# **Tipos De Datos**

## **Tipos, variables y valores**

C# es un lenguaje fuertemente tipado. Todas las variables y constantes tienen un tipo, al igual que todas las expresiones que se evalúan como un valor. Cada declaración del método especifica un nombre, un número de parámetros, un tipo y una naturaleza (valor, referencia o salida) para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET define un conjunto de tipos numéricos integrados, así como tipos más complejos que representan una amplia variedad de construcciones lógicas, como el sistema de archivos, conexiones de red, colecciones y matrices de objetos, y fechas. Los programas de C# típicos usan tipos de la biblioteca de clases, así como tipos definidos por el usuario que modelan los conceptos que son específicos del dominio del problema del programa.

Entre la información almacenada en un tipo se puede incluir lo siguiente:

* El espacio de almacenamiento que requiere una variable del tipo.
* Los valores máximo y mínimo que puede representar.
* Los miembros (métodos, campos, eventos, etc.) que contiene.
* El tipo base del que hereda.
* La ubicación donde se asignará la memoria para variables en tiempo de ejecución.
* Los tipos de operaciones permitidas.

El compilador usa información de tipo para garantizar que todas las operaciones que se realizan en el código cuentan *con seguridad de tipos*. Por ejemplo, si declara una variable de tipo [int](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types), el compilador le permite usar la variable en operaciones de suma y resta. Si intenta realizar esas mismas operaciones en una variable de tipo [bool](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/bool), el compilador genera un error, como se muestra en el siguiente ejemplo:

C#

int a = 5;

int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.

int c = a + test;

Nota

Los desarrolladores de C y C++ deben tener en cuenta que, en C#, [bool](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/bool) no se puede convertir en [int](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types).

El compilador inserta la información de tipo en el archivo ejecutable como metadatos. Common Language Runtime (CLR) usa esos metadatos en tiempo de ejecución para garantizar aún más la seguridad de tipos cuando asigna y reclama memoria.

### **Especificar tipos en declaraciones de variable**

Cuando declare una variable o constante en un programa, debe especificar su tipo o usar la palabra clave [var](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/var) para que el compilador infiera el tipo. En el ejemplo siguiente se muestran algunas declaraciones de variable que utilizan tanto tipos numéricos integrados como tipos complejos definidos por el usuario:

C#

// Declaration only:

float temperature;

string name;

MyClass myClass;

// Declaration with initializers (four examples):

char firstLetter = 'C';

var limit = 3;

int[] source = { 0, 1, 2, 3, 4, 5 };

var query = from item in source

where item <= limit

select item;

Los tipos de parámetros del método y los valores devueltos se especifican en la declaración del método. En la siguiente firma se muestra un método que requiere una variable [int](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) como argumento de entrada y devuelve una cadena:

C#

public string GetName(int ID)

{

if (ID < names.Length)

return names[ID];

else

return String.Empty;

}

private string[] names = { "Spencer", "Sally", "Doug" };

Tras declarar una variable, no se puede volver a declarar con un nuevo tipo y no se le puede asignar un valor que no sea compatible con su tipo declarado. Por ejemplo, no puede declarar un valor [int](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) y, luego, asignarle un valor booleano de true. En cambio, los valores se pueden convertir en otros tipos, por ejemplo, cuando se asignan a variables nuevas o se pasan como argumentos de método. El compilador realiza automáticamente una *conversión de tipo* que no da lugar a una pérdida de datos. Una conversión que pueda dar lugar a la pérdida de datos requiere un valor *cast* en el código fuente.

Para obtener más información, vea [Conversiones de tipos](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/types/casting-and-type-conversions).

## **Tipos integrados**

C# proporciona un conjunto estándar de tipos numéricos integrados para representar números enteros, valores de punto flotante, expresiones booleanas, caracteres de texto, valores decimales y otros tipos de datos. También hay tipos string y object integrados. Están disponibles para su uso en cualquier programa de C#. Para obtener una lista completa de los tipos integrados, vea [Tipos integrados](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/built-in-types).

