Uso de variables de tipo referencia

Contenido

Notas para el instructor

Este módulo ofrece a los estudiantes información detallada sobre tipos referencia. Define los tipos referencia y su relación con las clases. Explica las diferencias entre variables de tipo referencia y de tipo valor. También explica con detalle el tipo string como ejemplo de tipo referencia predefinido.

En este módulo se habla también de la jerarquía de clases **System.Object** y el tipo **object** en C#, mostrando las relaciones entre los distintos tipos. Igualmente se discuten espacios de nombres comunes en Microsoft® .NET Framework.

El módulo concluye explicando cómo convertir datos. Describe las conversiones explícitas para tratar datos de un tipo referencia como de otro. También explica cómo usar conversiones boxing y unboxing para convertir datos entre tipos referencia y tipos valor.

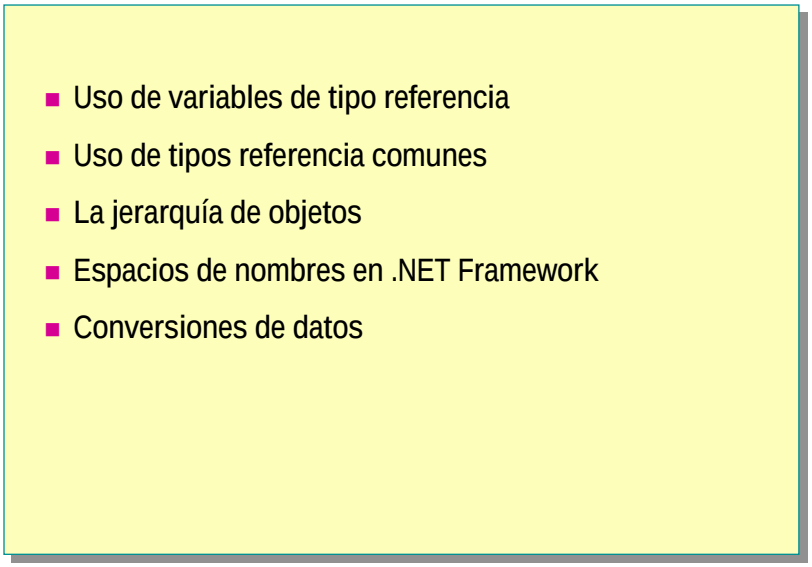
Al final de este módulo, los estudiantes serán capaces de:

- Describir las diferencias más importantes entre tipos referencia y tipos valor.
- Usar tipos referencia comunes, como string.
- Explicar cómo funciona el tipo **object** y familiarizarse con los métodos que proporciona.
- Describir espacios de nombres comunes en .NET Framework.
- Determinar si distintos tipos y objetos son compatibles.
- Hacer conversiones explícitas e implícitas de tipos de datos entre tipos referencia.
- Realizar conversiones boxing y unboxing entre datos de referencia y de valor.

Notas generales

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

Explicación previa
En este módulo discutiremos el uso de tipos referencia y estudiaremos la jerarquía de objetos en C#.



En este módulo aprenderemos a usar tipos referencia en C#. Discutiremos varios tipos referencia, como **string**, que están incluidos en el lenguaje C# y el entorno de tiempo de ejecución y que utilizaremos como ejemplos.

También hablaremos de la jerarquía de clases **System.Object** y en particular del tipo **object**, lo que nos permitirá comprender la relación de los tipos referencia entre sí y con los tipos valor. Estudiaremos cómo convertir datos entre tipos referencia usando conversiones explícitas e implícitas, y veremos cómo las conversiones boxing y unboxing convierten datos entre tipos referencia y tipos valor.

Al final de este módulo, usted será capaz de:

- Describir las diferencias más importantes entre tipos referencia y tipos valor.
- Usar tipos referencia comunes, como string.
- Explicar cómo funciona el tipo **object** y familiarizarse con los métodos que proporciona.
- Describir espacios de nombres comunes en .NET Framework.
- Determinar si distintos tipos y objetos son compatibles.
- Hacer conversiones explícitas e implícitas de tipos de datos entre tipos referencia.
- Realizar conversiones boxing y unboxing entre datos de referencia y de valor.

Uso de variables de tipo referencia

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

Explicación previa
En esta lección aprenderemos a usar tipos referencia en C#.

- Comparación de tipos valor y tipos referencia
- Declaración y liberación de variables referencia
- Referencias no válidas
- Comparación de valores y comparación de referencias
- Referencias múltiples a un mismo objeto
- Uso de referencias como parámetros de métodos

Recomendación al profesor
Aunque se da una breve explicación de cómo se libera memoria desde variables referencia, en este módulo no se discute en detalle la eliminación de elementos no utilizados (Garbage Collection o recolección de basura).

Los tipos referencia son muy importantes en el lenguaje C#, ya que permiten escribir aplicaciones complejas de altas prestaciones y hacer un uso eficaz del marco de trabajo en tiempo de ejecución.

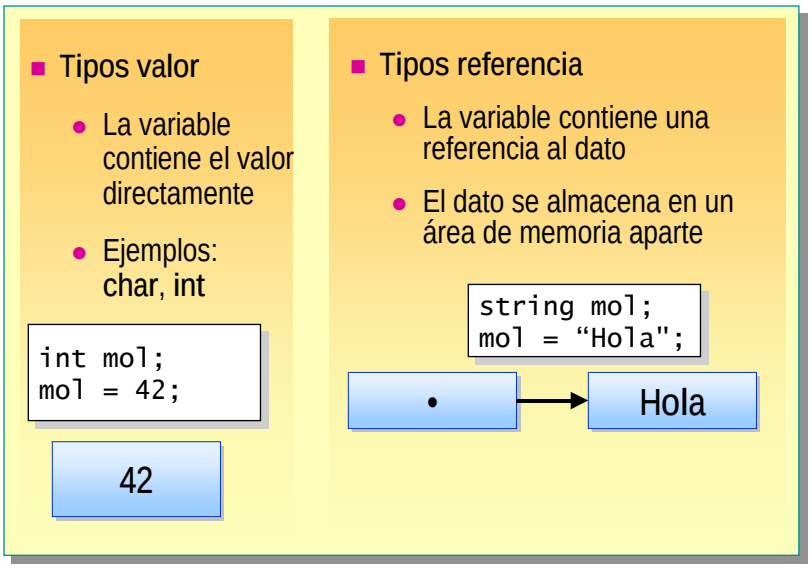
Al final de esta lección, usted será capaz de:

- Describir las diferencias más importantes entre tipos referencia y tipos valor.
- Usar y descartar variables referencia.
- Pasar tipos referencia como parámetros de métodos.

Comparación de tipos valor y tipos referencia

Objetivo del tema
Comparar tipos valor y tipos referencia.

Explicación previa
La mayor parte de los valores y variables que hemos visto hasta ahora en este curso son tipos valor. Vamos a hacer ahora una comparación entre tipos valor y tipos referencia.



Recomendación al profesor
Puede ser conveniente explicar que los tipos referencia no son lo mismo que los parámetros referencia, a pesar de la similitud de los nombres.

Probablemente también valga la pena discutir que C# tiene tipos valor porque son eficientes y se asignan en la pila, por lo que no es necesario hacer recolección de basura.

Recomendación al profesor
A los desarrolladores en C y C++ les puede ser útil comparar las variables referencia con punteros a objetos, pero insista en que las variables referencia en C# se especifican de forma totalmente segura y no pueden apuntar a objetos no válidos. Además, C y C++ no tienen recolección de basura automática.

C# admite tipos de datos básicos como **int**, **long** y **bool**, a los que también se llama *tipos valor*. Pero C# permite además otros datos más complejos y versátiles que se conocen como *tipos referencia*.

Tipos valor

Las variables de tipo valor son los tipos de datos básicos predeterminados, como char o int. Los tipos valor en C# son los más sencillos. Las variables de tipo valor contienen sus datos directamente.

Tipos referencia

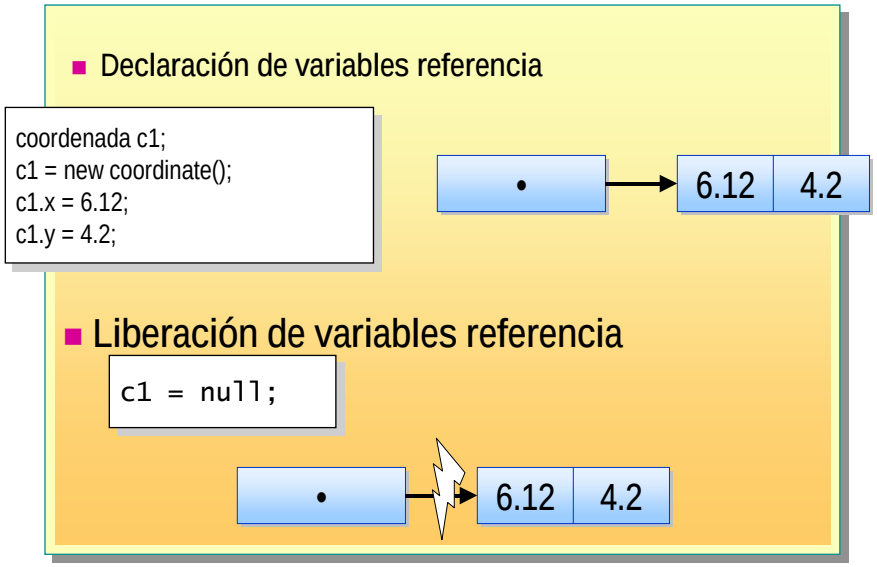
Las variables de tipo referencia no contienen datos, sino una referencia a los datos. Los datos se almacenan en áreas de memoria aparte.

En este curso ya hemos utilizado varios tipos referencia, tal vez sin ser conscientes de ello. Las tablas, las cadenas de caracteres y las excepciones son tipos referencia incluidas en el compilador de C# y en .NET Framework. También las clases, tanto predefinidas como definidas por el usuario, son en cierto modo un tipo referencia.

Declaración y liberación de variables referencia

Objetivo del tema
Explicar cómo se declaran e inicializan las variables referencia.

Explicación previa
Para usar variables de tipo referencia hay que saber cómo se declaran e inicializan y cómo se pueden liberar.



Para usar variables de tipo referencia hay que saber cómo se declaran e inicializan y cómo se pueden liberar.

Declaración de variables referencia

Para declarar variables de tipo referencia se utiliza la misma sintaxis que para variables de tipo valor:

```
coordinate c1;
```

En este ejemplo se declara una variable *c1* que puede tener una referencia a un objeto de tipo **coordinate**. Sin embargo, la variable no está inicializada hacia ningún objeto **coordinate**.

