

Contents

Documentación de F#

¿Qué es F#?

Primeros pasos

- Instalación de F#

- F# en Visual Studio

- F# en Visual Studio Code

- F# con la CLI de .NET Core

- F# en Visual Studio para Mac

Paseo por F#

Tutoriales

- Introducción a la programación funcional

 - Funciones de primera clase

- Programación asincrónica y simultánea

 - Programación asincrónica

- Proveedores de tipos

 - Creación de un proveedor de tipos

 - Seguridad de los proveedores de tipos

 - Solución de problemas en proveedores de tipos

- F# Interactive

Novedades de F#

- F# 4.7

- F# 4.6

- F# 4.5

Referencia del lenguaje F#

- Referencia de palabras clave

- Referencia de símbolos y operadores

 - Operadores aritméticos

 - Operadores booleanos

 - Operadores bit a bit

Operadores que aceptan valores NULL

Funciones

Enlaces let

do (Enlaces)

Expresiones lambda: palabra clave fun

Funciones recursivas: palabra clave rev

Punto de entrada

Funciones externas

Funciones insertadas

Valores

Valores NULL

Literales

Tipos en F#

Inferencia de tipos

Tipos básicos

Unit (Tipo)

Cadenas

Tuplas

Tipos de colección F#

Listas

Matrices

Secuencias

Segmentos

Opciones

Opciones de valores

Resultados

Genéricos

Generalización automática

Restricciones

Parámetros de tipo resueltos estáticamente

Registros

Registros anónimos

Expresiones de registro de copia y actualización

Uniones discriminadas

Enumeraciones

Abreviaturas de tipo

Clases

Estructuras

Herencia

Interfaces

Clases abstractas

Miembros

- Enlaces let en clases

- Enlaces do en clases

- Propiedades

- Propiedades indizadas

- Métodos

- Constructores

- Events

- Campos explícitos: palabra clave ``val``

Extensiones de tipo

Parámetros y argumentos

Sobrecarga de operadores

Tipos flexibles

Delegados

Expresiones de objeto

Conversiones

Control de acceso

Expresiones condicionales: `if...then...else`

Expresiones de coincidencia

Coincidencia de modelos

Patrones activos

Bucles: expresión `for...to`

Bucles: expresión `for...in`

Bucles: expresión while...do

Aserciones

Control de excepciones

Tipos de excepción

Expresión try...with

Expresión try...finally

Expresión raise

Expresión failwith

Expresión invalidArg

Atributos

Administración de recursos: palabra clave use

Espacios de nombres

Módulos

Declaraciones de importación: palabra clave open

Archivos de signatura

Unidades de medida

Documentación de XML

Expresiones diferidas

Expresiones de cálculo

Flujos de trabajo asincrónicos

Expresiones de consulta

Expresiones de código delimitadas

Palabra clave fixed

Byrefs

Celdas de referencia

Directivas de compilador

Opciones del compilador

Opciones de F# Interactive

Identificadores de línea, archivo y ruta de acceso de origen

Información del llamador

Sintaxis detallada

Errores y advertencias del compilador

Guía de estilo de F#

Instrucciones de formato de código de F#

Convenciones de código de F#

Instrucciones de diseño de componentes de F#

Uso de F# en Azure

Introducción a Azure Blob Storage mediante F#

Introducción a Azure File Storage mediante F#

Introducción a Azure Queue Storage mediante F#

Introducción a Azure Table Storage mediante F#

Administración de paquetes para las dependencias de Azure de F #

Qué es F#

23/11/2019 • 4 minutes to read • [Edit Online](#)

F# es un lenguaje de programación funcional que facilita la escritura de código correcto y fácil de mantener.

F# la programación implica principalmente definir tipos y funciones que se infieren de tipo y se generalizan automáticamente. Esto permite que el foco permanezca en el dominio del problema y manipule sus datos, en lugar de los detalles de la programación.

```
open System // Gets access to functionality in System namespace.

// Defines a function that takes a name and produces a greeting.
let getGreeting name =
    sprintf "Hello, %s! Isn't F# great?" name

[<EntryPoint>]
let main args =
    // Defines a list of names
    let names = [ "Don"; "Julia"; "Xi" ]

    // Prints a greeting for each name!
    names
    |> List.map getGreeting
    |> List.iter (fun greeting -> printfn "%s" greeting)

    0
```

F# tiene numerosas características, entre las que se incluyen:

- Sintaxis ligera
- Inmutable de forma predeterminada
- Inferencia de tipos y generalización automática
- Funciones de primera clase
- Tipos de datos eficaces
- Detección de patrones
- Programación asíncrona

En la [F# referencia del lenguaje](#) se documenta un conjunto completo de características.

Tipos de datos enriquecidos

Los tipos de datos como [registros](#) y [uniones discriminadas](#) permiten representar dominios y datos complejos.

```
// Group data with Records
type SuccessfulWithdrawal = {
    Amount: decimal
    Balance: decimal
}

type FailedWithdrawal = {
    Amount: decimal
    Balance: decimal
    IsOverdraft: bool
}

// Use discriminated unions to represent data of 1 or more forms
type WithdrawalResult =
    | Success of SuccessfulWithdrawal
    | InsufficientFunds of FailedWithdrawal
    | CardExpired of System.DateTime
    | UndisclosedFailure
```

F#los registros y las uniones discriminadas son no NULL, inmutables y comparables de forma predeterminada, lo que facilita su uso.

Aplicación de corrección con funciones y coincidencia de patrones

F#las funciones son fáciles de declarar y eficaces en la práctica. Cuando se combina con la [coincidencia de patrones](#), permiten definir el comportamiento cuya corrección se aplica mediante el compilador.

```
// Returns a WithdrawalResult
let withdrawMoney amount = // Implementation elided

let handleWithdrawal amount =
    let w = withdrawMoney amount

    // The F# compiler enforces accounting for each case!
    match w with
    | Success s -> printfn "Successfully withdrew %f" s.Amount
    | InsufficientFunds f -> printfn "Failed: balance is %f" f.Balance
    | CardExpired d -> printfn "Failed: card expired on %0" d
    | UndisclosedFailure -> printfn "Failed: unknown :{"
```

F#las funciones son también de primera clase, lo que significa que se pueden pasar como parámetros y devolverse desde otras funciones.

Funciones para definir operaciones en objetos

F#tiene compatibilidad total con objetos, que son tipos de datos útiles cuando es necesario combinar datos y funcionalidad. F#las funciones se usan para manipular objetos.

```

type Set<'T when 'T: comparison>(elements: seq<'T>) =
    member s.IsEmpty = // Implementation elided
    member s.Contains (value) = // Implementation elided
    member s.Add (value) = // Implementation elided
    // ...
    // Further Implementation elided
    // ...
    interface IEnumerable<'T>
    interface IReadOnlyCollection<'T>

module Set =
    let isEmpty (set: Set<'T>) = set.IsEmpty

    let contains element (set: Set<'T>) = set.Contains(element)

    let add value (set: Set<'T>) = set.Add(value)

```

En lugar de escribir código orientado a objetos, en F#, a menudo escribirá código que trata los objetos como otro tipo de datos para las funciones que se van a manipular. Las características como las [interfaces genéricas](#), las [expresiones de objeto](#) y el uso prudente de [los miembros](#) son comunes en los programas más grandes. F#

Pasos siguientes

Para obtener más información acerca de un conjunto F# mayor de características, consulte el [F# paseo](#).

Introducción a F#

19/12/2019 • 2 minutes to read • [Edit Online](#)

Puede empezar a trabajar con F# en el equipo o en línea.

Introducción a la máquina

Hay varias guías sobre cómo instalar y usar F# por primera vez en la máquina. Puede utilizar la tabla siguiente como ayuda para tomar una decisión:

SO	PREFERIR VISUAL STUDIO	PREFERIR VISUAL STUDIO CODE	PREFERIR LÍNEA DE COMANDOS
Windows	Introducción a Visual Studio	Introducción a Visual Studio Code	Introducción a la CLI de .NET Core
macOS	Introducción a VS para Mac	Introducción a Visual Studio Code	Introducción a la CLI de .NET Core
Linux	N/D	Introducción a Visual Studio Code	Introducción a la CLI de .NET Core

En general, no hay ningún específico que sea mejor que el resto. Se recomienda probar todas las formas de F# usar en el equipo para ver lo que le gusta mejor.

Empezar a trabajar en línea

Si prefiere no instalar F# y .net en el equipo, también puede empezar a trabajar con F# en el explorador:

- [La introducción F# a on Binder](#) es un [cuaderno de Jupyter Notebook](#) en Hosted mediante el servicio de [enlazador](#) gratuito. No es necesario realizar ningún registro.
- La [REPL de Fable](#) es una REPL interactiva y en el explorador que usa [Fable](#) para traducir código F# en JavaScript. Consulte los numerosos ejemplos que abarcan desde F# los aspectos básicos hasta un juego de vídeo totalmente completo que se ejecuta en el explorador.

Instalar F#

19/03/2020 • 3 minutes to read • [Edit Online](#)

Puede instalar F de varias maneras, dependiendo de su entorno.

Instalar F con Visual Studio

1. Si va a descargar [Visual Studio](#) por primera vez, primero instalará Visual Studio Installer. Instale la edición adecuada de Visual Studio desde el instalador.

Si ya tiene visual de Visual Studio instalado, elija **Modificar** junto a la edición a la que desea agregar F.

2. En la página Cargas de trabajo, seleccione la carga de **trabajo de desarrollo ASP.NET y web**, que incluye compatibilidad con F y .NET Core para proyectos de ASP.NET Core.
3. Elija **Modificar** en la esquina inferior derecha para instalar todo lo que ha seleccionado.

A continuación, puede abrir Visual Studio con F, elija **Iniciar** en el instalador de Visual Studio.

Instalar F con Visual Studio Code

1. Asegúrese de [que](#) git está instalado y disponible en su PATH. Puede comprobar que está instalado correctamente `git --version` introduciendo en un símbolo del sistema y pulsando **Intro**.
2. Instale el SDK de [.NET Core](#) y Visual Studio [Code](#).
3. Seleccione el icono Extensiones y busque "Ionide":

El único complemento necesario para la compatibilidad con F en Visual Studio Code es [Ionide-fsharp](#). Sin embargo, también puede instalar [Ionide-FAKE](#) para obtener soporte [FAKE](#) e [Ionide-Paket](#) para obtener soporte [de Paket](#). FAKE y Paket son herramientas adicionales de la comunidad de F para crear proyectos y administrar dependencias, respectivamente.