## **Tipos personalizados**

Las construcciones [struct](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/struct), [class](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/class), [interface](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/interface) y [enum](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/enum) se utilizan para crear sus propios tipos personalizados. La biblioteca de clases .NET es en sí misma una colección de tipos personalizados proporcionados por Microsoft que puede usar en sus propias aplicaciones. De forma predeterminada, los tipos usados con más frecuencia en la biblioteca de clases están disponibles en cualquier programa de C#. Otros están disponibles solo cuando agrega explícitamente una referencia de proyecto al ensamblado en el que se definen. Una vez que el compilador tenga una referencia al ensamblado, puede declarar variables (y constantes) de los tipos declarados en dicho ensamblado en el código fuente. Para más información, vea [Biblioteca de clases .NET](https://docs.microsoft.com/es-mx/dotnet/standard/class-library-overview).

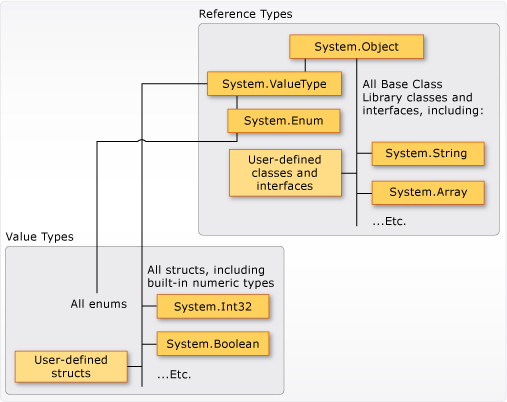
## **Common Type System**

Es importante entender dos aspectos fundamentales sobre el sistema de tipos en .NET:

* Es compatible con el principio de herencia. Los tipos pueden derivarse de otros tipos, denominados *tipos base*. El tipo derivado hereda (con algunas restricciones), los métodos, las propiedades y otros miembros del tipo base. A su vez, el tipo base puede derivarse de algún otro tipo, en cuyo caso el tipo derivado hereda los miembros de ambos tipos base en su jerarquía de herencia. Todos los tipos, incluidos los tipos numéricos integrados como [System.Int32](https://docs.microsoft.com/es-es/dotnet/api/system.int32) (palabra clave de C#: [int](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types)), derivan en última instancia de un único tipo base, que es [System.Object](https://docs.microsoft.com/es-es/dotnet/api/system.object) (palabra clave de C#: [object](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/reference-types)). Esta jerarquía de tipos unificada se denomina [Common Type System](https://docs.microsoft.com/es-mx/dotnet/standard/base-types/common-type-system) (CTS). Para más información sobre la herencia en C#, vea [Herencia](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/classes-and-structs/inheritance).
* En CTS, cada tipo se define como un *tipo de valor* o un *tipo de referencia*. Esto incluye todos los tipos personalizados de la biblioteca de clases .NET y también sus propios tipos definidos por el usuario. Los tipos que se definen mediante el uso de la palabra clave [struct](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/struct) son tipos de valor; todos los tipos numéricos integrados son structs. Los tipos que se definen mediante el uso de la palabra clave [class](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/class) son tipos de referencia. Los tipos de referencia y los tipos de valor tienen distintas reglas de tiempo de compilación y distintos comportamientos de tiempo de ejecución.

En la ilustración siguiente se muestra la relación entre los tipos de valor y los tipos de referencia en CTS.

En la imagen siguiente se muestran tipos de valores y tipos de referencias en CTS:



Nota

Puede ver que los tipos utilizados con mayor frecuencia están organizados en el espacio de nombres [System](https://docs.microsoft.com/es-es/dotnet/api/system). Sin embargo, el espacio de nombres que contiene un tipo no tiene ninguna relación con un tipo de valor o un tipo de referencia.

### **Tipos de valor**

Los tipos de valor derivan de [System.ValueType](https://docs.microsoft.com/es-es/dotnet/api/system.valuetype), el cual deriva de [System.Object](https://docs.microsoft.com/es-es/dotnet/api/system.object). Los tipos que derivan de [System.ValueType](https://docs.microsoft.com/es-es/dotnet/api/system.valuetype) tienen un comportamiento especial en CLR. Las variables de tipo de valor contienen directamente sus valores, lo que significa que la memoria se asigna insertada en cualquier contexto en el que se declare la variable. No se produce ninguna asignación del montón independiente ni sobrecarga de la recolección de elementos no utilizados para las variables de tipo de valor.