Para inicializar un objeto **coordinate** se emplea el operador **new**, que crea un objeto nuevo y devuelve una referencia al objeto que se puede almacenar en la variable de referencia.

Para su información
En C y C++, hay que usar un operador especial -> para eliminar una referencia a un puntero y acceder a un miembro. Esto no es necesario en C#.

```
coordinate c1;
c1 = new coordinate( );
```

También se puede declarar e inicializar la variable en una sola instrucción si se combina el operador **new** con la declaración de la variable, como vemos:

```
coordinate c1 = new coordinate( );
```

Una vez se ha creado en memoria un objeto al que apunta *c1*, es posible inicializar variables referencia de ese objeto mediante el operador **dot**, como en el siguiente ejemplo:

```
c1.x = 6.12;
c1.y = 4.2;
```

Ejemplo de declaración de variables referencia

Las clases son tipos referencia. El siguiente ejemplo ilustra cómo declarar una clase definida por el usuario llamada **coordinate**. Para simplificar, esta clase tiene sólo dos variables públicas: *x* e *y*.

```
class coordinate
{
    public double x = 0.0;
    public double y = 0.0;
}
```

Esta clase sencilla se usará en ejemplos posteriores para explicar cómo se crean, utilizan y destruyen variables referencia.

Recomendación al profesor
Esta clase tiene datos públicos que no están bien encapsulados, pero eso simplifica el ejemplo.

Liberación de variables referencia

Una vez asignada una referencia a un objeto nuevo, la variable de referencia continuará apuntando al objeto hasta que sea asignada para apuntar a un objeto diferente.

C# define un valor especial llamado **null**, que es el que contiene una variable de referencia cuando no apunta a ningún objeto válido. Se puede liberar una referencia asignando explícitamente el valor **null** a una variable de referencia (o simplemente dejando que la referencia quede fuera de ámbito).

Referencias no válidas

Objetivo del tema
Explicar cómo el compilador y el sistema de tiempo de ejecución tratan las referencias no válidas.

Explicación previa
No es posible acceder a variables o métodos si la referencia no es válida.

- Si hay referencias no válidas
  - No es posible acceder a miembros o variables
- Referencias no válidas en tiempo de compilación
  - El compilador detecta el uso de referencias no inicializadas
- Referencias no válidas en tiempo de ejecución
  - El sistema generará un error de excepción

Sólo se puede acceder a los miembros de un objeto a través de una variable de referencia si ésta ha sido inicializada par que apunte a una referencia válida. Si la referencia no es válida, no es posible acceder a variables o métodos.

El compilador puede detectar este problema en algunos casos. En otras ocasiones es necesario detectarlo y corregirlo en tiempo de ejecución.

Referencias no válidas en tiempo de compilación

El compilador puede detectar situaciones en las que no se inicializa una variable de referencia antes de utilizarla.

Por ejemplo, si una variable **coordinate** se declara pero no se inicializa, se recibirá un mensaje de error parecido al siguiente: “Uso de variable local no asignada c1.” El siguiente código muestra un ejemplo:

```
coordinate c1;
c1.x = 6.12; // Will fail: variable not assigned
```

Referencias no válidas en tiempo de ejecución

En general no es posible determinar en tiempo de compilación si la referencia de una variable es válida o no. Como consecuencia, C# comprueba el valor de una variable de referencia antes de su uso para asegurarse de que no es **null**.

Si se intenta utilizar una variable de referencia cuyo valor es **null**, el sistema de tiempo de ejecución lanza una excepción **NullReferenceException**. Esta situación se puede prevenir utilizando **try** y **catch**, como muestra el siguiente ejemplo:

```
try {
    c1.x = 45;
}
catch (NullReferenceException) {
    Console.WriteLine("c1 tiene valor null");
}
```

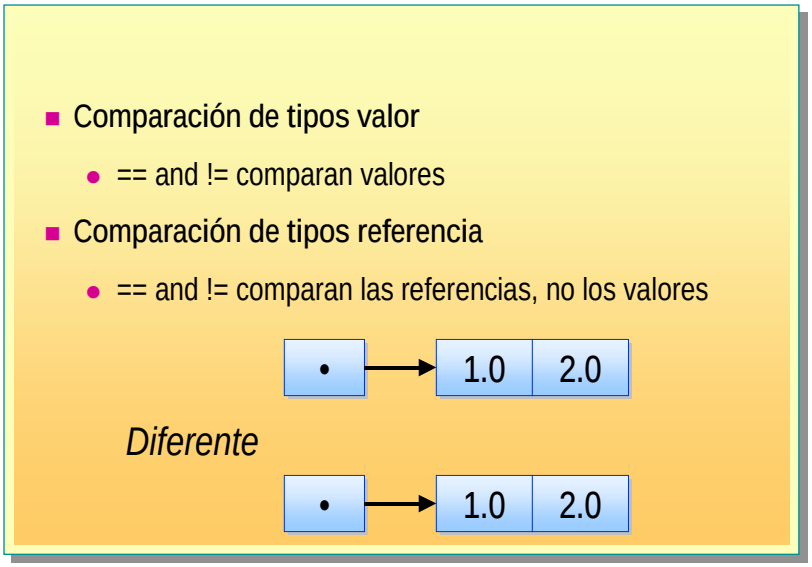
Esta comprobación también se puede efectuar de forma explícita para evitar excepciones. El siguiente ejemplo muestra cómo comprobar que una variable de referencia contiene una referencia distinta de **null** antes de intentar acceder a sus miembros:

```
if (c1 != null)
    c1.x = 45;
else
    Console.WriteLine("c1 tiene valor null");
```

Comparación de valores y comparación de referencias

Objetivo del tema
Explicar el uso predeterminado de los operadores == and != para variables referencia.

Explicación previa
Los operadores == and != pueden no funcionar del modo esperado con variables referencia.



Los operadores de igualdad (==) y desigualdad (!=)pueden no funcionar del modo esperado con variables referencia.

Comparación de tipos valor

Para tipos valor, los operadores == and **!=** se pueden usar para comparar valores.

Comparación de tipos referencia

Para tipos referencia que no sean **string**, los operadores == and **!=** determinan si las dos variables referencia están apuntando al mismo objeto, pero *no* comparan los contenidos de los objetos a los que apuntan las variables. En el caso del tipo **string**, == compara el valor de las cadenas de caracteres.

Para su información
C y C++ usan una semántica similar para la comparación de punteros. Visual Basic emplea el operador Is para comparar dos referencias.

Consideremos el siguiente ejemplo, en el que dos variables de coordenada se crean e inicializan con los mismos valores:

```
coordenada c1= new coordinate( );
coordenada c2= new coordinate ( );
c1.x = 1.0;
c1.y = 2.0;
c2.x = 1.0;
c2.y = 2.0;
if (c1 == c2)
    Console.WriteLine("Same ");
else
    Console.WriteLine("Different ");
```

La salida de este código es “Diferentes”. Aunque los objetos a los que apuntan tienen los mismos valores, *c1* y *c2* son referencias a distintos objetos y por tanto == devuelve **false**.

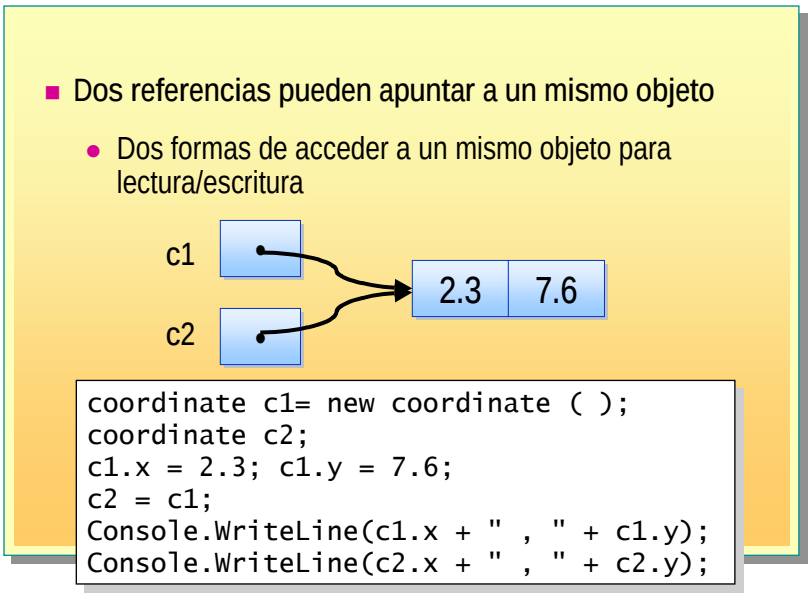
Recomendación al profesor
Los desarrolladores en C y C++ pueden sorprenderse de que no se puedan usar los operadores <, >, <= and >= con referencias. Esto se debe a que las referencias en C# no son direcciones.

Los otros operadores relacionales (<, >, <= y >=) no se pueden utilizar para comparar si dos variables referencia están apuntando a un objeto.

Referencias múltiples a un mismo objeto

Objetivo del tema
Explicar que dos variables referencia pueden apuntar a un mismo objeto.

Explicación previa
El hecho de que las variables de tipo referencia contengan una referencia a los datos permite que dos de ellas puedan apuntar al mismo objeto.



Dos variables referencia pueden apuntar al mismo objeto porque lo que contienen es una referencia a los datos. Esto significa que es posible utilizar una referencia para escribir datos y otra para leer datos.

Referencias múltiples a un mismo objeto

En el siguiente ejemplo, la variable *c1* se inicializa para que apunte a una instancia nueva de la clase y se inicializan sus variables *x* e *y*. A continuación se copia *c1* a *c2*. Finalmente se muestran los valores de los objetos a los que apuntan *c1* y *c2*.

```
coordenada c1 = new coordinate( );
coordenada c2;
c1.x = 2.3;
c1.y = 7.6;
c2 = c1;
Console.WriteLine(c1.x + " , " + c1.y);
Console.WriteLine(c2.x + " , " + c2.y);
```

La salida de este programa es la siguiente:

```
2.3 , 7.6
2.3 , 7.6
```

La asignación de *c2* a *c1* copia la referencia, por lo que ambas variables apuntan al mismo objeto. Como consecuencia, los valores que se imprimen para las variables de *c1* y *c2* son iguales.

