
    {
        Sobrecarga o = new Overload( );
        o. Method( );
        o. Method(42);
    }
}
```

En este ejemplo hay dos métodos llamados **Metodo** declarados en el ámbito de la clase **Sobrecarga**, y se hace una llamada a ambos en **Uso.Main**. No hay ninguna ambigüedad, puesto que el número y los tipos de los argumentos determinan el método que se está llamando.

Inicialización de un objeto de varias maneras

La posibilidad de inicializar un objeto de varias maneras es uno de los motivos más importantes para permitir la sobrecarga. Los constructores son un tipo especial de métodos y pueden estar sobrecargados exactamente del mismo modo que los métodos. Esto significa que es posible definir distintas formas de inicializar un objeto. El siguiente código muestra un ejemplo:

```
class Overload
{
    public Overload( ) { this.data = -1; }
    public Overload(int x) { this.data = x; }
    private int data;

}

class Use
{
    static void Main( )
    {
        Overload s1 = new Overload( );
        Overload s2 = new Overload(42);
        ...
    }
}
```

El objeto **s1** se crea usando el constructor que no recibe ningún argumento, y se asigna –1 a la variable de instancia privada *dato*. Por otra parte, el objeto **s2** se crea usando el constructor que recibe un argumento, y se asigna el valor 42 a la variable de instancia privada *dato*.

Inicialización de campos a valores distintos de los predeterminados

Hay muchas circunstancias en las que no resulta adecuado inicializar campos a cero. En esos casos se puede escribir otro constructor que reciba uno o más parámetros, que a su vez se emplean para inicializar los campos. Por ejemplo, consideremos la clase **Fecha** siguiente:

```
class Date
{
    public Date(int anno, int mes, int dia)
    {
        ccyy = anno;
        mm = mes;
        dd = dia;
    }
    private int ccyy, mm, dd;
}
```

Este constructor tiene el problema de que es fácil equivocarse en el orden de los argumentos. Por ejemplo:

```
Fecha cumpleannos = new Date(23, 11, 1968); // Error
```

El código debería ser `new Fecha(1968,11,23).` Este error no se detectará durante la compilación, ya que los tres argumentos son enteros. Una forma de corregir el problema sería usar el patrón Whole Value y convertir *Año*, *Mes* y *Día* en **struct**s en lugar de valores **int**:

```
struct Year
{
    public readonly int value;
    public Anno(int value) { this.value = value; }
}

struct Mes // O como enum
{
    public readonly int value;
    public Mes(int value) { this.value = value; }
}
struct Dia
{
    public readonly int value;
    public Dia(int valor) { this.value = value; }
}
class Date
{
    public Date (Anno y, Mes m, Dia d)
    {
        ccyy = y.value;
        mm = m.value;
        dd = d.value;
    }
    private int ccyy, mm, dd;
}
```

Consejo El uso de estructuras o enumeraciones en lugar de clases para *Día*, *Mes* y *Año* reduce la sobrecarga cuando se crea un objeto **Fecha**, como se explicará más adelante en este mismo módulo.

El siguiente código muestra un sencillo cambio que no sólo captura errores en el orden de los argumentos, sino que también permite crear constructores sobrecargados de **Fecha** con formato europeo, americano e ISO:

```
class Date
{
    public Date (Anno y, Mes m, Dia d) { ... } // ISO
    public Date (Mes m, Dia d, Anno y) { ... } // Americano
    public Date (Dia d, Mes m, Anno y) { ... } // Europeo
    ...
    private int ccyy, mm, dd;
}
```

La sobrecarga y el constructor por defecto

Si se declara una clase con un constructor, el compilador no genera el constructor por defecto. En el siguiente ejemplo, la clase **Fecha** se declara con un constructor, por lo que la expresión `new Fecha( )` no se compilará:

```
class Date
{
    public Date(Anno y, Mes m, Dia d) { ... }
    // No hay otros constructores
    private int ccyy, mm, dd;
}
class Fallo
{
    static void Main( )
    {
        Fecha predeterminada = new Date( ); // Error de
                                                compilación
    }
}
```

Esto significa que, si se quiere tener la posibilidad de crear objetos **Fecha** sin argumentos para el constructor, es necesario declarar explícitamente un constructor por defecto sobrecargado, como en el siguiente ejemplo:

```
class Date
{
    public Date( ) { ... }
    public Date(Anno y, Mes m, Dia d) { ... }
    ...
    private int ccyy, mm, dd;
}
class Vale
{
    static void Main( )
    {
        Fecha predeterminada = new Date( ); // Okay
    }
}
```

◆ Objetos y memoria

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En la sección anterior hemos aprendido qué ocurre cuando se crea un objeto. Ahora veremos qué pasa cuando se destruye un objeto.

- Tiempo de vida de un objeto
- Objetos y ámbito
- Recolección de basura

Es necesario saber qué ocurre en una aplicación cuando un objeto, y no un valor, está fuera de ámbito o se destruye.

Al final de esta lección, usted será capaz de:

- Identificar el papel jugado por la recolección de basura cuando un objeto está fuera de ámbito o se destruye.

Tiempo de vida de un objeto

Objetivo del tema
Examinar todo el ciclo de vida de un objeto desde la creación hasta su destrucción, pasando por el uso.

Explicación previa
En la primera sección vimos que el proceso de creación de un objeto tiene dos pasos, uno de los cuales se puede controlar. En esta sección veremos que la destrucción de un objeto también es un proceso en dos pasos de los que tampoco es posible controlar más que uno.

- Creación de objetos
  - Se usa new para asignar memoria
  - Se usa un constructor para inicializar un objeto en esa memoria
- Uso de objetos
  - Llamadas a métodos
- Destrucción de objetos
  - Se vuelve a convertir el objeto en memoria
  - Se libera la memoria

La destrucción de un objeto en C# es un proceso en dos pasos que corresponden, invirtiéndolos, a los dos pasos del proceso de creación del objeto.

Creación de objetos

En la primera sección vimos que el proceso de creación de un objeto en C# para un tipo referencia consta de los dos pasos siguientes:

1. Uso de la palabra reservada **new** para adquirir y asignar memoria.
2. Llamada a un constructor para convertir la memoria binaria adquirida por **new** en un objeto.

Destrucción de objetos

La destrucción de un objeto en C# también es un proceso en dos pasos:

1. Se anula la inicialización del objeto.

   El objeto es convertido de nuevo en memoria binaria. Este paso, que en C# se realiza con el destructor, es el inverso de la inicialización efectuada por el constructor. Es posible controlar lo que sucede en este paso escribiendo un destructor o método de finalización propio.

2. Se libera la memoria binaria; es decir, se devuelve la memoria al montón.

   Este paso es el inverso de la asignación realizada por **new** y no se puede controlar de ninguna manera.

Objetos y ámbito

Objetivo del tema
Insistir en que los valores locales se crean y se destruyen en momentos conocidos, y comparar esto con el caso de los objetos, cuyo momento de destrucción no se conoce.

Explicación previa
Los valores como ints y structs se asignan en la pila y se destruyen al final de su ámbito de validez. Por el contrario, los objetos se asignan en el montón administrado (Managed Heap) y no se destruyen al final de su ámbito de validez.

- El tiempo de vida de un valor a local está vinculado al ámbito en el que está declarado
  - Tiempo de vida corto (en general)
  - Creación y destrucción deterministas
- El tiempo de vida de un objeto dinámico no está vinculado a su ámbito
  - Tiempo de vida más largo
  - Destrucción no determinista

Al contrario de los valores como **int** y **struct**, que se asignan en la pila y se destruyen al final de su ámbito de validez, los objetos se asignan en el montón administrado (Managed Heap) y no se destruyen al final de su ámbito.

Valores

El tiempo de vida de un valor local está vinculado al ámbito en el que está declarado. Los valores locales son variables que se asignan en la pila y no en el montón administrado. Esto significa que, si se declara una variable cuyo tipo es uno de los primitivos (como **int**), **enum** o **struct**, no es posible usarla fuera del ámbito en el que se declara. Por ejemplo, en el siguiente fragmento de código se declaran tres valores dentro de una instrucción **for**, por lo que su ámbito terminará al final de esa instrucción:

```
struct Point { public int x, y; }
enum Season { Primavera, Verano, Otono, Invierno }
class Example
{
    void Method(int limite)
    {
        for (int i = 0; i < limite; i++) {
            int x = 42;
            Punto p = new Punto( );
            Season s = Season.Winter;
             ...
        }
        x = 42;                // Compile-time error
        p = new Point ( );     // Compile-time error
        s = Season.Winter; // Compile-time error
    }
}
```

Nota En el ejemplo anterior da la impresión de que se crea una **new Punto**. Pero **Punto** es una **struct** y **new** no asigna memoria del montón administrado. La "nueva" **Punto** *está* creada en la pila.

De aquí se deducen las siguientes características de los valores locales:

- Creación y destrucción deterministas

  Una variable local se crea en el momento de declararla y se destruye al final del ámbito en el que está declarada. El punto inicial y el punto final de la vida del valor son deterministas; es decir, tienen lugar en momentos conocidos y fijos.

- Tiempos de vida muy cortos por lo general

  Un valor se declara en alguna parte de un método y no puede existir más allá de una llamada al método. Cuando un método devuelve un valor, lo que se devuelve es una copia del valor.

Objetos

El tiempo de vida de un objeto no está vinculado al ámbito en el que se crea. Los objetos se inicializan en memoria del montón asignada mediante el operador **new**. Por ejemplo, en el siguiente código se declara la variable referencia *ej* dentro de una instrucción **for**, lo que significa que su terminará al final de esa instrucción y es un a variable local. Sin embargo, *ej* se inicializa con un objeto **new Example( )**, cuyo ámbito no se acaba con el de *ej*. Recordemos que una variable referencia y el objeto al que apunta son cosas distintas.