Existen dos categorías de tipos de valor: [struct](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/struct) y [enum](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/enum).

Los tipos numéricos integrados son structs y tienen propiedades y métodos a los que se puede obtener acceso:

C#

// Static method on type byte.

byte b = byte.MaxValue;

Pero se declaran y se les asignan valores como si fueran tipos simples no agregados:

C#

byte num = 0xA;

int i = 5;

char c = 'Z';

Los tipos de valor están *sellados*, lo que significa que, por ejemplo, no puede derivar un tipo de [System.Int32](https://docs.microsoft.com/es-es/dotnet/api/system.int32), y no puede definir un struct para que herede de cualquier clase o struct definido por el usuario, porque un struct solo puede heredar de [System.ValueType](https://docs.microsoft.com/es-es/dotnet/api/system.valuetype). A pesar de ello, un struct puede implementar una o más interfaces. Puede convertir un tipo struct en cualquier tipo de interfaz que implemente. Esto hace que una operación de conversión *boxing* encapsule el struct dentro de un objeto de tipo de referencia en el montón administrado. Las operaciones de conversión boxing se producen cuando se pasa un tipo de valor a un método que toma [System.Object](https://docs.microsoft.com/es-es/dotnet/api/system.object) o cualquier tipo de interfaz como parámetro de entrada. Para obtener más información, vea [Conversión boxing y unboxing](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/types/boxing-and-unboxing).

Puede usar la palabra clave [struct](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/struct) para crear sus propios tipos de valor personalizados. Normalmente, un struct se usa como un contenedor para un pequeño conjunto de variables relacionadas, como se muestra en el ejemplo siguiente:

C#

public struct Coords

{

public int x, y;

public Coords(int p1, int p2)

{

x = p1;

y = p2;

}

}

Para más información sobre estructuras, vea [Tipos de estructura](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/struct). Para más información sobre los tipos de valor, vea [Tipos de valor](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/value-types).

La otra categoría de tipos de valor es [enum](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/enum). Una enumeración define un conjunto de constantes integrales con nombre. Por ejemplo, la enumeración [System.IO.FileMode](https://docs.microsoft.com/es-es/dotnet/api/system.io.filemode) de la biblioteca de clases .NET contiene un conjunto de enteros constantes con nombre que especifican cómo se debe abrir un archivo. Se define como se muestra en el ejemplo siguiente:

C#

public enum FileMode

{

CreateNew = 1,

Create = 2,

Open = 3,

OpenOrCreate = 4,

Truncate = 5,

Append = 6,

}

La constante System.IO.FileMode.Create tiene un valor de 2. Sin embargo, el nombre es mucho más significativo para los humanos que leen el código fuente y, por esa razón, es mejor utilizar enumeraciones en lugar de números literales constantes. Para obtener más información, vea [System.IO.FileMode](https://docs.microsoft.com/es-es/dotnet/api/system.io.filemode).

Todas las enumeraciones se heredan de [System.Enum](https://docs.microsoft.com/es-es/dotnet/api/system.enum), el cual se hereda de [System.ValueType](https://docs.microsoft.com/es-es/dotnet/api/system.valuetype). Todas las reglas que se aplican a las estructuras también se aplican a las enumeraciones. Para más información sobre las enumeraciones, vea [Tipos de enumeración](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/enum).