Instalar F con Visual Studio para Mac

F se instala de forma predeterminada en [Visual Studio para Mac](#), independientemente de la configuración que elija.

Una vez completada la instalación, elija **Iniciar Visual Studio**. También puede abrir Visual Studio a través de Finder en macOS.

Instalar F en un servidor de compilación

Si usa .NET Core o .NET Framework a través del SDK de .NET, solo tiene que instalar el SDK de .NET en el servidor de compilación. Tiene todo lo que necesita.

Si usa .NET Framework y **no** usa el SDK de .NET, deberá instalar la [SKU](#) de Visual Studio Build Tools en Windows Server. En el instalador, seleccione Herramientas de compilación de escritorio de **.NET**, a continuación, seleccione el componente **del compilador** de F en el lado derecho del menú del instalador.

Introducción a F# en Visual Studio

08/01/2020 • 6 minutes to read • [Edit Online](#)

F# y las herramientas F# visuales se admiten en el entorno de desarrollo integrado (IDE) de Visual Studio.

Para empezar, asegúrese de que tiene [Visual Studio instalado con F# compatibilidad](#).

Creación de una aplicación de consola

Uno de los proyectos más básicos de Visual Studio es la aplicación de consola. Aquí se muestra cómo crear uno:

1. Abra Visual Studio 2019.
2. En la ventana de inicio, elija **Crear un proyecto nuevo**.
3. En la página **crear un nuevo proyecto**, elija **F#** en la lista idioma.
4. Seleccione la plantilla **aplicación de consola (.net Core)** y, a continuación, elija **siguiente**.
5. En la página **configurar el nuevo proyecto**, escriba un nombre en el cuadro **nombre del proyecto**. Luego, elija **Crear**.

Visual Studio crea el nuevo F# proyecto. Puede verlo en la ventana Explorador de soluciones.

Escribir el código

Comencemos por escribir código. Asegúrese de que el archivo de `Program.fs` está abierto y, a continuación, reemplace el contenido por lo siguiente:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

En el ejemplo de código anterior se define una función llamada `square` que toma una entrada denominada `x` y la multiplica por sí misma. Dado F# que utiliza la [inferencia de tipos](#), no es necesario especificar el tipo de `x`. El F# compilador entiende los tipos en los que la multiplicación es válida y asigna un tipo a `x` en función de cómo se llama a `square`. Si mantiene el mouse sobre `square`, debería ver lo siguiente:

```
val square: x:int -> int
```

Esto es lo que se conoce como la firma de tipo de la función. Se puede leer de la siguiente manera: "Square es una función que toma un entero denominado x y genera un entero". El compilador dio `square` el tipo de `int` por ahora; Esto se debe a que la multiplicación no es genérica en *todos los* tipos, sino en un conjunto cerrado de tipos. El F# compilador ajustará la firma de tipo si llama a `square` con un tipo de entrada diferente, como un `float`.

Otra función, `main`, se define, que está decorada con el atributo `EntryPoint`. Este atributo indica al F# compilador que la ejecución del programa debe comenzar allí. Sigue la misma Convención que otros [lenguajes de programación de estilo C](#), donde se pueden pasar argumentos de línea de comandos a esta función y se devuelve

un código entero (normalmente `0`).

Está en la función de punto de entrada, `main`, que llama a la función de `square` con un argumento de `12`. A F# continuación, el compilador asigna el tipo de `square` que se va a `int -> int` (es decir, una función que toma un `int` y genera un `int`). La llamada a `printfn` es una función de impresión con formato que usa una cadena de formato e imprime el resultado (y una nueva línea). La cadena de formato, similar a los lenguajes de programación de estilo C, tiene parámetros (`%d`) que corresponden a los argumentos que se le pasan, en este caso `12` y `(square 12)`.

Ejecución del código

Puede ejecutar el código y ver los resultados presionando **Ctrl+F5**. Como alternativa, puede elegir el **>** de **depurar iniciar sin depurar** en la barra de menús de nivel superior. Esto ejecuta el programa sin depuración.

El siguiente resultado se imprime en la ventana de la consola que abrió Visual Studio:

```
12 squared is: 144!
```

¡Enhorabuena! Ha creado su primer F# proyecto en Visual Studio, ha escrito una F# función que calcula e imprime un valor y ejecuta el proyecto para ver los resultados.

Pasos siguientes

Si no lo ha hecho ya, consulte el [paseo F#](#) por, que cubre algunas de las características principales del F# lenguaje. Proporciona información general sobre algunas de las funcionalidades de F# y ejemplos de código que puede copiar en Visual Studio y ejecutar.

[Paseo por F](#)

Vea también

- [F#Referencia del lenguaje](#)
- [Inferencia de tipos](#)
- [Referencia de símbolos y operadores](#)

Introducción a F# en Visual Studio Code

05/02/2020 • 14 minutes to read • [Edit Online](#)

Puede escribir F# en [Visual Studio Code](#) con el [complemento Ionide](#) para obtener una excelente experiencia de entorno de desarrollo integrado (IDE) multiplataforma y flexible con IntelliSense y refactorizaciones de código. Visite [Ionide.IO](#) para obtener más información sobre el complemento.

Para empezar, asegúrese de que tiene [F# y el complemento Ionide instalado correctamente](#).

Cree su primer proyecto con Ionide

Para crear un nuevo F# proyecto, abra una línea de comandos y cree un nuevo proyecto con el CLI de .net Core:

```
dotnet new console -lang "F#" -o FirstIonideProject
```

Una vez finalizado, cambie el directorio al proyecto y abra Visual Studio Code:

```
cd FirstIonideProject  
code .
```

Después de que el proyecto se cargue en Visual Studio Code, debería F# ver el panel de explorador de soluciones en el lado izquierdo de la ventana abierta. Esto significa que Ionide ha cargado correctamente el proyecto que acaba de crear. Puede escribir código en el editor antes de este momento, pero cuando esto suceda, todo ha terminado de cargarse.

Configuración F# de Interactive

En primer lugar, asegúrese de que el scripting de .NET Core es el entorno de scripting predeterminado:

1. Abra la configuración de Visual Studio Code (**código > preferencias > configuración**).
2. Busque el término **F# script**.
3. Haga clic en la casilla que indica **FSharp: usar scripts de SDK**.

Esto es necesario actualmente debido a algunos comportamientos heredados en scripting basado en .NET Framework que no funcionan con el scripting de .NET Core y Ionide está intentando actualmente la compatibilidad con versiones anteriores. En el futuro, el scripting de .NET Core se convertirá en el valor predeterminado.

Escribir el primer script

Una vez que haya configurado Visual Studio Code para usar el scripting de .NET Core, vaya a la vista del explorador en Visual Studio Code y cree un archivo nuevo. Asígnele el nombre *MyFirstScript.FSX*.

Ahora, agregue el siguiente código:

```

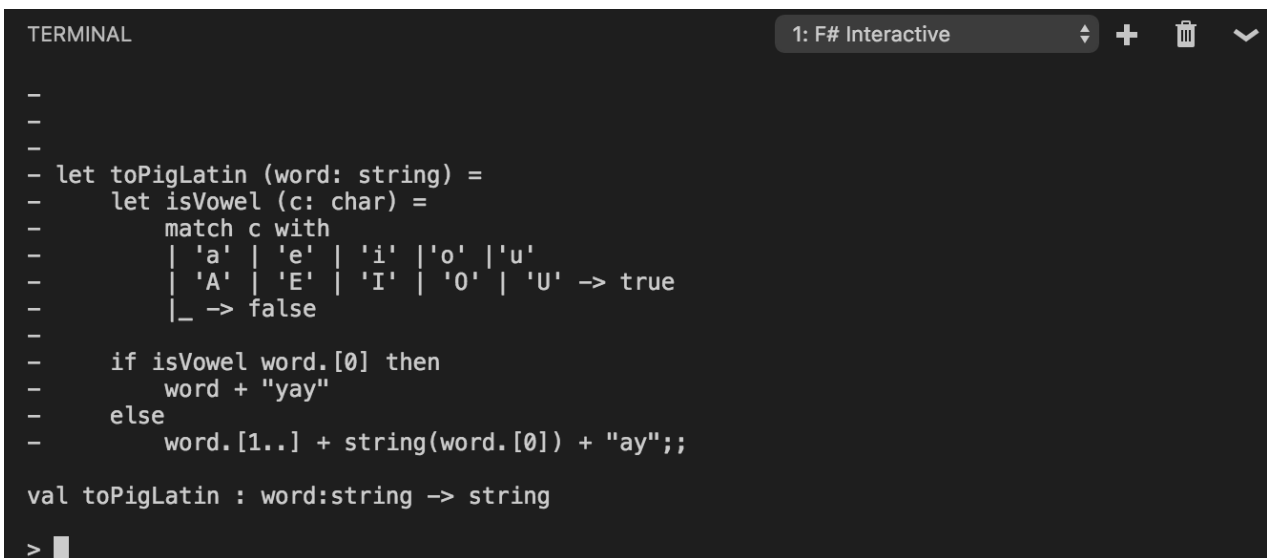
let toPigLatin (word: string) =
    let isVowel (c: char) =
        match c with
        | 'a' | 'e' | 'i' | 'o' | 'u'
        | 'A' | 'E' | 'I' | 'O' | 'U' -> true
        | _ -> false

    if isVowel word.[0] then
        word + "yay"
    else
        word.[1..] + string(word.[0]) + "ay"

```

Esta función convierte una palabra en una forma de [Pig Latin](#). El siguiente paso es evaluarlo mediante F# Interactive (FSI).

Resalte toda la función (debe tener 11 líneas de longitud). Una vez resaltado, mantenga presionada la tecla **Alt** y presione **entrar**. Observará que aparece una ventana de terminal en la parte inferior de la pantalla y debería tener un aspecto similar al siguiente:



The screenshot shows a terminal window titled "1: F# Interactive". The code from the previous block is pasted into the terminal, with each line preceded by a minus sign (-) as a visual indicator of selection. The code is:


```

- let toPigLatin (word: string) =
-     let isVowel (c: char) =
-         match c with
-         | 'a' | 'e' | 'i' | 'o' | 'u'
-         | 'A' | 'E' | 'I' | 'O' | 'U' -> true
-         | _ -> false
-
-     if isVowel word.[0] then
-         word + "yay"
-     else
-         word.[1..] + string(word.[0]) + "ay";;
-
- val toPigLatin : word:string -> string
  
```

 The prompt ">" is visible at the bottom left of the terminal.