Escritura y lectura de los mismos datos con distintas referencias

En el siguiente ejemplo se ha añadido una asignación inmediatamente antes de las llamadas a **Console.WriteLine**.

```
coordenada c1 = new coordinate( );
coordenada c2;
c1.x = 2.3;
c1.y = 7.6;
c2 = c1;
c1.x = 99; // Ésta es la nueva instrucción
Console.WriteLine(c1.x + " , " + c1.y);
Console.WriteLine(c2.x + " , " + c2.y);
```

La salida de este programa es la siguiente:

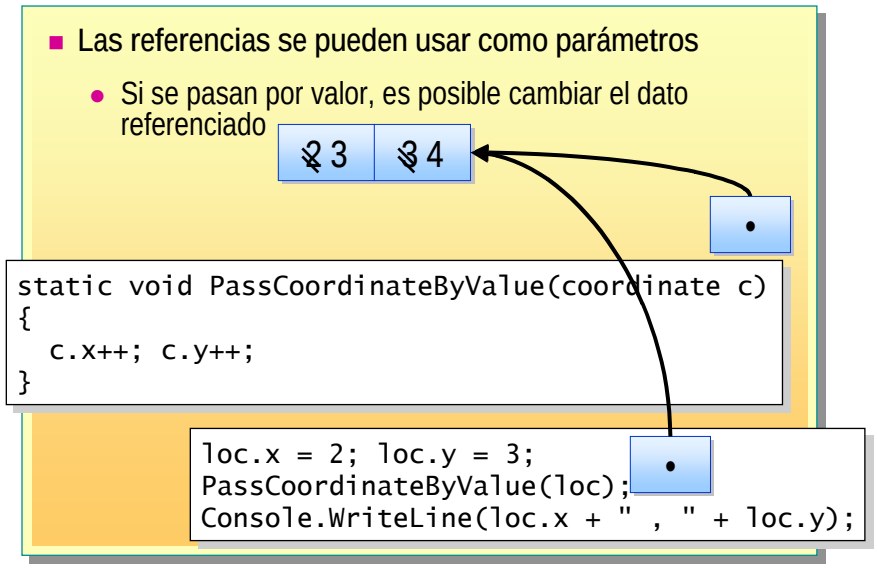
```
99 , 7.6
99 , 7.6
```

Como se ve, la asignación de 99 a *c1.x* cambia también *c2.x*. Como la referencia en *c1* se había asignado previamente a *c2*, un programa puede escribir datos por una referencia y leer los mismos datos por otra referencia.

Uso de referencias como parámetros de métodos

Objetivo del tema
Explicar que se puede pasar referencias a y desde métodos de distintas maneras.

Explicación previa
Es posible pasar variables referencia a métodos o fuera de ellos.



Es posible pasar variables referencia a métodos o fuera de ellos.

Referencias y métodos

Para pasar variables referencia como parámetros a métodos se puede emplear cualquiera de los tres mecanismos de llamada:

- Por valor
- Por referencia
- Parámetros de salida

El siguiente ejemplo muestra un método que pasa tres referencias de coordenada. La primera se pasa por valor, la segunda por referencia y la tercera es un parámetro de salida. El valor de devolución del método es una referencia de coordenada.

```
static coordinate Example(
  coordinate ca,
  ref coordinate cb,
  out coordinate cc)
{
  // ...
}
```

Paso de referencias por valor

Si se utiliza una variable de referencia como parámetro valor, el método recibe una copia de la referencia. Esto significa que, mientras dure la llamada, hay dos referencias que apuntan al mismo objeto. Por ejemplo, el siguiente código muestra los valores 3 y 4:

```
static void PassCoordinateByValue (coordenada c)
{
  c.x++; c.y++; //c referencing loc
}
coordinate loc = new coordinate( );
loc.x = 2; loc.y = 3;
PassCoordinateByValue (loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

También significa que los cambios en el parámetro del método no afectan a la referencia que hace la llamada. Por ejemplo, el siguiente código muestra los valores 0 y 0:

```
static void PassCoordinateByValue (coordenada c)
{
    c = new coordinate( ); //c no longer referencing loc
    c.x = c.y  = 22.22;
}
coordinate loc = new coordinate( );
PassCoordinateByValue (loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Paso de referencias por referencia

Si se utiliza una variable de referencia como parámetro **ref**, el método recibe la variable de referencia. Al contrario de lo que ocurre cuando se pasa por valor, en este caso hay sólo una referencia. El método *no* hace su propia copia. Esto significa que los cambios en el parámetro del método afectarán a la referencia que hace la llamada. Por ejemplo, el siguiente código muestra los valores 33.33 y 33.33:

```
static void PassCoordinateByRef(ref coordenada c)
{
    c = new coordinate ( );
    c.x = c.y  = 33.33;
}
coordinate loc = new coordinate( );
PassCoordinateByRef(ref loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Paso de referencias por salida

Si se utiliza una variable de referencia como parámetro **out**, the el método recibe la variable de referencia. Al contrario de lo que ocurre cuando se pasa por valor, en este caso hay sólo una referencia. El método *no* hace su propia copia. El paso por **out** es similar al paso por **ref**, salvo que el método *debe* tener el parámetro **out**. Por ejemplo, el siguiente código muestra los valores 44.44 y 44.44:

```
static void PassCoordinateByOut(out coordenada c)
{
    c = new coordinate( );
    c.x = c.y  = 44.44;
}
coordenada loc = new coordinate( );
PassCoordinateByOut(out loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Paso de referencias a métodos

Las variables de tipos referencia no contienen directamente un valor, sino una referencia a un valor. Lo mismo ocurre si son parámetros de métodos, lo que significa que el mecanismo de paso por valor puede producir resultados inesperados.

Recomendación al profesor
Explique que lo que se pasa al método es la referencia, no el objeto al que apunta.

Comente también que las variables referencia no son lo mismo que el paso por referencia, a pesar de la similitud de los nombres.

Consideremos como ejemplo el siguiente método de la clase **coordenada**:

```
static void PassCoordinateByValue(coordenada c)
{
    c.x++;
    c.y++;
}
```

El parámetro **coordinate** *c* se pasa por valor. En el método se incrementan las dos variables miembro *x* e *y*. Veamos ahora el siguiente código, que llama al método **PassCoordinateByValue**:

```
coordinate loc = new coordinate( );
loc.x = 2;
loc.y = 3;
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

La salida de este código es la siguiente:

```
3 , 4
```

Como se ve, el método ha cambiado los valores referenciados por **loc**. Puede parecer que esto contradice la explicación de paso por valor dada anteriormente en el curso, pero de hecho no es así. La variable referencia *loc* se copia en el parámetro *c* y no puede ser modificada por el método, pero la memoria a la que apunta no se copia y por tanto no está sujeta a esa limitación. La variable *loc* continúa apuntando a la misma área de memoria, pero ahora ésta contiene un dato distinto.

Uso de tipos referencia comunes

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

Explicación previa
Algunos tipos referencia están predefinidos en el lenguaje C#.

- Clase Exception
- Clase String
- Métodos, operadores y propiedades comunes de String
- Comparaciones de cadenas de caracteres
- Operadores de comparación en String

Algunos tipos referencia están predefinidos en el lenguaje C#.

Al final de esta lección, usted será capaz de:

- Describir el funcionamiento de clases predefinidas.
- Usar clases predefinidas como modelos para crear sus propias clases.

Clase Exception

Objetivo del tema
Describir la clase Exception.

Explicación previa
Exception es una clase cuyos objetos se crean para lanzarlos en caso de excepción.

- Exception es una clase
- Los objetos Exception se usan para lanzar excepciones
  - Para crear un objeto Exception se usa new
  - Para lanzar el objeto se usa throw
- Los tipos Exception son subclases de Exception

Los objetos **Exception** se crean para lanzarlos en caso de excepción.

Clase Exception

Exception es el nombre de una clase incluida en .NET Framework.

Objetos Exception

Los objetos de tipo **Exception** son los únicos que se pueden lanzar con **throw** y capturar con **catch**. Por lo demás, la clase **Exception** es similar a otros tipos referencia.

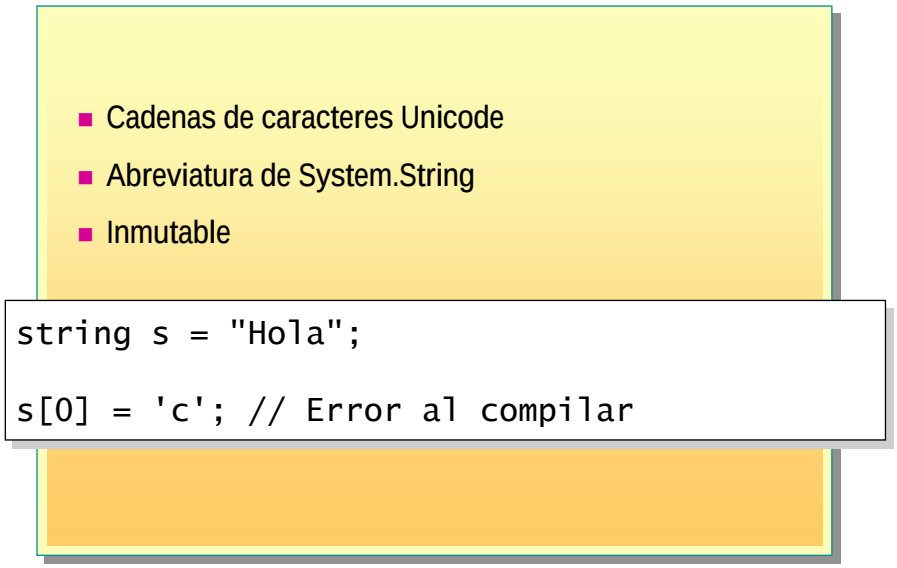
Tipos Exception

Exception representa un fallo genérico en una aplicación. También existen tipos para excepciones concretas (como **InvalidCastException**). Cada una de las clases que heredan de **Exception** representa una de estas excepciones concretas.

Clase String

Objetivo del tema
Describir con más detalle la clase String.

Explicación previa
El tipo string se utiliza para procesar cadenas de caracteres.



Recomendación al profesor
El procesamiento de cadenas de caracteres en C# se realiza a mucho más alto nivel que en C. Por ejemplo, es posible sobrescribir caracteres especiales de terminación. La recolección de basura (no discutida en este módulo) también se hace automáticamente.

En C# se utiliza el tipo string para procesar cadenas de caracteres Unicode (en contraste, el tipo char es un tipo valor para caracteres individuales).

El nombre del tipo **string** es una abreviatura de la clase **System.String**. El compilador procesa esta forma abreviada, por lo que **string** y **System.String** se pueden usar indistintamente.

La clase **String** representa una cadena inmutable de caracteres. Una instancia de **String** es inmutable, lo que significa que el texto de una cadena no se puede modificar una vez creado. A primera vista podría parecer que algunos métodos modifican un valor string, pero en realidad devuelven una nueva instancia de string que contiene la modificación.

Consejo A menudo se usa la clase **StringBuilder** en combinación con la clase **String**. **StringBuilder** crea una cadena modificable internamente y que una vez completa se puede convertir en una **String** inmutable. **StringBuilder** elimina la necesidad de crear repetidamente **String**s temporales inmutables y puede mejorar las prestaciones del código.

La clase **System.String** tiene muchos métodos. La información sobre procesamiento de cadenas de caracteres que se incluye en este curso no es completa, pero sí se tratan algunos de los métodos más útiles. Para más información, consulte los documentos de ayuda del kit de desarrollo de software (SDK) de .NET Framework.

Métodos, operadores y propiedades comunes de String

Objetivo del tema
Indicar y describir algunos métodos, operadores y propiedades comunes de la clase String.

Explicación previa
Presentamos aquí algunos ejemplos de métodos, operadores y propiedades de System.String.

- Corchetes
- Método Insert
- Propiedad Length
- Método Copy
- Método Concat
- Método Trim
- Métodos ToUpper y ToLower

Recomendación al profesor
Las páginas que siguen no contienen una lista completa de todos los métodos de la clase String. Puede ser conveniente que utilice ahora archivos de referencia para explicar a los estudiantes cómo pueden buscar más información.

Corchetes [ ]

Para extraer un solo carácter en una posición determinada de una cadena de caracteres utilizando el nombre de la cadena seguido del índice entre corchetes ([and]). Este proceso es similar al empleado para tablas. El índice del primer carácter de una cadena es cero.

El siguiente código muestra un ejemplo:

```
string s = "Alphabet";
char firstchar = s[2]; // 'p'
```

No está permitido emplear los corchetes para asignar un carácter, ya que las cadenas son inmutables. Si se intenta asignar un carácter a una cadena de esta forma, como en el ejemplo, se producirá un error en tiempo de compilación.

```
s[2] = '*'; // Not valid
```

Método Insert

Para insertar caracteres en una variable se puede emplear el método **Insert**, que devuelve una nueva string con un valor especificado insertado en la posición indicada de la cadena. Este método recibe dos parámetros: la posición de inicio de la inserción y la cadena que se desea insertar.

El siguiente código muestra un ejemplo:

```
string s = "¡Me encanta C!";
s = s.Insert(2, " Sharp ");
Console.WriteLine(s); // ¡Me encanta C Sharp!
```

Propiedad Length

La propiedad **Length** devuelve un entero que representa la longitud de la cadena:

```
string msj = "Hello";
int slen = msj.Length; // 5
```

Método Copy

El método de clase **Copy** copia una cadena para crear otra nueva; es decir, hace un duplicado de una cadena especificada.

El siguiente código muestra un ejemplo:

```
string s1 = "Hello";
string s2 = String.Copy(s1);
```

Método Concat

El método de clase **Concat** crea una nueva cadena de caracteres a partir de una o más cadenas o de objetos representados como cadenas.

El siguiente código muestra un ejemplo:

```
string s3 = String.Concat("a", "b", "c", "d", "e", "f", "g");
```

El operador + está sobrecargado para cadenas, por lo que el ejemplo anterior se podría reescribir de la siguiente manera:

```
string s = "a" + "b" + "c" + "d" + "e" + "f" + "g";
Console.WriteLine(s);
```

Método Trim

El método **Trim** elimina todos los caracteres indicados o escribe espacios entre el principio y el final de una cadena.

El siguiente código muestra un ejemplo:

```
string s = "    Hello  ";
s = s.Trim( );
Console.WriteLine(s); // "Hello"
```

Métodos ToUpper y ToLower

Los métodos **ToUpper** y **ToLower** devuelven una cadena con todos los caracteres cambiados a mayúsculas y minúsculas, respectivamente:

```
string sText = "El camino al triunfo ";
Console.WriteLine(sText.ToUpper( )); // EL CAMINO AL TRIUNFO
Console.WriteLine(sText.ToLower( )); // el camino al triunfo
```

Comparaciones de cadenas de caracteres

Objetivo del tema
Describir los operadores de comparación que se emplean con cadenas de caracteres.

Explicación previa
Se pueden utilizar los operadores == and != sobre variables string para comparar sus contenidos.

- Método Equals
  - Comparación de valores
- Método Compare
  - Más comparaciones
  - Opción para no distinguir mayúsculas y minúsculas
  - Ordenación alfabética
- Opciones locales de Compare

Se pueden utilizar los operadores == and != sobre variables string para comparar sus contenidos.

Método Equals

La clase **System.String** contiene un método llamado **Equals** que se puede emplear para comparar dos cadenas y ver si son iguales. El método devuelve un valor **bool** que es **true** si las cadenas son iguales y **false** en caso contrario. Es un método sobrecargado y se puede usar como método de instancia o como método estático. Ambas formas se muestran en el siguiente ejemplo.

```
string s1 = "Bienvenidos";
string s2 = "Bienvenidos";

if (s1.Equals(s2))
    Console.WriteLine("Las cadenas son iguales");

if (String.Equals(s1,s2))
    Console.WriteLine("Las cadenas son iguales");
```

Método Compare

El método **Compare** compara dos cadenas desde el punto de vista léxico; es decir, compara las cadenas según su ordenación. **Compare** puede devolver los siguientes valores:

- Un entero negativo si la primera cadena es anterior a la segunda
- 0 si las cadenas son iguales
- Un entero positivo si la primera cadena es posterior a la segunda

```
string s1 = "Supercalifragilístico";
string s2 = "Velocípedo";
int comp = String.Compare(s1,s2);  // Entero negativo
```

Por definición, cualquier cadena (incluyendo una cadena vacía) es siempre mayor que una referencia **null**, y dos referencias **null** se consideran iguales al compararlas.

Compare es un método sobrecargado. Existe una versión con tres parámetros, el tercero de los cuales es un valor **bool** si en la comparación se deben ignorar las mayúsculas. El siguiente ejemplo muestra una comparación que no distingue mayúsculas y minúsculas:

```
s1 = "repollo";
s2 = "Repollo";
comp = String.Compare(s1, s2, true); // Ignorar mayúsculas
```

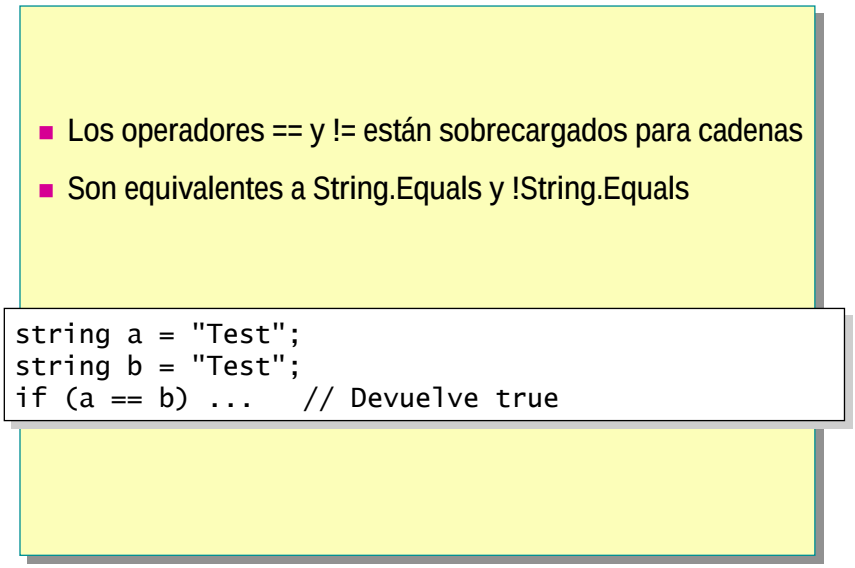
Opciones locales de Compare

Existen versiones sobrecargadas del método **Compare** que permiten comparar cadenas a partir de ordenaciones en distintos idiomas. Esto es útil cuando se escriben aplicaciones para un mercado internacional. Este aspecto no está incluido en el temario de este curso. Para más información, busque “espacio de nombres System.Globalization” y “clase CultureInfo” en los documentos de ayuda del SDK de .NET Framework.

Operadores de comparación en String

Objetivo del tema
Explicar cómo usar los operadores de comparación sobrecargados para cadenas de caracteres.

Explicación previa
Los operadores == and != están sobrecargados para la clase String.



Los operadores == and != están sobrecargados para la clase **String**. Estos operadores sirven para examinar los contenidos de cadenas de caracteres.