### **Tipos de referencia**

Un tipo que se define como [clase](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/class), [delegado](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/reference-types), matriz o [interfaz](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/interface) es un *tipo de referencia*. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor [null](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/null) hasta que se cree explícitamente un objeto mediante el operador [new](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/operators/new-operator), o bien que se le asigne un objeto creado en otro lugar mediante new, tal y como se muestra en el ejemplo siguiente:

C#

MyClass mc = new MyClass();

MyClass mc2 = mc;

Una interfaz debe inicializarse junto con un objeto de clase que lo implementa. Si MyClass implementa IMyInterface, cree una instancia de IMyInterface, tal como se muestra en el ejemplo siguiente:

C#

IMyInterface iface = new MyClass();

Cuando se crea el objeto, se asigna la memoria en el montón administrado y la variable solo contiene una referencia a la ubicación del objeto. Los tipos del montón administrado producen sobrecarga cuando se asignan y cuando los reclama la función de administración de memoria automática de CLR, conocida como *recolección de elementos no utilizados*. En cambio, la recolección de elementos no utilizados también está muy optimizada y no crea problemas de rendimiento en la mayoría de los escenarios. Para obtener más información sobre la recolección de elementos no utilizados, vea [Administración de memoria automática](https://docs.microsoft.com/es-mx/dotnet/standard/automatic-memory-management).

Todas las matrices son tipos de referencia, incluso si sus elementos son tipos de valor. Las matrices derivan de manera implícita de la clase [System.Array](https://docs.microsoft.com/es-es/dotnet/api/system.array), pero el usuario las declara y las usa con la sintaxis simplificada que proporciona C#, como se muestra en el ejemplo siguiente:

C#

// Declare and initialize an array of integers.

int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.

int len = nums.Length;

Los tipos de referencia admiten la herencia completamente. Al crear una clase, puede heredar de cualquier otra interfaz o clase que no esté definida como [sealed](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/sealed); y otras clases pueden heredar de la clase e invalidar sus métodos virtuales. Para más información sobre cómo crear clases propias, vea [Clases y structs](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/classes-and-structs/). Para más información sobre la herencia y los métodos virtuales, vea [Herencia](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/classes-and-structs/inheritance).

## **Tipos de valores literales**

En C#, los valores literales reciben un tipo del compilador. Puede especificar cómo debe escribirse un literal numérico; para ello, anexe una letra al final del número. Por ejemplo, para especificar que el valor 4.56 debe tratarse como un valor flotante, anexe "f" o "F" después del número: 4.56f. Si no se anexa ninguna letra, el compilador inferirá un tipo para el literal. Para obtener más información sobre los tipos que se pueden especificar con sufijos de letras, vea [Tipos numéricos integrales](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) y [Tipos numéricos de punto flotante](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types).

Dado que los literales tienen tipo y todos los tipos derivan en última instancia de [System.Object](https://docs.microsoft.com/es-es/dotnet/api/system.object), puede escribir y compilar código como el siguiente:

C#

string s = "The answer is " + 5.ToString();

// Outputs: "The answer is 5"

Console.WriteLine(s);

Type type = 12345.GetType();

// Outputs: "System.Int32"

Console.WriteLine(type);

## **Tipos genéricos**

Los tipos se pueden declarar con uno o varios *parámetros de tipo* que actúan como un marcador de posición para el tipo real (el *tipo concreto*) que proporcionará el código de cliente cuando cree una instancia del tipo. Estos tipos se denominan *tipos genéricos*. Por ejemplo, el tipo de .NET [System.Collections.Generic.List<T>](https://docs.microsoft.com/es-es/dotnet/api/system.collections.generic.list-1) tiene un parámetro de tipo al que, por convención, se le denomina *T*. Cuando crea una instancia del tipo, especifica el tipo de los objetos que contendrá la lista, por ejemplo, la cadena:

C#

List<string> stringList = new List<string>();

stringList.Add("String example");

// compile time error adding a type other than a string:

stringList.Add(4);

El uso del parámetro de tipo permite reutilizar la misma clase para incluir cualquier tipo de elemento, sin necesidad de convertir cada elemento en [object](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/reference-types). Las clases de colección genéricas se denominan *colecciones fuertemente tipadas* porque el compilador conoce el tipo específico de los elementos de la colección y puede generar un error en tiempo de compilación si, por ejemplo, intenta agregar un valor entero al objeto stringList del ejemplo anterior. Para más información, vea [Genéricos](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/generics/).