Esto hizo tres cosas:

1. Se inició el proceso FSI.
2. Envío el código que resaltó en el proceso FSI.
3. El proceso FSI evaluó el código que ha enviado.

Dado que lo que envió es una [función](#), ahora puede llamar a esa función con FSI. En la ventana interactiva, escriba lo siguiente:

```
toPigLatin "banana";;
```

Debería ver el siguiente resultado:

```
val it : string = "ananabay"
```

Ahora, vamos a probar una vocal como la primera letra. Escriba lo siguiente:

```
toPigLatin "apple";;
```

Debería ver el siguiente resultado:

```
val it : string = "applayay"
```

Parece que la función funciona según lo esperado. Enhorabuena, acaba de escribir su primera F# función en Visual Studio Code y la evaluó con FSI.

NOTE

Como puede haber observado, las líneas de FSI finalizan con `;;`. Esto se debe a que FSI le permite escribir varias líneas. El `;;` al final permite que FSI sepa cuándo finaliza el código.

Explicación del código

Si no está seguro de lo que realmente está haciendo el código, aquí se muestra paso a paso.

Como puede ver, `toPigLatin` es una función que toma una palabra como su entrada y la convierte en una representación de un Pig de esa palabra. Las reglas para esto son las siguientes:

Si el primer carácter de una palabra comienza con una vocal, agregue "Yay" al final de la palabra. Si no se inicia con una vocal, mueva el primer carácter al final de la palabra y agréguele "Ay".

Es posible que haya observado lo siguiente en FSI:

```
val toPigLatin : word:string -> string
```

Esto indica que `toPigLatin` es una función que toma un `string` como entrada (denominada `word`) y devuelve otro `string`. Esto se conoce como la [firma de tipo de la función](#), una parte fundamental F# de la clave para entender F# el código. También observará esto si mantiene el puntero sobre la función en Visual Studio Code.

En el cuerpo de la función, observará dos partes distintas:

1. Función interna, denominada `isVowel`, que determina si un carácter determinado (`c`) es una vocal comprobando si coincide con uno de los patrones proporcionados a través de la [coincidencia de patrones](#):

```
let isVowel (c: char) =  
    match c with  
    | 'a' | 'e' | 'i' | 'o' | 'u'  
    | 'A' | 'E' | 'I' | 'O' | 'U' -> true  
    | _ -> false
```

2. `if..then..else` expresión que comprueba si el primer carácter es una vocal y construye un valor devuelto fuera de los caracteres de entrada en función de si el primer carácter era una vocal o no:

```
if isVowel word.[0] then  
    word + "yay"  
else  
    word.[1..] + string(word.[0]) + "ay"
```

El flujo de `toPigLatin` es así:

Comprueba si el primer carácter de la palabra de entrada es una vocal. Si es así, adjunte "Yay" al final de la palabra. En caso contrario, mueva el primer carácter al final de la palabra y agréguele "Ay".

Hay un aspecto final que se debe tener en cuenta: F# en, no hay ninguna instrucción explícita que devolver desde la función. Esto se debe F# a que se basa en expresiones y la última expresión evaluada en el cuerpo de una función

determina el valor devuelto de esa función. Dado que `if..then..else` es en sí misma una expresión, la evaluación del cuerpo del bloque de `then` o el cuerpo del bloque de `else` determina el valor devuelto por la función de `toPigLatin`.

Convertir la aplicación de consola en un generador de Pig Latin

En las secciones anteriores de este artículo se ha mostrado un primer paso F# común en la escritura del código: escribir una función inicial y ejecutarla de forma interactiva con FSI. Esto se conoce como desarrollo controlado por REPL, donde [REPL](#) representa el bucle de lectura-evaluación-impresión. Es una excelente manera de experimentar con la funcionalidad hasta que tenga algo que funcione.

El siguiente paso del desarrollo controlado por REPL es trasladar el código de trabajo F# a un archivo de implementación. Después, el F# compilador puede compilar en un ensamblado que se pueda ejecutar.

Para empezar, abra el archivo *Program.FS* que creó anteriormente con el CLI de .net Core. Observará que parte del código ya está en él.

A continuación, cree una nueva `module` denominada `PigLatin` y copie en ella la función de `toPigLatin` que creó anteriormente:

```
module PigLatin =
    let toPigLatin (word: string) =
        let isVowel (c: char) =
            match c with
            | 'a' | 'e' | 'i' | 'o' | 'u'
            | 'A' | 'E' | 'I' | 'O' | 'U' -> true
            | _ -> false

        if isVowel word.[0] then
            word + "yay"
        else
            word.[1..] + string(word.[0]) + "ay"
```

Este módulo debe estar por encima de la `main` función y debajo de la declaración de `open System`. El orden de las declaraciones es F#importante en, por lo que deberá definir la función antes de llamarla en un archivo.

Ahora, en la función `main`, llame a la función de generador de Pig Latin en los argumentos:

```
[<EntryPoint>]
let main argv =
    for name in argv do
        let newName = PigLatin.toPigLatin name
        printfn "%s in Pig Latin is: %s" name newName

    0
```

Ahora puede ejecutar la aplicación de consola desde la línea de comandos:

```
dotnet run apple banana
```

Y verá que genera el mismo resultado que el archivo de script, pero esta vez como un programa en ejecución.

Solución de problemas de Ionide

Estas son algunas de las formas en que puede solucionar ciertos problemas que pueden surgir:

1. Para obtener las características de edición de código de Ionide F#, los archivos deben guardarse en el disco y

dentro de una carpeta que esté abierta en el área de trabajo Visual Studio Code.

2. Si ha realizado cambios en el sistema o ha instalado los requisitos previos de Ionide con Visual Studio Code abrir, reinicie Visual Studio Code.
3. Si tiene caracteres no válidos en los directorios del proyecto, es posible que Ionide no funcione. Cambie el nombre de los directorios del proyecto si éste es el caso.
4. Si no funciona ninguno de los comandos de Ionide, compruebe los [enlaces de teclado de Visual Studio Code](#) para ver si se están reemplazando por accidente.
5. Si Ionide se ha interrumpido en el equipo y ninguno de los anteriores ha solucionado el problema, intente quitar el directorio de `ionide-fsharp` de la máquina y vuelva a instalar el conjunto de complementos.
6. Si un proyecto no se cargó (el F# explorador de soluciones lo mostrará), haga clic con el botón derecho en el proyecto y haga clic en **Ver detalles** para obtener más información de diagnóstico.

Ionide es un proyecto de código abierto creado y mantenido por miembros de F# la comunidad. Informe de los problemas y no dude en contribuir en el [repositorio de github ionide-vscode-FSharp](#).

También puede solicitar más ayuda de la comunidad y F# los desarrolladores de Ionide en el [canal de Gitter de Ionide](#).

Pasos siguientes

Para obtener más información F# acerca de y las características del lenguaje, consulte el [paseo F#](#) por.

Comience con F# el CLI de .net Core

21/02/2020 • 4 minutes to read • [Edit Online](#)

En este artículo se explica cómo puede empezar a F# trabajar con en cualquier sistema operativo (Windows, MacOS o Linux) con el CLI de .net Core. Pasa por la creación de una solución de varios proyectos con una biblioteca de clases a la que llama una aplicación de consola.

Prerequisites

Para empezar, debe instalar la [SDK de .net Core](#) más reciente.

En este artículo se supone que sabe cómo usar una línea de comandos y tener un editor de texto preferido. Si aún no lo usa, [Visual Studio Code](#) es una excelente opción como editor de texto para F#.

Compilar una solución sencilla de varios proyectos

Abra un símbolo del sistema o terminal y use el comando `dotnet New` para crear un nuevo archivo de solución denominado `FSNetCore`:

```
dotnet new sln -o FSNetCore
```

Después de ejecutar el comando anterior, se genera la siguiente estructura de directorios:

```
FSNetCore
├── FSNetCore.sln
```

Escribir una biblioteca de clases

Cambie los directorios a `FSNetCore`.

Use el comando `dotnet new`, cree un proyecto de biblioteca de clases en la carpeta `src` denominada `Library`.

```
dotnet new classlib -lang "F#" -o src/Library
```

Después de ejecutar el comando anterior, se genera la siguiente estructura de directorios:

```
├── FSNetCore
│   ├── FSNetCore.sln
│   └── src
│       ├── Library
│       │   ├── Library.fs
│       └── Library.fsproj
```

Reemplace el contenido de `Library.fs` por el código siguiente:

```
module Library

open Newtonsoft.Json

let getJsonNetJson value =
    sprintf "I used to be %s but now I'm %s thanks to JSON.NET!" value (JsonConvert.SerializeObject(value))
```

Agregue el paquete NuGet Newtonsoft.Json al proyecto de biblioteca.

```
dotnet add src/Library/Library.fsproj package Newtonsoft.Json
```

Agregue el proyecto `Library` a la solución de `FSNetCore` con el comando [dotnet sln Add](#) :

```
dotnet sln add src/Library/Library.fsproj
```

Ejecute `dotnet build` para compilar el proyecto. Las dependencias no resueltas se restaurarán al compilar.

Escribir una aplicación de consola que consuma la biblioteca de clases

Use el comando `dotnet new`, cree una aplicación de consola en la carpeta `src` denominada `app`.

```
dotnet new console -lang "F#" -o src/App
```

Después de ejecutar el comando anterior, se genera la siguiente estructura de directorios:

```
└─ FSNetCore
   └─ FSNetCore.sln
      └─ src
         └─ App
            ├── App.fsproj
            └─ Program.fs
         └─ Library
            ├── Library.fs
            └─ Library.fsproj
```

Reemplace el contenido del archivo `Program.fs` por el código siguiente:

```
open System
open Library

[<EntryPoint>]
let main argv =
    printfn "Nice command-line arguments! Here's what JSON.NET has to say about them:"

    argv
    |> Array.map getJsonNetJson
    |> Array.iter (printfn "%s")

    0 // return an integer exit code
```

Agregue una referencia al proyecto `Library` mediante [dotnet Add Reference](#).

```
dotnet add src/App/App.fsproj reference src/Library/Library.fsproj
```

Agregue el proyecto `App` a la solución de `FSNetCore` con el comando `dotnet sln add` :

```
dotnet sln add src/App/App.fsproj
```

Restaurar las dependencias de NuGet, `dotnet restore` y ejecute `dotnet build` para compilar el proyecto.