```
string a = "Test";
string b = "Test";
if (a == b) ...  // Devuelve true
```

Los siguientes operadores y métodos son equivalentes:

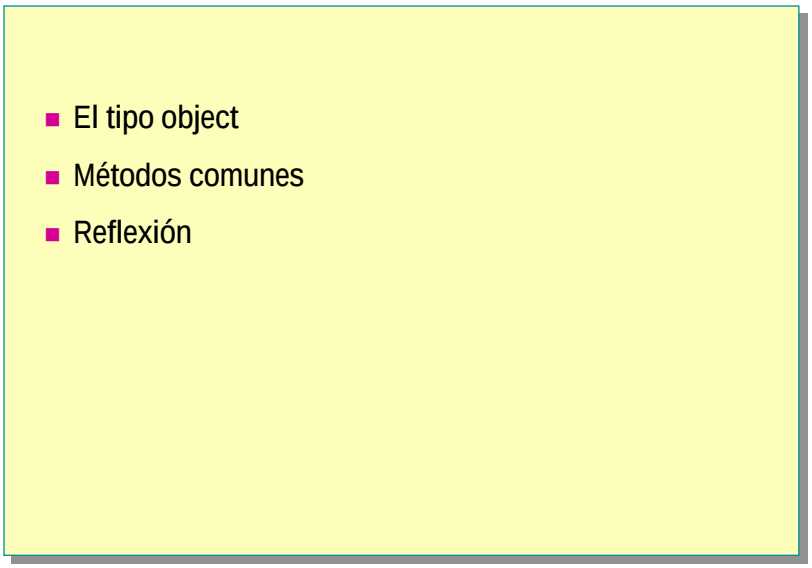
- El operador == es equivalente al método **String.Equals**.
- El operador!= es equivalente al método**!String.Equals**.

Los otros operadores relacionales (<, >**,** <= y >=) no están sobrecargados para la clase **String**.

La jerarquía de objetos

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

Explicación previa
Las clases base se organizan en una jerarquía en cuya cima está la clase **Object**.



Todas las clases es .NET son derivadas de la clase **System.Object**. El tipo **object** es un alias de la clase System.Object en lenguaje C#.

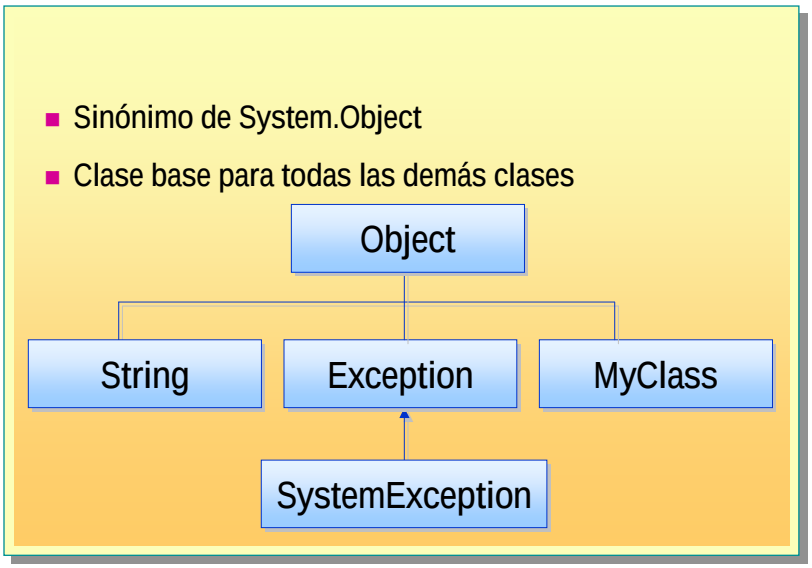
Al final de esta lección, usted será capaz de:

- Describir el funcionamiento de la jerarquía de objetos.
- Usar el tipo **object**.

El tipo object

Objetivo del tema
Presentar el tipo **object**.

Explicación previa
El tipo **object** es la clase base pasa todas las demás clases en C#.



El tipo **object** es la clase base pasa todas los demás tipos en C#.

System.Object

La palabra reservada **object** es un sinónimo de la clase **System.Object** en.NET Framework. Siempre que aparezca la palabra **object** se puede poner en su lugar el nombre de clase **System.Object**. Normalmente se emplea la forma más corta.

Recomendación al profesor
La herencia se discute en el Módulo 10, "Herencia en C#", del Curso 2124C, *Programación en C#*. Por el momento no es necesario que discuta en profundidad el concepto de herencia.

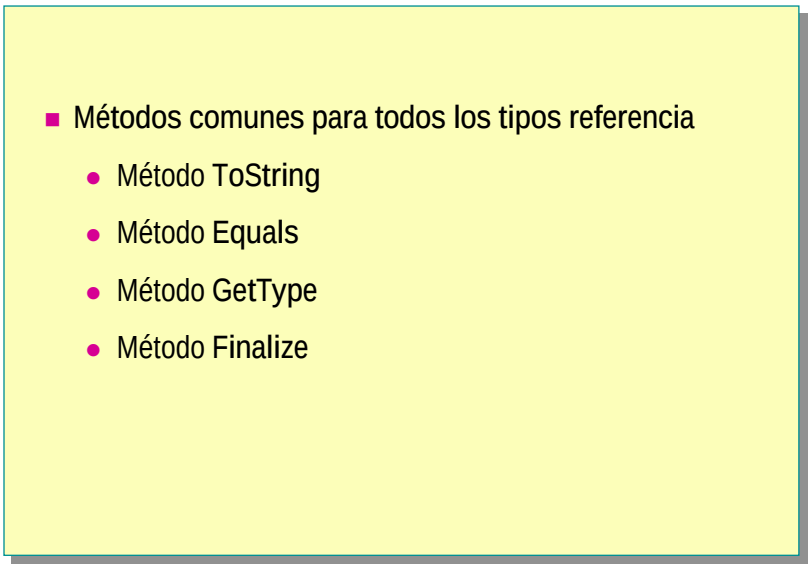
Clase base

Todas las clases, incluidas las que se escriben para una aplicación o las que forman parte del marco de trabajo del sistema, heredan directa o indirectamente de **object**. Cuando se declara una clase sin una clase padre explícita, lo que se está haciendo en realidad es heredar de **object**.

Métodos comunes

Objetivo del tema
Presentar algunos de los métodos comunes del tipo object.

Explicación previa
El tipo object contiene varios métodos comunes que heredan todos los demás tipos referencia.



El tipo **object** contiene varios métodos comunes que heredan todos los demás tipos referencia.

Métodos comunes para todos los tipos referencia

El tipo **object** proporciona varios métodos comunes. Los métodos derivados también incluyen estos métodos, ya que todos ellos heredan de **object**. Entre estos métodos comunes están los siguientes:

- **ToString**
- **Equals**
- **GetType**
- **Finalize**

Método ToString

El método **ToString** devuelve una cadena de caracteres que representa el objeto actual.

Su forma predeterminada, que se encuentra en la clase **Object**, devuelve el nombre de tipo de la clase. El siguiente ejemplo utiliza la clase **coordinate** definida anteriormente:

```
coordenada c = new coordinate( );
Console.WriteLine(c.ToString( ));
```

Este ejemplo escribirá “coordenada” en la consola.

Sin embargo, es posible sobrecargar el método **ToString** en la clase **coordinate** para que devuelva objetos de ese tipo que sean más representativos, como una cadena con los valores que contiene el objeto.

Método Equals

El método **Equals** determina si el objeto especificado es el mismo que el objeto actual. Como ya hemos visto, la forma predeterminada de **Equals** incluye únicamente la igualdad entre referencias.

Este método puede ser reemplazado por subclases para incluir la igualdad entre valores.

Método GetType

Este método permite extraer de un objeto información sobre su tipo en tiempo de ejecución. Se discutirá con más detalle en la sección sobre conversiones de datos de este mismo módulo.

Método Finalize

El sistema de tiempo de ejecución hace una llamada a este método cuando un objeto se hace inaccesible.

Reflexión

Objetivo del tema
Introducir el concepto de reflexión.

Explicación previa
En ocasiones es necesario obtener información sobre el tipo de objeto. Es lo que en C# se conoce como reflexión.

- Es posible obtener información sobre el tipo de un objeto
- Espacio de nombres System.Reflection
- El operador typeof devuelve el tipo de un objeto
  - Sólo para clases en tiempo de compilación
- Método GetType en System.Object
  - Información sobre clases en tiempo de ejecución

Recomendación al profesor
En otros lenguajes, la reflexión se conoce como *identificación de tipos en tiempo de ejecución* (RTTI).

Recomendación al profesor
Al contrario de lo que ocurre en C y C++, **typeof** sólo se puede aplicar a nombres de clases, no a variables ni a expresiones.

Es posible obtener información sobre el tipo de un objeto empleado un mecanismo llamado reflexión.

El mecanismo de reflexión en C# está controlado por el espacio de nombres **System.Reflection** en .NET Framework, que contiene clases e interfaces que proporcionan una vista de tipos, métodos y campos.

La clase **System.Type** incluye métodos que permiten obtener información sobre una declaración de tipo, como los constructores, métodos, campos, propiedades y eventos de una clase. Un objeto **Type** que representa un tipo es único; es decir, dos referencias de objeto **Type** sólo apuntan al mismo objeto si representan el mismo tipo. Esto hace posible comparar objetos **Type** utilizando comparaciones de referencias (los operadores == and **!=**).

El operador typeof

En tiempo de compilación se puede utilizar el operador **typeof** para obtener información de tipo para un nombre de tipo concreto.

Este ejemplo obtiene información en tiempo de compilación para el tipo byte y muestra el nombre del tipo en la consola.

```
using System;
using System.Reflection;
Type t = typeof(byte);
Console.WriteLine("Type: {0}", t);
```

El siguiente ejemplo muestra información más detallada sobre una clase. Más concretamente, muestra los métodos para esa clase.

```
using System;
using System.Reflection;
Type t = typeof(string); // Obtener información sobre tipo
InfoMetodos [ ] mi = t. GetMethods( );
foreach (MethodInfo m in mi)  {
    Console.WriteLine("Method: {0}", m);
}
```

Método GetType

El operador **typeof** sólo funciona con clases que existen en tiempo de compilación. Cuando se necesita información sobre tipo en tiempo de ejecución se puede emplear el método **GetType** de la clase **Object**.

Para más información sobre reflexión, busque “System.Reflection” en los documentos de ayuda del SDK de .NET Framework.

◆ Espacios de nombres en .NET Framework

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

Explicación previa
En esta lección veremos cómo C# utiliza .NET Framework.

- Espacio de nombres System.IO
- Espacio de nombres System.Xml
- Espacio de nombres System.Data
- Otros espacios de nombres útiles

El marco de trabajo .NET Framework incluye servicios de lenguaje común para distintas herramientas de desarrollo de aplicaciones. Las clases que contiene proporcionan una interfaz para el runtime de lenguaje común, el sistema operativo y la red.

.NET Framework es demasiado grande y ofrece demasiadas prestaciones como para poder incluir en este curso todas sus características. Si desea información más detallada, consulte los documentos de ayuda de Microsoft Visual Studio® .NET y del SDK de .NET Framework.

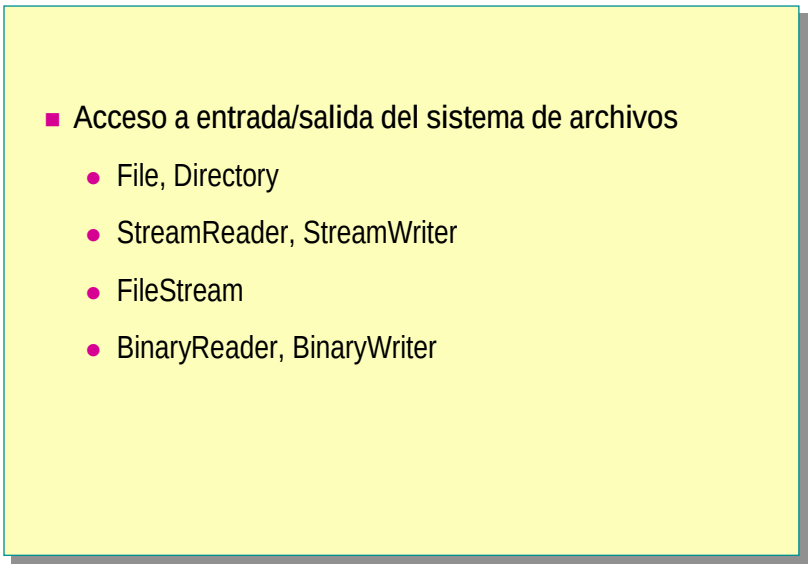
Al final de esta lección, usted será capaz de:

- Identificar espacios de nombres comunes en el marco de trabajo.
- Usar algunos de los espacios de nombres comunes en el marco de trabajo.

Espacio de nombres System.IO

Objetivo del tema
Describir brevemente la funcionalidad de System.IO.

Explicación previa
El espacio de nombres System.IO proporciona muchas clases útiles para trabajar con archivos y directorios.



El espacio de nombres **System.IO** es importante porque contiene muchas clases que permiten a una aplicación efectuar distintas operaciones de entrada y salida (I/O) en el sistema de archivos.

Incluye también clases para que una aplicación pueda realizar operaciones de entrada y salida sobre archivos y directorios.

El espacio de nombres **System.IO** es demasiado grande para que podamos explicarlo aquí con detalle. No obstante, la siguiente lista da una idea de las posibilidades:

- Las clases **File** y **Directory** permiten a una aplicación crear, borrar y manipular directorios y archivos.
- Las clases **StreamReader** y **StreamWriter** permiten a un programa acceder a los contenidos de un archivo como una secuencia de bytes o caracteres.
- La clase **FileStream** se puede emplear para dar acceso aleatorio a archivos.
- Las clases **BinaryReader** y **BinaryWriter** ofrecen una forma de leer y escribir tipos de datos primitivos como valores binarios.

Ejemplo de System.IO

A continuación se incluye un breve ejemplo en el que se muestra cómo abrir y leer un archivo como una secuencia. Este ejemplo no es indicativo de todas las maneras posibles de emplear el espacio de nombres **System.IO**, pero ilustra cómo se puede efectuar una sencilla operación de copia de archivos.