## **Tipos implícitos, tipos anónimos y tipos que admiten un valor NULL**

Como se ha mencionado anteriormente, puede asignar implícitamente un tipo a una variable local (pero no miembros de clase) mediante la palabra clave [var](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/var). La variable sigue recibiendo un tipo en tiempo de compilación, pero este lo proporciona el compilador. Para más información, vea [Variables locales con asignación implícita de tipos](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables).

En algunos casos, resulta conveniente crear un tipo con nombre para conjuntos sencillos de valores relacionados que no desea almacenar ni pasar fuera de los límites del método. Puede crear *tipos anónimos* para este fin. Para obtener más información, consulte [Tipos anónimos](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/classes-and-structs/anonymous-types) (Guía de programación de C#).

Los tipos de valor normales no pueden tener un valor [null](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/null), pero se pueden crear tipos de valor que aceptan valores NULL mediante la adición de ? después del tipo. Por ejemplo, int? es un tipo int que también puede tener el valor [null](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/keywords/null). Los tipos que admiten un valor NULL son instancias del tipo struct genérico [System.Nullable<T>](https://docs.microsoft.com/es-es/dotnet/api/system.nullable-1). Los tipos que admiten un valor NULL son especialmente útiles cuando hay un intercambio de datos con bases de datos en las que los valores numéricos podrían ser nulos. Para más información, vea [Tipos que admiten un valor NULL](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/nullable-value-types).

# **Conversiones de tipos**

Dado que C# tiene tipos estáticos en tiempo de compilación, después de declarar una variable, no se puede volver a declarar ni se le puede asignar un valor de otro tipo a menos que ese tipo sea convertible de forma implícita al tipo de la variable. Por ejemplo, string no se puede convertir de forma implícita a int. Por tanto, después de declarar i como un valor int, no se le puede asignar la cadena "Hello", como se muestra en el código siguiente:

C#

int i;

i = "Hello"; // error CS0029: Cannot implicitly convert type 'string' to 'int'

Pero es posible que en ocasiones sea necesario copiar un valor en una variable o parámetro de método de otro tipo. Por ejemplo, es posible que tenga una variable de entero que se necesita pasar a un método cuyo parámetro es de tipo double. O es posible que tenga que asignar una variable de clase a una variable de tipo de interfaz. Estos tipos de operaciones se denominan *conversiones de tipos*. En C#, se pueden realizar las siguientes conversiones de tipos:

* **Conversiones implícitas:** no se requiere ninguna sintaxis especial porque la conversión tiene seguridad de tipos y no se perderá ningún dato. Los ejemplos incluyen conversiones de tipos enteros más pequeños a más grandes, y conversiones de clases derivadas a clases base.
* **Conversiones explícitas**: las conversiones explícitas requieren una [expresión Cast](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/operators/type-testing-and-cast#cast-expression). La conversión es necesaria si es posible que se pierda información en la conversión, o cuando es posible que la conversión no sea correcta por otros motivos. Entre los ejemplos típicos están la conversión numérica a un tipo que tiene menos precisión o un intervalo más pequeño, y la conversión de una instancia de clase base a una clase derivada.
* **Conversiones con clases del asistente**: para realizar conversiones entre tipos no compatibles, como enteros y objetos [System.DateTime](https://docs.microsoft.com/es-es/dotnet/api/system.datetime), o cadenas hexadecimales y matrices de bytes puede usar la clase [System.BitConverter](https://docs.microsoft.com/es-es/dotnet/api/system.bitconverter), la clase [System.Convert](https://docs.microsoft.com/es-es/dotnet/api/system.convert) y los métodos Parse de los tipos numéricos integrados, como [Int32.Parse](https://docs.microsoft.com/es-es/dotnet/api/system.int32.parse). Para obtener más información, consulte [Procedimiento Convertir una matriz de bytes en un valor int](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/types/how-to-convert-a-byte-array-to-an-int), [Procedimiento Convertir una cadena en un número](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/types/how-to-convert-a-string-to-a-number) y [Procedimiento Convertir cadenas hexadecimales en tipos numéricos](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/types/how-to-convert-between-hexadecimal-strings-and-numeric-types).