Cambie el directorio al proyecto de consola de `src/App` y ejecute el proyecto pasando `Hello World` como argumentos:

```
cd src/App  
dotnet run Hello World
```

Debería ver los siguientes resultados:

```
Nice command-line arguments! Here's what JSON.NET has to say about them:
```

```
I used to be Hello but now I'm ""Hello"" thanks to JSON.NET!
```

```
I used to be World but now I'm ""World"" thanks to JSON.NET!
```

Pasos siguientes

A continuación, consulte el [paseo F#](#) por para obtener más información sobre F# las distintas características.

Introducción a F# en Visual Studio para Mac

29/11/2019 • 10 minutes to read • [Edit Online](#)

F# y las herramientas F# visuales se admiten en el IDE de Visual Studio para Mac. Asegúrese de que ha [instalado Visual Studio para Mac](#).

Creación de una aplicación de consola

Uno de los proyectos más básicos de Visual Studio para Mac es la aplicación de consola. A continuación le mostramos cómo hacerlo. Una vez que Visual Studio para Mac está abierto:

1. En el menú **archivo**, elija **nueva solución**.
2. En el cuadro de diálogo nuevo proyecto, hay dos plantillas diferentes para la aplicación de consola. Hay una en otras > .NET que tiene como destino el .NET Framework. La otra plantilla está en .NET Core-> aplicación que tiene como destino .NET Core. Cualquier plantilla debe funcionar para este artículo.
3. En aplicación de consola, C# cambie F# a si es necesario. Elija el botón **siguiente** para avanzar.
4. Asigne un nombre al proyecto y elija las opciones que quiera para la aplicación. Fíjese en el panel de vista previa en el lateral de la pantalla que mostrará la estructura de directorios que se creará en función de las opciones seleccionadas.
5. Haga clic en **Crear**. Ahora debería ver un F# proyecto en el explorador de soluciones.

Escritura del código

Vamos a empezar por escribir código primero. Asegúrese de que el archivo de `Program.fs` está abierto y, a continuación, reemplace el contenido por lo siguiente:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

En el ejemplo de código anterior, se ha definido una función `square` que toma una entrada denominada `x` y la multiplica por sí misma. Dado F# que utiliza la [inferencia de tipos](#), no es necesario especificar el tipo de `x`. El F# compilador entiende los tipos en los que la multiplicación es válida y asignará un tipo a `x` basándose en cómo se llama a `square`. Si mantiene el mouse sobre `square`, debería ver lo siguiente:

```
val square: x:int -> int
```

Esto es lo que se conoce como la firma de tipo de la función. Se puede leer de la siguiente manera: "Square es una función que toma un entero denominado x y genera un entero". Tenga en cuenta que el compilador dio `square` el tipo de `int` por ahora: esto se debe a que la multiplicación no es genérica en *todos los* tipos, sino que es genérica en un conjunto cerrado de tipos. El F# compilador eligió `int` en este punto, pero ajustará la firma del tipo si llama a `square` con un tipo de entrada diferente, como un `float`.

Otra función, `main`, se define, que está decorada con el atributo `EntryPoint` para indicar F# al compilador que la ejecución del programa debe comenzar allí. Sigue la misma Convención que otros [lenguajes de programación de estilo C](#), donde se pueden pasar argumentos de línea de comandos a esta función y se devuelve un código entero (normalmente `0`).

En esta función se llama a la función `square` con un argumento de `12`. A F# continuación, el compilador asigna el tipo de `square` que se va a `int -> int` (es decir, una función que toma un `int` y genera un `int`). La llamada a `printfn` es una función de impresión con formato que usa una cadena de formato, similar a los lenguajes de programación de estilo C, los parámetros que se corresponden con los especificados en la cadena de formato y, a continuación, imprime el resultado y una nueva línea.

Ejecución del código

Puede ejecutar el código y ver los resultados haciendo clic en **Ejecutar** en el menú de nivel superior y, a continuación, **iniciar sin depurar**. Así se ejecutará el programa sin depurar y podrá ver los resultados.

Ahora debería ver lo siguiente impreso en la ventana de la consola que Visual Studio para Mac emerge:

```
12 squared is 144!
```

¡Enhorabuena! Ha creado su primer F# proyecto en Visual Studio para Mac, ha escrito una F# función que ha impreso los resultados de llamar a esa función y ejecuta el proyecto para ver algunos resultados.

Uso F# de Interactive

Una de las mejores características de las herramientas F# visuales en Visual Studio para Mac es la F# ventana interactiva. Permite enviar código a un proceso en el que se puede llamar a ese código y ver el resultado de forma interactiva.

Para empezar a usarlo, resalte la función `square` definida en el código. A continuación, haga clic en **Editar** en el menú de nivel superior. A continuación, seleccione **enviar F# selección a interactivo**. Esto ejecuta el código en la F# ventana interactiva. Como alternativa, puede hacer clic con el botón derecho en la selección y elegir **Enviar selección a F# interactivo**. Debería ver que la F# ventana interactiva aparece con lo siguiente en ella:

```
>

val square : x:int -> int

>
```

Esto muestra la misma firma de función para la función `square`, que vio anteriormente al mantener el mouse sobre la función. Dado que `square` se define ahora en F# el proceso interactivo, puede llamarlo con valores diferentes:

```
> square 12;;
val it : int = 144
>square 13;;
val it : int = 169
```

De esta forma, se ejecuta la función, se enlaza el resultado a un nuevo nombre `it` y se muestra el tipo y el valor de `it`. Tenga en cuenta que debe terminar cada línea con `;;`. Así es como F# la interactiva sabe cuándo finaliza la llamada de función. También puede definir nuevas funciones en F# Interactive:

```
> let isOdd x = x % 2 <> 0;;  
  
val isOdd : x:int -> bool  
  
> isOdd 12;;  
val it : bool = false
```

Lo anterior define una nueva función, `isOdd`, que toma un `int` y comprueba si es impar. Puede llamar a esta función para ver lo que devuelve con diferentes entradas. Puede llamar a funciones dentro de las llamadas de función:

```
> isOdd (square 15);;  
val it : bool = true
```

También puede usar el [operador de canalización directa](#) para canalizar el valor en las dos funciones:

```
> 15 |> square |> isOdd;;  
val it : bool = true
```

El operador de canalización hacia delante, y más, se trata en los tutoriales posteriores.

Esto solo es un vistazo a lo que puede hacer con F# el Interactive. Para obtener más información, consulte [programación interactiva F#con](#) .

Pasos siguientes

Si no lo ha hecho ya, consulte el [paseo F#](#) por, que cubre algunas de las características principales del F# lenguaje. Le proporcionará una visión general de algunas de las funcionalidades de F#y proporcionará ejemplos de código que puede copiar en Visual Studio para Mac y ejecutar. También hay algunos excelentes recursos externos que puede usar, que se muestran en la [F# guía](#).

Vea también

- [F#Guía](#)
- [Paseo por F](#)
- [F#Referencia del lenguaje](#)
- [Inferencia de tipos](#)
- [Referencia de símbolos y operadores](#)

Paseo por F#

12/02/2020 • 42 minutes to read • [Edit Online](#)

Es la mejor manera para obtener información sobre F# leer y escribir código de F#. En este artículo actuará como un paseo por algunas de las características clave del lenguaje F# y proporcionarle algunos fragmentos de código que se pueden ejecutar en el equipo. Para obtener información sobre cómo configurar un entorno de desarrollo, consulte [Introducción](#).

Hay dos conceptos principales en F#: tipos y funciones. En este paseo se resaltan las características del lenguaje que se encuentran en estos dos conceptos.

Ejecutar el código en línea

Si no tiene F# instalado en el equipo, puede ejecutar todos los ejemplos en el explorador con [try F# on webassembly](#). Fable es un dialecto de F# que se ejecuta directamente en el explorador. Para ver los ejemplos siguientes en REPL, consulte los [ejemplos](#) > [Aprenda](#) > [Tour F#](#) en la barra de menús de la izquierda del fable repl.

Funciones y módulos

Las partes más fundamentales de cualquier F# programa son *funciones* organizadas en *módulos*. [Las funciones](#) realizan trabajos en entradas para generar salidas y se organizan en [módulos](#), que son la manera principal de F#agrupar elementos. Se definen mediante el [enlace de `let`](#), que asigna un nombre a la función y definen sus argumentos.


```

module BasicFunctions =

    /// You use 'let' to define a function. This one accepts an integer argument and returns an integer.
    /// Parentheses are optional for function arguments, except for when you use an explicit type
    annotation.
    let sampleFunction1 x = x*x + 3

    /// Apply the function, naming the function return result using 'let'.
    /// The variable type is inferred from the function return type.
    let result1 = sampleFunction1 4573

    // This line uses '%d' to print the result as an integer. This is type-safe.
    // If 'result1' were not of type 'int', then the line would fail to compile.
    printfn "The result of squaring the integer 4573 and adding 3 is %d" result1

    /// When needed, annotate the type of a parameter name using '(argument:type)'. Parentheses are
    required.
    let sampleFunction2 (x:int) = 2*x*x - x/5 + 3

    let result2 = sampleFunction2 (7 + 4)
    printfn "The result of applying the 2nd sample function to (7 + 4) is %d" result2

    /// Conditionals use if/then/elif/else.
    ///
    /// Note that F# uses white space indentation-aware syntax, similar to languages like Python.
    let sampleFunction3 x =
        if x < 100.0 then
            2.0*x*x - x/5.0 + 3.0
        else
            2.0*x*x + x/5.0 - 37.0

    let result3 = sampleFunction3 (6.5 + 4.5)

    // This line uses '%f' to print the result as a float. As with '%d' above, this is type-safe.
    printfn "The result of applying the 3rd sample function to (6.5 + 4.5) is %f" result3

```

`let` enlaces también son cómo enlazar un valor a un nombre, de forma similar a una variable en otros lenguajes. de forma predeterminada, los enlaces de `let` son *inmutables*, lo que significa que una vez que un valor o una función se enlaza a un nombre, no se puede cambiar en contexto. Esto contrasta con las variables de otros lenguajes, que son *mutables*, lo que significa que sus valores se pueden cambiar en cualquier momento en el tiempo. Si necesita un enlace mutable, puede usar `let mutable ...` sintaxis.

```

module Immutability =

    /// Binding a value to a name via 'let' makes it immutable.
    ///
    /// The second line of code fails to compile because 'number' is immutable and bound.
    /// Re-defining 'number' to be a different value is not allowed in F#.
    let number = 2
    // let number = 3

    /// A mutable binding. This is required to be able to mutate the value of 'otherNumber'.
    let mutable otherNumber = 2

    printfn "'otherNumber' is %d" otherNumber

    // When mutating a value, use '<-' to assign a new value.
    //
    // Note that '=' is not the same as this. '=' is used to test equality.
    otherNumber <- otherNumber + 1

    printfn "'otherNumber' changed to be %d" otherNumber

```

Números, valores booleanos y cadenas

Como lenguaje .NET, F# admite los mismos [tipos primitivos](#) subyacentes que existen en .net.