```
using System;
using System.IO; // Usar espacio de nombres IO
// ...
StreamReader reader = new StreamReader("archivoentrada.txt");
  // Entrada de texto desde archivo
StreamWriter writer = new StreamWriter("archivosalida.txt");
  // Salida de texto a archivo
string line;
while ((line = reader.ReadLine( )) != null)
{
    writer.WriteLine(line);
}

reader.Close( );
writer.Close( );
```

Para abrir un archivo para lectura, el código del ejemplo crea un nuevo objeto **StreamReader** y pasa el nombre del archivo que se desea abrir en el constructor. Del mismo modo, el ejemplo crea un nuevo objeto **StreamWriter** y pasa el nombre del archivo de salida en su constructor para abrir un archivo para escritura. En este ejemplo los nombres de los archivos están incluidos en el código, pero también podrían ser variables string.

El programa del ejemplo copia un archivo leyendo línea a línea de la secuencia de entrada y escribiendo esa línea en la secuencia de salida.

ReadLine y **WriteLine** nos pueden resultar familiares, ya que a clase **Console** incluye dos métodos estáticos del mismo nombre. En el ejemplo, estos métodos son métodos de instancia de las clases **StreamReader** y **StreamWriter**, respectivamente.

Para más información sobre el espacio de nombres **System.IO**, busque “espacio de nombres System.IO” en los documentos de ayuda del SDK de .NET Framework.

Espacio de nombres System.Xml

Objetivo del tema
Describir el espacio de nombres System.Xml.

Explicación previa
Las aplicaciones que tienen que interactuar con XML pueden usar el espacio de nombres System.Xml.

- Compatibilidad con XML
- Estándares de XML

Las aplicaciones que tienen que interactuar con XML (lenguaje de marcado extensible) pueden usar el espacio de nombres **System.Xml**, que proporciona la funcionalidad necesaria para procesar XML a partir de estándares del sector.

El espacio de nombres **System.Xml** es compatible con varios estándares de XML, incluyendo los siguientes:

- XML 1.0 con definición de tipo de documento (Document Type Definition, DTD)
- Espacios de nombres XML
- Esquemas XSD
- Expresiones XPath
- Transformaciones XSL/T
- DOM Level 1 core
- DOM Level 2 core

La clase **XmlDocument** se emplea para representar un documento XML completo. Los elementos y atributos en un documento XML se representan en las clases **XmlElement** y **XmlAttribute**.

Este curso no incluye una discusión detallada de los espacios de nombres para XML. Para más información, busque “espacio de nombres System.Xml” en los documentos de ayuda del SDK de .NET Framework.

Espacio de nombres System.Data

Objetivo del tema
Describir los espacios de nombres System.Data.SqlClient y System.Data.ADO.

Explicación previa
El espacio de nombres System.Data.SqlClient permite acceder a SQL Server. El espacio de nombres System.Data consta básicamente de las clases que constituyen la arquitectura ADO.NET.

- System.Data.SqlClient
  - SQL Server .NET Data Provider
- System.Data
  - Consta básicamente de las clases que constituyen la arquitectura ADO.NET

El espacio de nombres **System.Data** contiene clases que constituyen la arquitectura ADO.NET, que permite crear componentes que administran de forma eficaz datos procedentes de distintas fuentes. ADO.NET proporciona las herramientas necesarias para solicitar, actualizar y reconciliar datos en sistemas de *n* niveles.

Dentro de ADO.NET se puede usar la clase **DataSet**. En cada **DataSet** hay objetos **DataTable**, y cada **DataTable** contiene a su vez datos procedentes de una única fuente, como Microsoft SQL Server™.

El espacio de nombres **System.Data.SqlClient** proporciona acceso directo a SQL Server. Este espacio de nombres es específico para SQL Server.

Para acceder a otras bases de datos relacionales y fuentes de datos estructurados se puede utilizar el espacio de nombres **System.Data.OleDb**, que proporciona acceso de alto nivel a los controladores de bases de datos OLEDB.

Este curso no incluye una discusión detallada de los espacios de nombres **System**. Para más información, busque “espacio de nombres System.Data” en los documentos de ayuda del SDK de .NET Framework.

Otros espacios de nombres útiles

Objetivo del tema
Ofrecer un breve resumen de otros componentes útiles de .NET Framework.

Explicación previa
.NET Framework contiene muchos otros componentes útiles.

- Espacio de nombres System
- Espacio de nombres System.Net
- Espacio de nombres System.Net.Sockets
- Espacio de nombres System.Windows.Forms

Recomendación al profesor
No intente explicar el uso de estos componentes del marco de trabajo. Limítese a mencionar que existen y siga adelante.

Hay muchos más espacios de nombres y clases útiles en .NET Framework. Aunque en este curso no los vamos a tratar con detalle, la siguiente información puede resultar útil para buscar la documentación y los archivos de referencia:

- El espacio de nombres **System** contiene clases que definen tipos de datos valor y referencia, eventos e identificadores de eventos, interfaces, atributos y excepciones de procesamiento que se utilizan con frecuencia. Otras clases proporcionan servicios para conversión de tipos de datos, manipulación de parámetros de métodos, matemáticas, invocación remota y local a programas y administración de aplicaciones.
- El espacio de nombres **System.Net** proporciona una sencilla interfaz de programación para muchos de los protocolos que se utilizan hoy en día en la red. El espacio de nombres **System.Net.Sockets** contiene una implementación de la interfaz Microsoft Windows® Sockets para desarrolladores que requieren acceso de bajo nivel a sistemas de redes TCP/IP (protocolo de control de transporte/protocolo Internet).

Para su información
System.WAO se llamaba System.WinForms en las primeras versiones del SDK de .NET Framework.

- **System.Windows.Forms** es el marco de trabajo de interfaz gráfica de usuario (GUI) para aplicaciones Windows y permite utilizar formularios, controles e identificadores de eventos.

Para más información sobre espacios de nombres **System**, busque “espacio de nombres System” en los documentos de ayuda del SDK de .NET Framework.

◆ Conversiones de datos

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

Explicación previa
Esta sección explica cómo realizar conversiones de datos entre tipos referencia en C#.

- Conversión de tipos valor
- Conversiones padre/hija
- El operador is
- El operador as
- Conversiones y el tipo object
- Conversiones e interfaces
- Boxing y unboxing

Esta sección explica cómo realizar conversiones de datos entre tipos referencia en C#. Es posible convertir referencias de un tipo a otro, pero debe existir una relación entre los tipos referencia.

Al final de esta lección, usted será capaz de:

- Identificar conversiones permitidas y prohibidas entre tipos referencia.
- Usar mecanismos de conversión (casts, **is** y **as**).
- Identificar aspectos especiales a tener en cuenta para la conversión a y desde el tipo **object**.
- Usar el mecanismo de reflexión, que permite examinar información sobre tipos en tiempo de ejecución.
- Realizar conversiones automáticas (boxing y unboxing) entre tipos valor y tipos referencia.

Conversión de tipos valor

Objetivo del tema
Repasar los mecanismos de conversión de tipo para tipos valor.

Explicación previa
Este tema es un repaso de lo ya aprendido sobre conversiones de tipo para tipos valor.

- Conversiones implícitas
- Conversiones explícitas
  - Operador de cast
- Excepciones
- Clase System.Convert
  - Control interno de conversiones

C# permite las conversiones de datos implícitas y explícitas.

Conversiones implícitas

Para tipos valor hemos aprendido dos formas de convertir datos: conversión implícita y conversión explícita utilizando el operador de cast (molde).

Recomendación al profesor
Este tema es fundamentalmente un repaso de la conversión de datos tal como se explicó en módulos anteriores. La única novedad se refiere a la clase System.Convert.

Se dice que una conversión es implícita cuando un valor de un tipo se asigna a otro tipo. C# únicamente permite la conversión implícita para ciertas combinaciones de tipos, normalmente cuando el primer valor se puede convertir al segundo sin pérdida de datos. El siguiente ejemplo muestra una conversión implícita de datos de **int** a **long**:

```
int a = 4;
long b;
b = a; // Conversión implícita de int a long
```

Conversiones explícitas

Los tipos valor se pueden convertir explícitamente usando el operador de cast, como en este ejemplo:

```
int a;
long b = 7;
a = (int) b;
```

Excepciones

Al utilizar el operador de cast hay que tener en cuenta que pueden surgir problemas si la variable de destino no puede contener el valor. Si se detecta un problema durante una conversión explícita (como tratar de encajar el valor 9.999.999.999 en una variable **int**), C# puede lanzar una excepción (en este caso, **OverflowException**). Si se desea, se pueden utilizar bloques **try-catch** para capturar la excepción:

```
try {
    a = checked((int) b);
}
catch (Exception) {
    Console.WriteLine("Problema en cast");
}
```

En operaciones con enteros hay que emplear la palabra reservada **checked** o bien compilar utilizando la configuración adecuada del compilador, ya que de lo contrario no se efectuará la comprobación.

Clase System.Convert

Las conversiones entre los distintos tipos base (como int, long y bool) se controlan mediante la clase **System.Convert** de .NET Framework. Normalmente no es necesario hacer llamadas a métodos de **System.Convert**, ya que el compilador realiza esas llamadas automáticamente.

Conversiones padre/hija

Objetivo del tema
Explicar las reglas para conversión de tipo entre referencias para clases padres e hijas.

Explicación previa
Bajo ciertas circunstancias, es posible convertir una referencia a un objeto de una clase hija a un objeto de su clase padre, y viceversa.

- Conversión a referencia de clase padre
  - Implícita o explícita
  - Siempre tiene éxito
  - Siempre se puede asignar a un objeto
- Conversión a referencia de clase hija
  - Es necesario cast explícito
  - Comprueba que la referencia es del tipo correcto
  - Si no lo es, causa una excepción InvalidCastException

Bajo ciertas circunstancias, es posible convertir una referencia a un objeto de una clase hija a un objeto de su clase padre, y viceversa.

Conversión a referencia de clase padre

Las referencias a objetos de un tipo de clase se pueden convertir en referencias a otro tipo si una clase hereda directa o indirectamente de la otra.

Siempre es posible convertir una referencia a un objeto en una referencia a un objeto de una clase padre. Esta conversión se puede realizar de forma implícita (por asignación o como parte de una expresión) o explícita (utilizando el operador de cast).

En los siguientes ejemplos se usarán dos clases: **Animal** y **Pájaro**. **Animal** es la clase padre de **Pájaro** o, por decirlo de otro modo, **Pájaro** hereda de **Animal**.

En este ejemplo se declara una variable de tipo **Animal** y otra de tipo **Pájaro**:

```
Animal a;
Pajaro b = new Pajaro(...);
```

Supongamos ahora la siguiente asignación, en la que la referencia en *b* se copia a *a*:

```
a = b;
```

La clase **Pájaro** hereda de la clase **Animal** y, por lo tanto, un método que esté en **Animal** estará también en **Pájaro** (es posible que la clase **Pájaro** haya sustituido algunos métodos de **Animal** para crear su propia versión de ellos, pero en cualquier caso el método estará presente en alguna forma). Como

consecuencia, es posible asignar referencias a objetos **Pájaro** a variables que contengan referencias a objetos de tipo **Animal**.

En este caso, C# efectúa una conversión de tipo de **Pájaro** a **Animal**. Para hacer una conversión explícita de **Pájaro** a **Animal** se puede usar un operador de cast:

```
a = (Animal) b;
```

Este código dará exactamente el mismo resultado.

Conversión a referencia de clase hija

También es posible convertir una referencia a un tipo de clase hija, pero es necesario hacerlo de forma explícita con un operador de cast. Una conversión explícita se somete a una comprobación en tiempo de ejecución para garantizar que los tipos son compatibles, como se ve en el siguiente ejemplo:

```
Pajaro b = (Pajaro) a; // Okay
```

Este código se compilará sin problemas. El operador de cast efectúa una comprobación en tiempo de ejecución para determinar si el objeto al que se apunta está realmente en la clase **Pájaro**, y si no lo está lanza la excepción **InvalidCastException**.

Si se intenta hacer una asignación a un tipo de clase hija sin emplear un operador de conversión, como en el siguiente código, el compilador mostrará el mensaje de error “Imposible convertir tipo ‘Animal’ a ‘Pájaro’ de forma implícita”.

```
b = a; // No se compila
```

Como con cualquier otra excepción, se puede utilizar un bloque **try-catch** para capturar un error de conversión de tipo, como en el siguiente código:

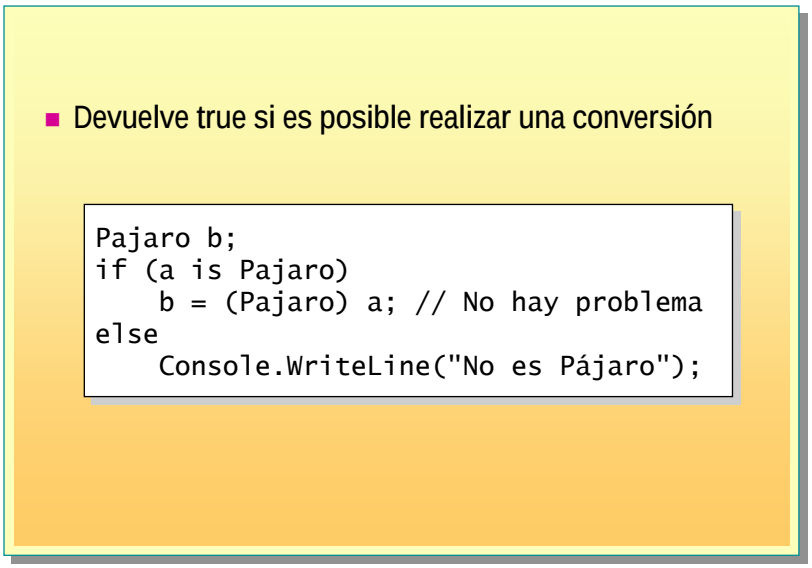
```
try {
    b = (Pajaro) a;
}
catch (InvalidCastException) {
    Console.WriteLine("No es Pájaro");
}
```

Recomendación al profesor
Recuerde que todos los Pájaros son Animales, pero no todos los Animales son Pájaros. Por esta razón es necesario que el sistema de tiempo de ejecución compruebe el operador de cast.

El operador is

Objetivo del tema
Introducir y explicar el operador **is**.

Explicación previa
El operador **is** permite determinar si una referencia a un objeto se puede convertir en una referencia a una clase concreta.



Es posible controlar los tipos incompatibles capturando la excepción **InvalidCastException**, pero hay otras formas de hacer frente a este problema, como el operador **is**.

El operador **is** se puede utilizar para probar el tipo del objeto sin efectuar una conversión. Este operador devuelve **true** si el valor de la izquierda no es **null** y es posible realizar un cast a la clase de la derecha sin que se lance una excepción; en caso contrario, devuelve **false**.

```
if (a is Pajaro)
    b = (Pajaro) a; // No hay problema, ya que "a es Pajaro "
devuelve true
else
    Console.WriteLine("No es Pájaro");
```

Podemos pensar en las relaciones entre clases heredadas como si indicaran una relación “es un tipo de”, como en “Un pájaro es un tipo de animal”. Las referencias en la variable *a* deben ser referencias a objetos **Animal**, y *b* es un tipo de animal. Por supuesto, *b* es también un pájaro, pero un pájaro no es más que un caso especial de un animal. Lo contrario no es cierto: Un animal no es un tipo de pájaro; algunos animales son pájaros, pero no es cierto (true) que todos los animales sean pájaros.

Recomendación al profesor
Puede ser conveniente discutir el significado de “*a* es Pájaro” si *a* es una instancia de una nueva clase **Loro**, que hereda de **Pájaro**.

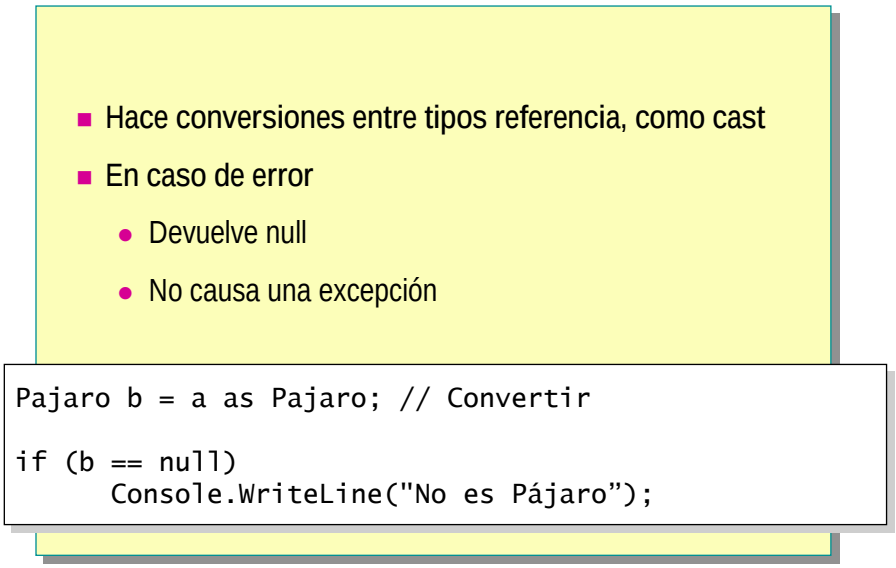
Así, la siguiente expresión se puede leer como “Si *a* es un tipo de pájaro” o “Si *a* es un pájaro o un tipo derivado de pájaro”:

```
if (a is Pajaro)
```

El operador as

Objetivo del tema
Introducir y explicar el operador as.

Explicación previa
El operador as permite hacer conversiones entre tipos referencia sin causar una excepción.



Se puede utilizar el operador **as** para realizar conversiones entre tipos.

Ejemplo

La siguiente instrucción realiza una conversión de la referencia en *a* a un valor que apunta a una clase de tipo **Pájaro**, y el runtime comprueba automáticamente que la conversión está permitida.