## **Conversiones implícitas**

Para los tipos numéricos integrados, se puede realizar una conversión implícita cuando el valor que se va a almacenar se puede encajar en la variable sin truncarse ni redondearse. Para los tipos enteros, esto significa que el intervalo del tipo de origen es un subconjunto apropiado del intervalo para el tipo de destino. Por ejemplo, una variable de tipo [long](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) (entero de 64 bits) puede almacenar cualquier valor que un tipo [int](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) (entero de 32 bits) pueda almacenar. En el ejemplo siguiente, el compilador convierte de forma implícita el valor de num en la parte derecha a un tipo long antes de asignarlo a bigNum.

C#

// Implicit conversion. A long can

// hold any value an int can hold, and more!

int num = 2147483647;

long bigNum = num;

byte miByte = 124;

num = miByte;

bigNum = miByte;

Para obtener una lista completa de las conversiones numéricas implícitas, consulte la sección [Conversiones numéricas implícitas](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/numeric-conversions#implicit-numeric-conversions) del artículo [Conversiones numéricas integradas](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/numeric-conversions).

Para los tipos de referencia, siempre existe una conversión implícita desde una clase a cualquiera de sus interfaces o clases base directas o indirectas. No se necesita ninguna sintaxis especial porque una clase derivada siempre contiene a todos los miembros de una clase base.

C#

Derived d = new Derived();

Base b = d; // Always OK.

## **Conversiones explícitas**

Pero si no se puede realizar una conversión sin riesgo de perder información, el compilador requiere que se realice una conversión explícita, que se denomina *conversión*. Una conversión de tipos es una manera de informar explícitamente al compilador de que se pretende realizar la conversión y se es consciente de que se puede producir pérdida de datos. Para realizar una conversión, especifique el tipo al que se va a convertir entre paréntesis delante del valor o la variable que se va a convertir. El siguiente programa convierte un tipo [double](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) en un tipo [int](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/integral-numeric-types). El programa no se compilará sin la conversión.

C#

class Test

{

static void Main()

{

double x = 1234.7;

int a;

// Cast double to int.

a = (int)x;

System.Console.WriteLine(a);

}

}

// Output: 1234

Para obtener una lista completa de las conversiones numéricas explícitas admitidas, consulte la sección [Conversiones numéricas explícitas](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/numeric-conversions#explicit-numeric-conversions) del artículo [Conversiones numéricas integradas](https://docs.microsoft.com/es-mx/dotnet/csharp/language-reference/builtin-types/numeric-conversions).

Para los tipos de referencia, se requiere una conversión explícita si es necesario convertir de un tipo base a un tipo derivado:

C#

// Create a new derived type.

Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.

Animal a = g;

// Explicit conversion is required to cast back

// to derived type. Note: This will compile but will

// throw an exception at run time if the right-side

// object is not in fact a Giraffe.

Giraffe g2 = (Giraffe) a;

Una operación de conversión entre tipos de referencia no cambia el tipo en tiempo de ejecución del objeto subyacente. Solo cambia el tipo del valor que se usa como referencia a ese objeto. Para obtener más información, vea [Polimorfismo](https://docs.microsoft.com/es-mx/dotnet/csharp/programming-guide/classes-and-structs/polymorphism) .

## **Excepciones de conversión de tipos en tiempo de ejecución**

En algunas conversiones de tipos de referencia, el compilador no puede determinar si una conversión será válida. Es posible que una operación de conversión que se compile correctamente produzca un error en tiempo de ejecución. Como se muestra en el ejemplo siguiente, una conversión de tipo que produce un error en tiempo de ejecución produce una excepción [InvalidCastException](https://docs.microsoft.com/es-es/dotnet/api/system.invalidcastexception).

C#

using System;

class Animal

{

public void Eat() { Console.WriteLine("Eating."); }

public override string ToString()

{

return "I am an animal.";

}

}

class Reptile : Animal { }

class Mammal : Animal { }

class UnSafeCast

{

static void Main()

{

Test(new Mammal());

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

static void Test(Animal a)

{

// Cause InvalidCastException at run time

// because Mammal is not convertible to Reptile.

Reptile r = (Reptile)a;

}

}