Le mostramos cómo varios tipos numéricos se representan en F#:

```
module IntegersAndNumbers =

    /// This is a sample integer.
    let sampleInteger = 176

    /// This is a sample floating point number.
    let sampleDouble = 4.1

    /// This computed a new number by some arithmetic. Numeric types are converted using
    /// functions 'int', 'double' and so on.
    let sampleInteger2 = (sampleInteger/4 + 5 - 7) * 4 + int sampleDouble

    /// This is a list of the numbers from 0 to 99.
    let sampleNumbers = [ 0 .. 99 ]

    /// This is a list of all tuples containing all the numbers from 0 to 99 and their squares.
    let sampleTableOfSquares = [ for i in 0 .. 99 -> (i, i*i) ]

    // The next line prints a list that includes tuples, using '%A' for generic printing.
    printfn "The table of squares from 0 to 99 is:\n%A" sampleTableOfSquares
```

Estos son los valores booleanos y la lógica condicional básica tiene el siguiente aspecto:

```
module Booleans =

    /// Booleans values are 'true' and 'false'.
    let boolean1 = true
    let boolean2 = false

    /// Operators on booleans are 'not', '&&' and '||'.
    let boolean3 = not boolean1 && (boolean2 || false)

    // This line uses '%b' to print a boolean value. This is type-safe.
    printfn "The expression 'not boolean1 && (boolean2 || false)' is %b" boolean3
```

Y este es el aspecto básico de la manipulación de [cadenas](#) :

```

module StringManipulation =

    /// Strings use double quotes.
    let string1 = "Hello"
    let string2 = "world"

    /// Strings can also use @ to create a verbatim string literal.
    /// This will ignore escape characters such as '\', '\n', '\t', etc.
    let string3 = @"C:\Program Files\"

    /// String literals can also use triple-quotes.
    let string4 = """The computer said "hello world" when I told it to!"""

    /// String concatenation is normally done with the '+' operator.
    let helloWorld = string1 + " " + string2

    // This line uses '%s' to print a string value. This is type-safe.
    printfn "%s" helloWorld

    /// Substrings use the indexer notation. This line extracts the first 7 characters as a substring.
    /// Note that like many languages, Strings are zero-indexed in F#.
    let substring = helloWorld.[0..6]
    printfn "%s" substring

```

Tuplas

Las **tuplas** son un gran trato F#en. Son una agrupación de valores sin nombre, pero ordenados, que se pueden tratar como propios valores. Considérelos como valores que se agregan a partir de otros valores. Tienen muchos usos, como la devolución cómoda de varios valores de una función o la agrupación de valores para una conveniencia ad hoc.

```

module Tuples =

    /// A simple tuple of integers.
    let tuple1 = (1, 2, 3)

    /// A function that swaps the order of two values in a tuple.
    ///
    /// F# Type Inference will automatically generalize the function to have a generic type,
    /// meaning that it will work with any type.
    let swapElems (a, b) = (b, a)

    printfn "The result of swapping (1, 2) is %A" (swapElems (1,2))

    /// A tuple consisting of an integer, a string,
    /// and a double-precision floating point number.
    let tuple2 = (1, "fred", 3.1415)

    printfn "tuple1: %A\ttuple2: %A" tuple1 tuple2

```

También puede crear tuplas `struct`. También interoperan totalmente con las tuplas de C# 7/Visual Basic 15, que también se `struct` tuplas:

```

/// Tuples are normally objects, but they can also be represented as structs.
///
/// These interoperate completely with structs in C# and Visual Basic.NET; however,
/// struct tuples are not implicitly convertible with object tuples (often called reference tuples).
///
/// The second line below will fail to compile because of this. Uncomment it to see what happens.
let sampleStructTuple = struct (1, 2)
//let thisWillNotCompile: (int*int) = struct (1, 2)

// Although you can
let convertFromStructTuple (struct(a, b)) = (a, b)
let convertToStructTuple (a, b) = struct(a, b)

printfn "Struct Tuple: %A\nReference tuple made from the Struct Tuple: %A" sampleStructTuple
(sampleStructTuple |> convertFromStructTuple)

```

Es importante tener en cuenta que, dado que `struct` tuplas son tipos de valor, no se pueden convertir implícitamente en tuplas de referencia, o viceversa. Debe convertir explícitamente entre una tupla de referencia y `struct`.

Canalizaciones y composición

Los operadores de canalización como `|>` se usan en gran medida al F# procesar los datos en. Estos operadores son funciones que permiten establecer "canalizaciones" de funciones de una manera flexible. En el ejemplo siguiente se describe cómo puede aprovechar estos operadores para crear una canalización funcional sencilla:

```

module PipelinesAndComposition =

    /// Squares a value.
    let square x = x * x

    /// Adds 1 to a value.
    let addOne x = x + 1

    /// Tests if an integer value is odd via modulo.
    let isOdd x = x % 2 <> 0

    /// A list of 5 numbers. More on lists later.
    let numbers = [ 1; 2; 3; 4; 5 ]

    /// Given a list of integers, it filters out the even numbers,
    /// squares the resulting odds, and adds 1 to the squared odds.
    let squareOddValuesAndAddOne values =
        let odds = List.filter isOdd values
        let squares = List.map square odds
        let result = List.map addOne squares
        result

    printfn "processing %A through 'squareOddValuesAndAddOne' produces: %A" numbers
    (squareOddValuesAndAddOne numbers)

    /// A shorter way to write 'squareOddValuesAndAddOne' is to nest each
    /// sub-result into the function calls themselves.
    ///
    /// This makes the function much shorter, but it's difficult to see the
    /// order in which the data is processed.
    let squareOddValuesAndAddOneNested values =
        List.map addOne (List.map square (List.filter isOdd values))

    printfn "processing %A through 'squareOddValuesAndAddOneNested' produces: %A" numbers
    (squareOddValuesAndAddOneNested numbers)

    /// A preferred way to write 'squareOddValuesAndAddOne' is to use F# pipe operators.
    /// This allows you to avoid creating intermediate results, but is much more readable
    /// than nesting function calls like 'squareOddValuesAndAddOneNested'
    let squareOddValuesAndAddOnePipeline values =
        values
        |> List.filter isOdd
        |> List.map square
        |> List.map addOne

    printfn "processing %A through 'squareOddValuesAndAddOnePipeline' produces: %A" numbers
    (squareOddValuesAndAddOnePipeline numbers)

    /// You can shorten 'squareOddValuesAndAddOnePipeline' by moving the second `List.map` call
    /// into the first, using a Lambda Function.
    ///
    /// Note that pipelines are also being used inside the lambda function. F# pipe operators
    /// can be used for single values as well. This makes them very powerful for processing data.
    let squareOddValuesAndAddOneShorterPipeline values =
        values
        |> List.filter isOdd
        |> List.map(fun x -> x |> square |> addOne)

    printfn "processing %A through 'squareOddValuesAndAddOneShorterPipeline' produces: %A" numbers
    (squareOddValuesAndAddOneShorterPipeline numbers)

```

En el ejemplo anterior se utilizaban muchas características F#de, incluidas las funciones de procesamiento de lista, las funciones de primera clase y la [aplicación parcial](#). Aunque una comprensión profunda de cada uno de estos conceptos puede ser un poco más avanzada, debe estar claro cómo se pueden usar las funciones para procesar datos al crear canalizaciones.

Listas, matrices y secuencias

Las listas, matrices y las secuencias son tres tipos de colección principal en la biblioteca básica de F#.

[Las listas](#) son colecciones ordenadas e inmutables de elementos del mismo tipo. Se trata de listas vinculadas individualmente, lo que significa que están pensadas para la enumeración, pero es una opción deficiente para el acceso aleatorio y la concatenación si son grandes. Esto contrasta con las listas de otros lenguajes populares, que normalmente no usan una lista vinculada individualmente para representar listas.

```
module Lists =

    /// Lists are defined using [ ... ]. This is an empty list.
    let list1 = [ ]

    /// This is a list with 3 elements. ';' is used to separate elements on the same line.
    let list2 = [ 1; 2; 3 ]

    /// You can also separate elements by placing them on their own lines.
    let list3 = [
        1
        2
        3
    ]

    /// This is a list of integers from 1 to 1000
    let numberList = [ 1 .. 1000 ]

    /// Lists can also be generated by computations. This is a list containing
    /// all the days of the year.
    let daysList =
        [ for month in 1 .. 12 do
            for day in 1 .. System.DateTime.DaysInMonth(2017, month) do
                yield System.DateTime(2017, month, day) ]

    // Print the first 5 elements of 'daysList' using 'List.take'.
    printfn "The first 5 days of 2017 are: %A" (daysList |> List.take 5)

    /// Computations can include conditionals. This is a list containing the tuples
    /// which are the coordinates of the black squares on a chess board.
    let blackSquares =
        [ for i in 0 .. 7 do
            for j in 0 .. 7 do
                if (i+j) % 2 = 1 then
                    yield (i, j) ]

    /// Lists can be transformed using 'List.map' and other functional programming combinators.
    /// This definition produces a new list by squaring the numbers in numberList, using the pipeline
    /// operator to pass an argument to List.map.
    let squares =
        numberList
        |> List.map (fun x -> x*x)

    /// There are many other list combinations. The following computes the sum of the squares of the
    /// numbers divisible by 3.
    let sumOfSquares =
        numberList
        |> List.filter (fun x -> x % 3 = 0)
        |> List.sumBy (fun x -> x * x)

    printfn "The sum of the squares of numbers up to 1000 that are divisible by 3 is: %d" sumOfSquares
```