```
class Example
{
    void Method(int limit)
    {
        for (int i = 0; i < limit; i++) {
            Example ej = new Example( );
            ...
        }
        // ej está fuera de ámbito
        // ¿Sigue existiendo ej?
        // ¿Sigue existiendo el objeto?
    }
}
```

De aquí se deducen las siguientes características generales de los objetos:

- Destrucción no determinista

  Un objeto aparece cuando se crea pero, a diferencia de un valor, no se destruye al final del ámbito en el que se crea. La creación de un objeto es determinista, pero no así su destrucción. No es posible controlar exactamente cuándo se destruye un objeto.

- Tiempos de vida más largos

  Puesto que el tiempo de vida de un objeto no está vinculado al método que lo crea, un objeto puede existir mucho más allá de una llamada al método.

Recolección de basura

Objetivo del tema
Presentar la recolección de basura y explicar cómo funciona.

Explicación previa
En C# no es posible destruir objetos de forma explícita. En lugar de ello se utiliza un proceso llamado recolección de basura (Garbage Collection).

- No es posible destruir objetos de forma explícita
  - C# no incluye un inverso de new (como delete)
  - Ello se debe a que una función de eliminación explícita es una importante fuente de errores en otros lenguajes
- Los objetos se destruyen por recolección de basura
  - Busca objetos inalcanzables y los destruye
  - Los convierte de nuevo en memoria binaria no utilizada
  - Normalmente lo hace cuando empieza a faltar memoria

Hasta ahora hemos visto que los objetos en C# se crean exactamente de la misma forma que en otros lenguajes, como C++. Se utiliza la palabra reservada **new** para asignar memoria del montón y se hace una llamada a un constructor para convertir esa memoria en un objeto. Si embargo, no existen parecidos entre C# y sus predecesores por lo que se refiere al método para destruir objetos.

No es posible destruir objetos de forma explícita

Muchos lenguajes de programación permiten controlar de forma explícita cuándo se destruye un objeto. Por ejemplo, en C++ se puede usar una expresión **delete** para deshacer (o finalizar) la inicialización del objeto (convertirlo de nuevo en memoria) y devolver la memoria al montón. En C# no hay ninguna forma de destruir objetos explícitamente. Esta restricción es útil porque a menudo los programadores hacen mal uso de la destrucción de objetos. Por ejemplo:

- Se olvidan de destruir objetos.

  Si tuviéramos la responsabilidad de escribir el código que debe destruir un objeto, es probable que a veces olvidáramos hacerlo. Esto puede ocurrir en C++ y constituye un problema, ya que el programa usa más memoria y hace que el sistema sea más lento. Es lo que se conoce como *pérdida de memoria*. A menudo, la única forma de recuperar la memoria perdida es cerrar y volver a iniciar el programa en cuestión.

- Intentan destruir el mismo objeto más de una vez.

  A veces se puede intentar accidentalmente destruir el mismo objeto más de una vez. Esto puede ocurrir en C++ y es un error serio de consecuencias impredecibles. El problema es que, al destruir el objeto por primera vez, la memoria se libera y se puede usar para crear un nuevo objeto,

probablemente de una clase distinta. Si se vuelve a intentar destruir el mismo objeto, la memoria apunta aun objeto totalmente distinto.

- Destruyen un objeto activo.

  A veces se puede destruir un objeto que todavía estaba siendo utilizado por otra parte del programa. Se trata de un error grave que se conoce como *problema de puntero colgado* y que también tiene consecuencias impredecibles.

Para su información
Técnicamente es posible finalizar un objeto más de una vez si se usa la resurrección, pero no vale la pena mencionar este aspecto ya que no forma parte del temario del curso.

Los objetos se destruyen por recolección de basura

En C# no es posible destruir un objeto de forma explícita mediante código. En lugar de ello, C# tiene *recolección de basura* (Garbage Collection), que destruye objetos automáticamente. Esto garantiza que:

- Los objetos se destruyen.

  Sin embargo, la recolección de basura no indica cuándo exactamente se destruirá un objeto.

- Los objetos se destruyen sólo una vez.

  Esto significa que nunca se dará el comportamiento impredecible causado por una doble eliminación, como puede ocurrir en C++. Esto es importante porque ayuda a garantizar que un programa C# se comporta siempre de una forma bien definida.

- Sólo se destruyen los objetos inalcanzables.

  La recolección de basura garantiza que un objeto nunca será destruido si todavía hay algún otro objeto que apunte hacia él. Sólo se destruyen los objetos que ya no son utilizados por ninguna otra parte del programa. La capacidad de un objeto de acceder a otro a través de una variable referencia se llama *alcance* (reachability). Únicamente se destruyen objetos inalcanzables. La recolección de basura se encarga de seguir todas las referencias de objetos para determinar cuáles son alcanzables y, por un proceso de eliminación, encontrar los objetos inalcanzables. Esta operación puede llevar bastante tiempo, por lo que la recolección de basura sólo se activa para recuperar memoria no utilizada cuando empieza a faltar memoria.

Nota La recolección de basura se puede forzar de forma explícita en el código, pero es mejor no hacerlo. Lo más recomendable es dejar la administración de memoria al runtime de .NET.