```
b = a as Pajaro;
```

Tratamiento de errores

El operador **as** se diferencia del operador de cast por su forma de controlar los errores. Si en el ejemplo anterior no fuera posible convertir la referencia en la variable *a* en una referencia a un objeto de clase **Pájaro**, se almacenaría en *b* el valor **null** y el programa continuaría. El operador **as** nunca causa una excepción.

El código anterior se puede rescribir de la siguiente manera para que muestre un mensaje de error si no se puede hacer la conversión:

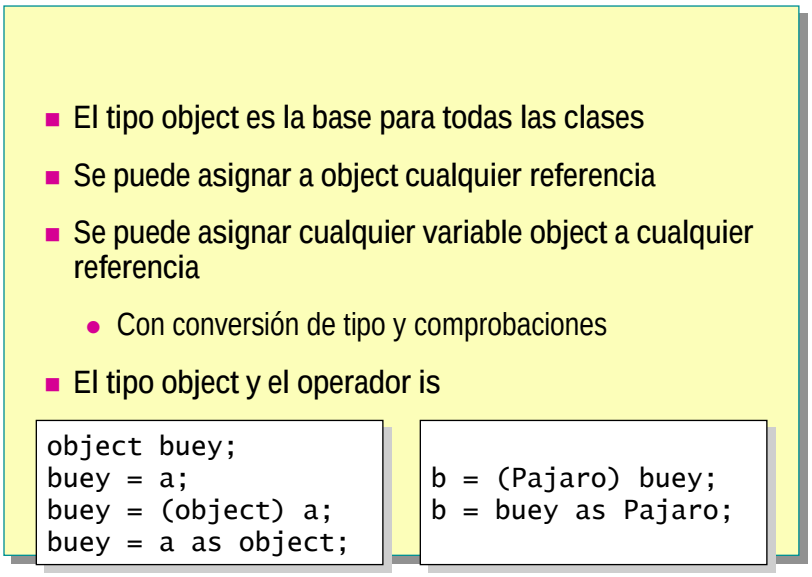
```
Pajaro b = a as Pajaro;
if (b == null)
    Console.WriteLine("No es Pájaro");
```

Aunque **as** nunca causa una excepción, cualquier intento de acceso por el valor convertido producirá una excepción **NullReferenceException** si ese valor es **null**. Por lo tanto hay que comprobar siempre el valor devuelto por **as**.

Conversiones y el tipo object

Objetivo del tema
Explicar cómo convertir el tipo **object** a y desde otros tipos referencia.

Explicación previa
Todos los tipos referencia se basan en el tipo **object**. Esto significa que es posible almacenar cualquier referencia en una variable de tipo **object**.



Todos los tipos referencia se basan en el tipo **object**. Esto significa que es posible almacenar cualquier referencia en una variable de tipo **object**.

El tipo object es la base para todas las clases

El tipo **object** es la base para todos los tipos referencia.

Se puede asignar a object cualquier referencia

Puesto que todas las clases se basan directa o indirectamente en el tipo **object**, es posible asignar cualquier referencia a una variable de tipo **object** usando una conversión implícita o un operador de cast. El código siguiente muestra un ejemplo:

Recomendación al profesor
Puede comparar con Variant en Visual Basic, pero el mecanismo no es igual. Un Variant de Visual Basic es una estructura con un designador de tipo y un valor, y no se basa en referencias.

En Visual Basic, Object representa una interfaz IDispatch COM "late-bound". Asegúrese de que los programadores de Visual Basic comprenden que **object** en C# no tiene relación con COM.

```
object buey;
buey = a;
buey = (object) a;
buey = a as object;
```

Se puede asignar cualquier variable object a cualquier referencia

Es posible asignar un valor de tipo **object** a la referencia de cualquier otro objeto, siempre y cuando se aplique correctamente el operador cast. Hay que recordar que el sistema de tiempo de ejecución comprobará que el valor asignado es del tipo correcto. El código siguiente muestra un ejemplo:

```
b = (Pajaro) buey;
b = buey as Pajaro;
```

Los ejemplos anteriores se pueden escribir con comprobación completa de errores de la siguiente manera:

```
try {
    b = (Pajaro) buey;
}
catch (InvalidCastException) {
    Console.WriteLine("No es posible convertir a Pájaro");
}
b = buey as Pajaro;
if (b == null)
    Console.WriteLine("No es posible convertir a Pájaro");
```

El tipo object y el operador is

Puesto que en último término todos los valores derivan de **object**, el operador **is** siempre devolverá **true** cuando compruebe si un valor es **object**.

```
if (a is object) // Siempre devuelve true
```

Conversiones e interfaces

Objetivo del tema
Explicar las conversiones de tipos con interfaces.

Explicación previa
Los operadores as, is y de cast también se pueden aplicar a interfaces.

- Una interfaz sólo se puede usar para acceder a sus propios miembros
- No es posible acceder a otros miembros y variables de la clase a través de la interfaz

Cuando se trabaja con interfaces es posible utilizar los operadores **as**, **is** y de cast para hacer conversiones.

Por ejemplo, se puede declarar una variable de un tipo interfaz, como:

```
IHashCodeProvider hcp;
```

Conversión de una referencia a una interfaz

El operador de cast se puede usar para convertir la referencia al objeto en una referencia a una interfaz determinada:

```
IHashCodeProvider hcp;
hcp = (IHashCodeProvider) x;
```

Como ocurre en la conversión entre referencias de clase, el operador de cast producirá una excepción **InvalidCastException** si el objeto indicado no implementa la interfaz. Antes de aplicar el cast a un objeto hay que determinar si ese objeto es compatible con una interfaz, o bien usar un bloque **try-catch** para capturar la excepción.

Implementación de una interfaz

Se puede usar el operador **is** para determinar si un objeto es compatible con una interfaz. La sintaxis es la misma que la que se utiliza para clases:

```
if (x is IHashCodeProvider) ...
```

Recomendación al profesor
Podría ser aconsejable leer esto como "si x implementa IHashCodeProvider".

Uso del operador as

También se puede utilizar el operador **as** como alternativa a cast:

```
IHashCodeProvider hcp;
hcp = x as IHashCodeProvider;
```

Como ocurre en la conversión entre clases, el operador **as** devuelve **null** si la referencia convertida no es compatible con la interfaz.

Una vez convertida una referencia a una clase en una referencia a una interfaz, la nueva referencia puede acceder únicamente a miembros de esa interfaz, y no a los otros miembros públicos de la clase.

Ejemplo

Consideremos el siguiente ejemplo para cómo funciona la conversión de referencias en interfaces. Supongamos que hemos creado una interfaz llamada **IVisual** que especifica un método llamado **Paint**:

```
interface IVisual
{
    void Paint( );
}
```

Supongamos que tenemos también una clase **Rectangle** que implementa la interfaz **IVisual**. Implementa por tanto el método **Paint**, aunque también puede definir sus propios métodos. En este ejemplo, **Rectangle** ha definido otro método llamado **Move** que no forma parte de **IVisual**.

Recomendación al profesor
No se muestra la sintaxis para herencia porque se discutirá en un módulo posterior.

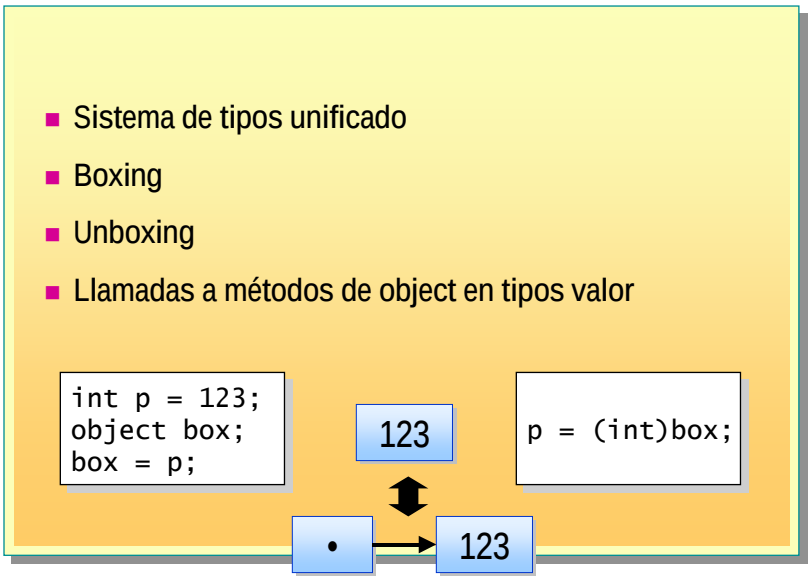
Como era de esperar, es posible crear un **Rectangle**, *r*, y usar sus métodos **Move** y **Paint**. Incluso podemos referenciarlo a través de una variable **IVisual**, *v*. Sin embargo, a pesar del hecho de que tanto *v* como *r* apuntan al mismo objeto en la memoria, no es posible usar *v* para llamar al método **Move** porque no forma parte de la interfaz **IVisual**. El siguiente código ofrece algunos ejemplos:

```
Rectangle r = new Rectangle( );
r.Move( );       // Okay
r.Paint( );  // Okay
IVisual v = (IVisual) r;
v.Move( );       // Not valid
v. Paint( );     // Okay
```

Boxing y unboxing

Objetivo del tema
Explicar los conceptos de boxing y unboxing.

Explicación previa
C# puede convertir automáticamente tipos valor en referencias a objetos y referencias a objetos en tipos valor.



C# puede convertir tipos valor en referencias a objetos y referencias a objetos en tipos valor.

Sistema de tipos unificado

C# tiene un sistema unificado de tipos que permite convertir tipos valor en referencias de tipo **object** y referencias a objetos en tipos valor. Es posible convertir tipos valor en referencias de tipo **object** y viceversa.

De esta forma, los valores de tipos como **int** y **bool** se pueden tratar como valores simples en casi todos los casos. Normalmente ésta es la técnica más eficaz, ya que se evitan las complicaciones causadas por las referencias. Sin embargo, también hay un sistema que permite poner estos valores temporalmente en una "caja" (*box*) y utilizarlos como si fueran referencias cuando sea necesario.

Boxing

Las expresiones de tipos valor también se pueden convertir en valores de tipo **object**, y viceversa. Si se quiere convertir una variable de tipo valor en un tipo **object**, se asigna un objeto *box* para contener el valor y se copia éste en la caja. Este proceso se conoce con el nombre de *boxing*.

```
int p = 123;
object box;
box = p;          // Boxing (implicit)
box = (object) p; // Boxing (explicit)
```

La operación de boxing puede ser implícita o explícita (con un cast a un objeto). El boxing se usa especialmente cuando se pasa un tipo valor a un parámetro de tipo **object**.

Unboxing

Cuando un valor en un objeto se vuelve a convertir en un tipo valor, el valor se saca de la caja y se copia en la ubicación correspondiente. Este proceso se conoce con el nombre de *unboxing*.

```
p = (int) box;     // Unboxing
```

La operación de unboxing se tiene que hacer con un operador de cast explícito.

Si el valor en la referencia no es exactamente del mismo tipo que el cast, éste produce una excepción **InvalidCastException**.

Llamadas a métodos de object en tipos valor

El hecho de que el boxing pueda ser implícito permite hacer llamadas a métodos del tipo object type en cualquier variable o expresión, aunque tengan tipos valor. El código siguiente muestra un ejemplo:

```
static void Show(object o)
{
    Console.WriteLine(o.ToString( ));
}
Show(42);
```

Este código funciona porque de forma implícita se está aplicando boxing al valor 42 para convertirlo en un parámetro **object**, y a continuación se hace una llamada al método **ToString** de este parámetro.

El resultado es el mismo que con el código siguiente:

```
object o = (object) 42; // Boxing
Console.WriteLine(o.ToString( ));
```

Nota Cuando se llama a métodos de **Object** *directamente* en un valor *no* se produce boxing. Por ejemplo, la expresión `42.ToString( )` no aplica boxing a 42 para convertirlo en un **object**. Esto se debe a que el compilador puede determinar estáticamente el tipo y decidir qué método llamar.