Las [matrices](#) son colecciones *mutables* de tamaño fijo de elementos del mismo tipo. Se admiten el acceso aleatorio rápido de elementos y son más rápidas que F# listas porque son simplemente contiguos bloques de memoria.

```

module Arrays =

    /// This is The empty array. Note that the syntax is similar to that of Lists, but uses `[| ... |]`
    instead.
    let array1 = [| |]

    /// Arrays are specified using the same range of constructs as lists.
    let array2 = [| "hello"; "world"; "and"; "hello"; "world"; "again" |]

    /// This is an array of numbers from 1 to 1000.
    let array3 = [| 1 .. 1000 |]

    /// This is an array containing only the words "hello" and "world".
    let array4 =
        [| for word in array2 do
            if word.Contains("l") then
                yield word |]

    /// This is an array initialized by index and containing the even numbers from 0 to 2000.
    let evenNumbers = Array.init 1001 (fun n -> n * 2)

    /// Sub-arrays are extracted using slicing notation.
    let evenNumbersSlice = evenNumbers.[0..500]

    /// You can loop over arrays and lists using 'for' loops.
    for word in array4 do
        printfn "word: %s" word

    // You can modify the contents of an array element by using the left arrow assignment operator.
    //
    // To learn more about this operator, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/values/index#mutable-variables
    array2.[1] <- "WORLD!"

    /// You can transform arrays using 'Array.map' and other functional programming operations.
    /// The following calculates the sum of the lengths of the words that start with 'h'.
    let sumOfLengthsOfWords =
        array2
        |> Array.filter (fun x -> x.StartsWith("h"))
        |> Array.sumBy (fun x -> x.Length)

    printfn "The sum of the lengths of the words in Array 2 is: %d" sumOfLengthsOfWords

```

Las **secuencias** son una serie lógica de elementos, todo del mismo tipo. Se trata de un tipo más general que las listas y matrices, capaz de ser su "vista" en cualquier serie lógica de elementos. También destacan porque pueden ser *Lazy*, lo que significa que los elementos solo se pueden calcular cuando son necesarios.

```

module Sequences =

    /// This is the empty sequence.
    let seq1 = Seq.empty

    /// This a sequence of values.
    let seq2 = seq { yield "hello"; yield "world"; yield "and"; yield "hello"; yield "world"; yield "again"
    }

    /// This is an on-demand sequence from 1 to 1000.
    let numbersSeq = seq { 1 .. 1000 }

    /// This is a sequence producing the words "hello" and "world"
    let seq3 =
        seq { for word in seq2 do
            if word.Contains("l") then
                yield word }

    /// This sequence producing the even numbers up to 2000.
    let evenNumbers = Seq.init 1001 (fun n -> n * 2)

    let rnd = System.Random()

    /// This is an infinite sequence which is a random walk.
    /// This example uses yield! to return each element of a subsequence.
    let rec randomWalk x =
        seq { yield x
            yield! randomWalk (x + rnd.NextDouble() - 0.5) }

    /// This example shows the first 100 elements of the random walk.
    let first100ValuesOfRandomWalk =
        randomWalk 5.0
        |> Seq.truncate 100
        |> Seq.toList

    printfn "First 100 elements of a random walk: %A" first100ValuesOfRandomWalk

```

Funciones recursivas

El procesamiento de colecciones o secuencias de elementos se realiza normalmente con F# [recursividad](#) en. Aunque F# tiene compatibilidad con bucles y la programación imperativa, recursividad es preferible porque es más fácil de garantizar la corrección.

NOTE

En el ejemplo siguiente se usa la coincidencia de patrones a través de la expresión `match`. Esta construcción fundamental se trata más adelante en este artículo.


```

module RecursiveFunctions =

    /// This example shows a recursive function that computes the factorial of an
    /// integer. It uses 'let rec' to define a recursive function.
    let rec factorial n =
        if n = 0 then 1 else n * factorial (n-1)

    printfn "Factorial of 6 is: %d" (factorial 6)

    /// Computes the greatest common factor of two integers.
    ///
    /// Since all of the recursive calls are tail calls,
    /// the compiler will turn the function into a loop,
    /// which improves performance and reduces memory consumption.
    let rec greatestCommonFactor a b =
        if a = 0 then b
        elif a < b then greatestCommonFactor a (b - a)
        else greatestCommonFactor (a - b) b

    printfn "The Greatest Common Factor of 300 and 620 is %d" (greatestCommonFactor 300 620)

    /// This example computes the sum of a list of integers using recursion.
    let rec sumList xs =
        match xs with
        | [] -> 0
        | y::ys -> y + sumList ys

    /// This makes 'sumList' tail recursive, using a helper function with a result accumulator.
    let rec private sumListTailRecHelper accumulator xs =
        match xs with
        | [] -> accumulator
        | y::ys -> sumListTailRecHelper (accumulator+y) ys

    /// This invokes the tail recursive helper function, providing '0' as a seed accumulator.
    /// An approach like this is common in F#.
    let sumListTailRecursive xs = sumListTailRecHelper 0 xs

    let oneThroughTen = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

    printfn "The sum 1-10 is %d" (sumListTailRecursive oneThroughTen)

```

F# también tiene compatibilidad total para la optimización de llamar al final, que es una forma de optimizar las llamadas recursivas para que sean tan rápidos como una construcción de bucle.

Tipos de unión de registro y discriminado

Registro y tipos de unión son dos tipos de datos fundamentales utilizados en el código de F# y suelen ser la mejor manera de representar los datos en un programa de F#. Aunque esto hace que sean similares a las clases de otros lenguajes, una de sus principales diferencias es que tienen una semántica de igualdad estructural. Esto significa que se comparan de forma "nativa" y la igualdad es sencilla. Compruebe si uno es igual que el otro.

Los [registros](#) son un agregado de valores con nombre, con miembros opcionales (como métodos). Si está familiarizado con C# o Java, estos deberían ser similares a poco o POJO, solo con igualdad estructural y menos ceremonia.

```

module RecordTypes =

    /// This example shows how to define a new record type.
    type ContactCard =
        { Name      : string
          Phone     : string
          Verified  : bool }

    /// This example shows how to instantiate a record type.
    let contact1 =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false }

    /// You can also do this on the same line with ';' separators.
    let contactOnSameLine = { Name = "Alf"; Phone = "(206) 555-0157"; Verified = false }

    /// This example shows how to use "copy-and-update" on record values. It creates
    /// a new record value that is a copy of contact1, but has different values for
    /// the 'Phone' and 'Verified' fields.
    ///
    /// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/copy-and-update-record-expressions
    let contact2 =
        { contact1 with
          Phone = "(206) 555-0112"
          Verified = true }

    /// This example shows how to write a function that processes a record value.
    /// It converts a 'ContactCard' object to a string.
    let showContactCard (c: ContactCard) =
        c.Name + " Phone: " + c.Phone + (if not c.Verified then " (unverified)" else "")

    printfn "Alf's Contact Card: %s" (showContactCard contact1)

    /// This is an example of a Record with a member.
    type ContactCardAlternate =
        { Name      : string
          Phone     : string
          Address   : string
          Verified  : bool }

    /// Members can implement object-oriented members.
    member this.PrintedContactCard =
        this.Name + " Phone: " + this.Phone + (if not this.Verified then " (unverified)" else "") +
        this.Address

    let contactAlternate =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false
          Address = "111 Alf Street" }

    // Members are accessed via the '.' operator on an instantiated type.
    printfn "Alf's alternate contact card is %s" contactAlternate.PrintedContactCard

```

También puede representar registros como Structs. Esto se hace con el atributo `[<Struct>]`:

```
/// Records can also be represented as structs via the 'Struct' attribute.  
/// This is helpful in situations where the performance of structs outweighs  
/// the flexibility of reference types.  
[<Struct>]  
type ContactCardStruct =  
    { Name      : string  
      Phone     : string  
      Verified  : bool }
```

Las [uniones discriminadas \(DUs\)](#) son valores que pueden ser una serie de formularios o casos con nombre. Los datos almacenados en el tipo pueden tener uno de varios valores distintos.

```

module DiscriminatedUnions =

    /// The following represents the suit of a playing card.
    type Suit =
        | Hearts
        | Clubs
        | Diamonds
        | Spades

    /// A Discriminated Union can also be used to represent the rank of a playing card.
    type Rank =
        /// Represents the rank of cards 2 .. 10
        | Value of int
        | Ace
        | King
        | Queen
        | Jack

    /// Discriminated Unions can also implement object-oriented members.
    static member GetAllRanks() =
        [ yield Ace
          for i in 2 .. 10 do yield Value i
          yield Jack
          yield Queen
          yield King ]

    /// This is a record type that combines a Suit and a Rank.
    /// It's common to use both Records and Discriminated Unions when representing data.
    type Card = { Suit: Suit; Rank: Rank }

    /// This computes a list representing all the cards in the deck.
    let fullDeck =
        [ for suit in [ Hearts; Diamonds; Clubs; Spades] do
          for rank in Rank.GetAllRanks() do
            yield { Suit=suit; Rank=rank } ]

    /// This example converts a 'Card' object to a string.
    let showPlayingCard (c: Card) =
        let rankString =
            match c.Rank with
            | Ace -> "Ace"
            | King -> "King"
            | Queen -> "Queen"
            | Jack -> "Jack"
            | Value n -> string n
        let suitString =
            match c.Suit with
            | Clubs -> "clubs"
            | Diamonds -> "diamonds"
            | Spades -> "spades"
            | Hearts -> "hearts"
        rankString + " of " + suitString

    /// This example prints all the cards in a playing deck.
    let printAllCards() =
        for card in fullDeck do
            printfn "%s" (showPlayingCard card)

```

También puede usar DUs como *uniones discriminadas de un solo caso* para ayudar con el modelado de dominios a través de tipos primitivos. A menudo, las veces, las cadenas y otros tipos primitivos se utilizan para representar algo y, por tanto, se les da un significado determinado. Sin embargo, el uso de la representación primitiva de los datos puede dar lugar a la asignación errónea de un valor incorrecto. Representar cada tipo de información como una Unión de un solo caso distinto puede exigir una corrección en este escenario.

```
// Single-case DUs are often used for domain modeling. This can buy you extra type safety
// over primitive types such as strings and ints.
//
// Single-case DUs cannot be implicitly converted to or from the type they wrap.
// For example, a function which takes in an Address cannot accept a string as that input,
// or vice versa.
type Address = Address of string
type Name = Name of string
type SSN = SSN of int

// You can easily instantiate a single-case DU as follows.
let address = Address "111 Alf Way"
let name = Name "Alf"
let ssn = SSN 1234567890

/// When you need the value, you can unwrap the underlying value with a simple function.
let unwrapAddress (Address a) = a
let unwrapName (Name n) = n
let unwrapSSN (SSN s) = s

// Printing single-case DUs is simple with unwrapping functions.
printfn "Address: %s, Name: %s, and SSN: %d" (address |> unwrapAddress) (name |> unwrapName) (ssn |>
unwrapSSN)
```

Como se muestra en el ejemplo anterior, para obtener el valor subyacente en una Unión discriminada de un solo caso, debe desencapsularlo explícitamente.

Además, DUs también admite definiciones recursivas, lo que le permite representar fácilmente árboles y datos recursivos de forma inherente. Por ejemplo, aquí se muestra cómo puede representar un árbol de búsqueda binaria con funciones `exists` y `insert`.

```
/// Discriminated Unions also support recursive definitions.
///
/// This represents a Binary Search Tree, with one case being the Empty tree,
/// and the other being a Node with a value and two subtrees.
type BST<'T> =
    | Empty
    | Node of value:'T * left: BST<'T> * right: BST<'T>

/// Check if an item exists in the binary search tree.
/// Searches recursively using Pattern Matching. Returns true if it exists; otherwise, false.
let rec exists item bst =
    match bst with
    | Empty -> false
    | Node (x, left, right) ->
        if item = x then true
        elif item < x then (exists item left) // Check the left subtree.
        else (exists item right) // Check the right subtree.

/// Inserts an item in the Binary Search Tree.
/// Finds the place to insert recursively using Pattern Matching, then inserts a new node.
/// If the item is already present, it does not insert anything.
let rec insert item bst =
    match bst with
    | Empty -> Node(item, Empty, Empty)
    | Node(x, left, right) as node ->
        if item = x then node // No need to insert, it already exists; return the node.
        elif item < x then Node(x, insert item left, right) // Call into left subtree.
        else Node(x, left, insert item right) // Call into right subtree.
```

Dado que DUs le permite representar la estructura recursiva del árbol en el tipo de datos, el funcionamiento de esta estructura recursiva es sencillo y garantiza la corrección. También se admite en la coincidencia de patrones, como se muestra a continuación.

Además, puede representar DUs como `struct`s con el atributo `[<Struct>]`:

```
/// Discriminated Unions can also be represented as structs via the 'Struct' attribute.
/// This is helpful in situations where the performance of structs outweighs
/// the flexibility of reference types.
///
/// However, there are two important things to know when doing this:
///     1. A struct DU cannot be recursively-defined.
///     2. A struct DU must have unique names for each of its cases.
[<Struct>]
type Shape =
    | Circle of radius: float
    | Square of side: float
    | Triangle of height: float * width: float
```

Sin embargo, hay dos aspectos clave que hay que tener en cuenta al hacerlo:

1. No se puede definir un struct DU de forma recursiva.
2. Una estructura DU debe tener nombres únicos para cada uno de sus casos.

Si no se sigue el anterior, se producirá un error de compilación.

Coincidencia de modelos

La [coincidencia](#) de patrones F# es la característica de lenguaje que permite el funcionamiento F# en tipos. En los ejemplos anteriores, probablemente detectó bastante `match x with ...` sintaxis. Esta construcción permite al compilador, que puede comprender la "forma" de los tipos de datos, para obligarle a tener en cuenta todos los casos posibles al usar un tipo de datos a través de lo que se conoce como coincidencia de patrones exhaustiva. Esto es increíblemente eficaz para corregir y se puede usar de forma inteligente para "levantar" lo que normalmente sería un problema de tiempo de ejecución en tiempo de compilación.

```

module PatternMatching =

  /// A record for a person's first and last name
  type Person = {
    First : string
    Last  : string
  }

  /// A Discriminated Union of 3 different kinds of employees
  type Employee =
    | Engineer of engineer: Person
    | Manager of manager: Person * reports: List<Employee>
    | Executive of executive: Person * reports: List<Employee> * assistant: Employee

  /// Count everyone underneath the employee in the management hierarchy,
  /// including the employee. The matches bind names to the properties
  /// of the cases so that those names can be used inside the match branches.
  /// Note that the names used for binding do not need to be the same as the
  /// names given in the DU definition above.
  let rec countReports(emp : Employee) =
    1 + match emp with
    | Engineer(person) ->
      0
    | Manager(person, reports) ->
      reports |> List.sumBy countReports
    | Executive(person, reports, assistant) ->
      (reports |> List.sumBy countReports) + countReports assistant

  /// Find all managers/executives named "Dave" who do not have any reports.
  /// This uses the 'function' shorthand to as a lambda expression.
  let rec findDaveWithOpenPosition(emps : List<Employee>) =
    emps
    |> List.filter(function
      | Manager({First = "Dave"}, []) -> true // [] matches an empty list.
      | Executive({First = "Dave"}, [], _) -> true
      | _ -> false) // '_' is a wildcard pattern that matches anything.
      // This handles the "or else" case.

```

Algo que puede haber observado es el uso del patrón de `_`. Esto se conoce como el [patrón de carácter comodín](#), que es una forma de decir "no me importa qué es algo". Aunque es práctico, puede omitir accidentalmente la coincidencia de patrones exhaustivos y dejar de beneficiarse de las impuestas en tiempo de compilación si no tiene cuidado al usar `_`. Se recomienda usar cuando no le interesan ciertas partes de un tipo descompuesto al buscar coincidencias de patrones o la cláusula final cuando ha enumerado todos los casos significativos en una expresión de coincidencia de patrones.

En el ejemplo siguiente, se utiliza el caso `_` cuando se produce un error en una operación de análisis.

```

open System

/// You can also use the shorthand function construct for pattern matching,
/// which is useful when you're writing functions which make use of Partial Application.
let private parseHelper (f: string -> bool * 'T) = f >> function
    | (true, item) -> Some item
    | (false, _) -> None

let parseDateTimeOffset = parseHelper DateTimeOffset.TryParse

let result = parseDateTimeOffset "1970-01-01"
match result with
| Some dto -> printfn "It parsed!"
| None -> printfn "It didn't parse!"

// Define some more functions which parse with the helper function.
let parseInt = parseHelper Int32.TryParse
let parseDouble = parseHelper Double.TryParse
let parseTimeSpan = parseHelper TimeSpan.TryParse

```

Los **modelos activos** son otra construcción eficaz que se usa con la coincidencia de patrones. Permiten particionar los datos de entrada en formularios personalizados, descomponerlos en el sitio de llamada de coincidencia de patrones. También se pueden parametrizar, lo que permite a definir la partición como una función. Expandir el ejemplo anterior para admitir patrones activos tiene un aspecto similar al siguiente:

```

// Active Patterns are another powerful construct to use with pattern matching.
// They allow you to partition input data into custom forms, decomposing them at the pattern match call site.
//
// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/active-patterns
let (|Int|_|) = parseInt
let (|Double|_|) = parseDouble
let (|Date|_|) = parseDateTimeOffset
let (|TimeSpan|_|) = parseTimeSpan

/// Pattern Matching via 'function' keyword and Active Patterns often looks like this.
let printParseResult = function
    | Int x -> printfn "%d" x
    | Double x -> printfn "%f" x
    | Date d -> printfn "%s" (d.ToString())
    | TimeSpan t -> printfn "%s" (t.ToString())
    | _ -> printfn "Nothing was parse-able!"

// Call the printer with some different values to parse.
printParseResult "12"
printParseResult "12.045"
printParseResult "12/28/2016"
printParseResult "9:01PM"
printParseResult "banana!"

```

Tipos opcionales

Un caso especial de tipos de unión discriminada es el tipo de opción, que es tan útil que forma parte de la biblioteca básica de F#.

El **tipo de opción** es un tipo que representa uno de los dos casos: un valor o nada en absoluto. Se usa en cualquier escenario en el que un valor pueda o no ser el resultado de una operación determinada. Esto le obliga a tener en cuenta los dos casos, convirtiéndolo en un problema de tiempo de compilación en lugar de un problema en tiempo de ejecución. A menudo se usan en las API donde `null` se usa para representar "Nothing" en su lugar, lo que elimina la necesidad de preocuparse por `NullReferenceException` en muchas circunstancias.


```

/// Option values are any kind of value tagged with either 'Some' or 'None'.
/// They are used extensively in F# code to represent the cases where many other
/// languages would use null references.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/options
module OptionValues =

    /// First, define a zip code defined via Single-case Discriminated Union.
    type ZipCode = ZipCode of string

    /// Next, define a type where the ZipCode is optional.
    type Customer = { ZipCode: ZipCode option }

    /// Next, define an interface type that represents an object to compute the shipping zone for the
    customer's zip code,
    /// given implementations for the 'getState' and 'getShippingZone' abstract methods.
    type IShippingCalculator =
        abstract GetState : ZipCode -> string option
        abstract GetShippingZone : string -> int

    /// Next, calculate a shipping zone for a customer using a calculator instance.
    /// This uses combinators in the Option module to allow a functional pipeline for
    /// transforming data with Optionals.
    let CustomerShippingZone (calculator: IShippingCalculator, customer: Customer) =
        customer.ZipCode
        |> Option.bind calculator.GetState
        |> Option.map calculator.GetShippingZone

```

Unidades de medida

Una característica exclusiva de sistema de tipos de F# es la capacidad para proporcionar contexto para literales numéricos a través de las unidades de medida.

Las [unidades de medida](#) permiten asociar un tipo numérico a una unidad, como los medidores, y hacer que las funciones realicen trabajos en unidades en lugar de literales numéricos. Esto permite al compilador comprobar que los tipos de literales numéricos pasados tienen sentido en un contexto determinado, con lo que se eliminan los errores en tiempo de ejecución asociados a ese tipo de trabajo.

```

/// Units of measure are a way to annotate primitive numeric types in a type-safe way.
/// You can then perform type-safe arithmetic on these values.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/units-of-measure
module UnitsOfMeasure =

    /// First, open a collection of common unit names
    open Microsoft.FSharp.Data.UnitSystems.SI.UnitNames

    /// Define a unitized constant
    let sampleValue1 = 1600.0<meter>

    /// Next, define a new unit type
    [<Measure>]
    type mile =
        /// Conversion factor mile to meter.
        static member asMeter = 1609.34<meter/mile>

    /// Define a unitized constant
    let sampleValue2 = 500.0<mile>

    /// Compute metric-system constant
    let sampleValue3 = sampleValue2 * mile.asMeter

    // Values using Units of Measure can be used just like the primitive numeric type for things like
    printing.
    printfn "After a %f race I would walk %f miles which would be %f meters" sampleValue1 sampleValue2
    sampleValue3

```

La biblioteca básica de F# define muchos tipos de unidad del SI y conversiones de unidades. Para obtener más información, consulte el [espacio de nombres Microsoft.FSharp.Data.UnitSystems.Si](#).

Clases e interfaces

F#también es totalmente compatible con las clases, [interfaces](#), [clases abstractas](#), [herencia](#), etc. de .net.

[Las clases](#) son tipos que representan objetos .net, que pueden tener propiedades, métodos y eventos como [miembros](#).

```

/// Classes are a way of defining new object types in F#, and support standard Object-oriented constructs.
/// They can have a variety of members (methods, properties, events, etc.)
///
/// To learn more about Classes, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/classes
///
/// To learn more about Members, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/members
module DefiningClasses =

    /// A simple two-dimensional Vector class.
    ///
    /// The class's constructor is on the first line,
    /// and takes two arguments: dx and dy, both of type 'double'.
    type Vector2D(dx : double, dy : double) =

        /// This internal field stores the length of the vector, computed when the
        /// object is constructed
        let length = sqrt (dx*dx + dy*dy)

        /// 'this' specifies a name for the object's self-identifier.
        /// In instance methods, it must appear before the member name.
        member this.DX = dx

        member this.DY = dy

        member this.Length = length

        /// This member is a method. The previous members were properties.
        member this.Scale(k) = Vector2D(k * this.DX, k * this.DY)

    /// This is how you instantiate the Vector2D class.
    let vector1 = Vector2D(3.0, 4.0)

    /// Get a new scaled vector object, without modifying the original object.
    let vector2 = vector1.Scale(10.0)

    printfn "Length of vector1: %f\nLength of vector2: %f" vector1.Length vector2.Length

```

La definición de clases genéricas también es muy sencilla.

```

/// Generic classes allow types to be defined with respect to a set of type parameters.
/// In the following, 'T' is the type parameter for the class.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/generics/
module DefiningGenericClasses =

    type StateTracker<'T>(initialElement: 'T) =

        /// This internal field store the states in a list.
        let mutable states = [ initialElement ]

        /// Add a new element to the list of states.
        member this.UpdateState newState =
            states <- newState :: states // use the '<-' operator to mutate the value.

        /// Get the entire list of historical states.
        member this.History = states

        /// Get the latest state.
        member this.Current = states.Head

    /// An 'int' instance of the state tracker class. Note that the type parameter is inferred.
    let tracker = StateTracker 10

    // Add a state
    tracker.UpdateState 17

```

Para implementar una interfaz, puede usar `interface ... with` sintaxis o una expresión de [objeto](#).

```

/// Interfaces are object types with only 'abstract' members.
/// Object types and object expressions can implement interfaces.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/interfaces
module ImplementingInterfaces =

    /// This is a type that implements IDisposable.
    type ReadFile() =

        let file = new System.IO.StreamReader("readme.txt")

        member this.ReadLine() = file.ReadLine()

        // This is the implementation of IDisposable members.
        interface System.IDisposable with
            member this.Dispose() = file.Close()

    /// This is an object that implements IDisposable via an Object Expression
    /// Unlike other languages such as C# or Java, a new type definition is not needed
    /// to implement an interface.
    let interfaceImplementation =
        { new System.IDisposable with
            member this.Dispose() = printfn "disposed" }

```

Qué tipos se van a usar

La presencia de clases, registros, uniones discriminadas y tuplas conduce a una pregunta importante: ¿Qué debería usar? Como la mayoría de los casos, la respuesta depende de sus circunstancias.

Las tuplas son excelentes para devolver varios valores de una función y usar un agregado ad hoc de valores como un valor en sí.

Los registros son un "paso activo" de las tuplas, que tienen etiquetas con nombre y admiten miembros

opcionales. Son excelentes para una representación de datos en tránsito con un bajo nivel de ceremonia a través del programa. Dado que tienen igualdad estructural, son fáciles de usar con la comparación.

Las uniones discriminadas tienen muchos usos, pero la ventaja principal es poder usarlas junto con la coincidencia de patrones para tener en cuenta todas las posibles "formas" que pueden tener los datos.

Las clases son excelentes para una gran cantidad de razones, como cuando es necesario representar información y también asociar esa información a la funcionalidad. Como regla general, cuando tiene funcionalidad que está vinculada conceptualmente a algunos datos, el uso de clases y los principios de la programación orientada a objetos es una gran ventaja. Las clases son también el tipo de datos preferido al interoperar con C# y Visual Basic, ya que estos lenguajes usan clases para casi todo.

Pasos siguientes

Ahora que ha visto algunas de las principales características del lenguaje, debería estar listo para escribir sus primeros programas de F#! Consulte [Introducción](#) para obtener información sobre cómo configurar el entorno de desarrollo y escribir código.

Los siguientes pasos para obtener más información pueden ser el que desee, pero se recomienda la [Introducción a la F# programación funcional en](#) para familiarizarse con los conceptos básicos de la programación funcional. Estos serán esenciales en la creación de programas sólidos en F#.

Además, consulte la [F# referencia del lenguaje](#) para ver una colección completa de contenido conceptual sobre F#.

Introducción a la programación funcional en F#

04/11/2019 • 15 minutes to read • [Edit Online](#)

La programación funcional es un estilo de programación que enfatiza el uso de funciones y datos inmutables. La programación funcional con tipo es cuando la programación funcional se combina con tipos estáticos, F# como con. En general, los conceptos siguientes se destacan en la programación funcional:

- Funciona como construcciones principales que se usan
- Expresiones en lugar de instrucciones
- Valores inmutables en variables
- Programación declarativa a través de la programación imperativa

A lo largo de esta serie, explorará conceptos y patrones en la F#programación funcional con. A lo largo del proceso, también aprenderá algunas F# .

Terminología

La programación funcional, al igual que otros paradigmas de programación, incluye un vocabulario que finalmente tendrá que aprender. Estos son algunos de los términos comunes que verá todo el tiempo:

- **Función** : una función es una construcción que generará una salida cuando se proporcione una entrada. Más formalmente, *asigna* un elemento de un conjunto a otro conjunto. Este formalismo se eleva en concreto de muchas maneras, especialmente cuando se usan funciones que operan en colecciones de datos. Es el concepto más básico (y importante) de la programación funcional.
- **Expresión** : una expresión es una construcción de código que genera un valor. En F#, este valor debe estar enlazado o omitirse explícitamente. Una expresión se puede reemplazar trivialmente por una llamada de función.
- **La pureza-pureza** es una propiedad de una función de modo que el valor devuelto sea siempre el mismo para los mismos argumentos, y que su evaluación no tenga efectos secundarios. Una función pura depende completamente de sus argumentos.
- **Transparencia referencial** : la transparencia referencial es una propiedad de expresiones de modo que se pueden reemplazar por su salida sin afectar al comportamiento de un programa.
- **Inmutabilidad** : la inmutabilidad significa que un valor no se puede cambiar en contexto. Esto contrasta con las variables, que pueden cambiar en su lugar.

Ejemplos

En los siguientes ejemplos se muestran estos conceptos básicos.

Funciones

La construcción más común y fundamental en la programación funcional es la función. Esta es una función simple que suma 1 a un entero:

```
let addOne x = x + 1
```

Su signatura de tipo es la siguiente:

```
val addOne: x:int -> int
```

La firma se puede leer como "`addOne` acepta un `int` denominado `x` y generará una `int`". Más formalmente, `addOne` está *asignando* un valor del conjunto de enteros al conjunto de enteros. El token `->` significa esta asignación. En F#, normalmente puede ver la firma de la función para obtener una idea de lo que hace.

Por lo tanto, ¿por qué es importante la firma? En la programación funcional con tipo, la implementación de una función suele ser menos importante que la firma de tipo real. El hecho de que `addOne` agregue el valor 1 a un entero es interesante en tiempo de ejecución, pero cuando se crea un programa, el hecho de que acepte y devuelva un `int` es lo que informa de cómo se utilizará realmente esta función. Además, una vez que use esta función correctamente (con respecto a su firma de tipo), el diagnóstico de cualquier problema solo puede realizarse dentro del cuerpo de la función `addOne`. Este es el estímulo detrás de la programación funcional con tipo.

Expresiones

Las expresiones son construcciones que se evalúan como un valor. A diferencia de las instrucciones, que realizan una acción, se pueden considerar expresiones al realizar una acción que devuelve un valor. Las expresiones casi siempre se usan en favor de instrucciones en la programación funcional.

Considere la función anterior, `addOne`. El cuerpo de `addOne` es una expresión:

```
// 'x + 1' is an expression!  
let addOne x = x + 1
```

Es el resultado de esta expresión que define el tipo de resultado de la función `addOne`. Por ejemplo, la expresión que constituye esta función podría cambiarse para ser un tipo diferente, como una `string`:

```
let addOne x = x.ToString() + "1"
```

La firma de la función es ahora:

```
val addOne: x:'a -> string
```

Dado que cualquier tipo F# de puede tener `ToString()` llama a en él, el tipo de `x` se ha convertido en genérico (denominado *generalización automática*) y el tipo resultante es una `string`.

Las expresiones no son solo los cuerpos de las funciones. Puede tener expresiones que generen un valor que se utilice en otro lugar. Una común es `if`:

```
// Checks if 'x' is odd by using the mod operator  
let isOdd x = x % 2 <> 0  
  
let addOneIfOdd input =  
    let result =  
        if isOdd input then  
            input + 1  
        else  
            input  
  
    result
```

La expresión `if` produce un valor denominado `result`. Tenga en cuenta que puede omitir `result` por completo, lo que hace que la expresión de `if` sea el cuerpo de la función `addOneIfOdd`. Lo más importante que debe recordar sobre las expresiones es que generan un valor.

Hay un tipo especial, `unit`, que se usa cuando no hay nada que devolver. Por ejemplo, considere esta función simple:

```
let printString (str: string) =  
    printfn "String is: %s" str
```

La firma tiene el siguiente aspecto:

```
val printString: str:string -> unit
```

El tipo de `unit` indica que no se devuelve ningún valor real. Esto resulta útil cuando se tiene una rutina que debe "trabajar" a pesar de no tener ningún valor para devolver como resultado de ese trabajo.

Esto está en contraste con la programación imperativa, donde la construcción de `if` equivalente es una instrucción, y la generación de valores se suele realizar con variables mutantes. Por ejemplo, en C#, el código podría escribirse de la siguiente manera:

```
bool IsOdd(int x) => x % 2 != 0;  
  
int AddOneIfOdd(int input)  
{  
    var result = input;  
  
    if (IsOdd(input))  
    {  
        result = input + 1;  
    }  
  
    return result;  
}
```

Merece la pena mencionar que C# y otros lenguajes de estilo C admiten la [expresión ternaria](#), que permite la programación condicional basada en expresiones.

En la programación funcional, es poco habitual mutar los valores con las instrucciones. Aunque algunos lenguajes funcionales admiten instrucciones y mutación, no es habitual utilizar estos conceptos en la programación funcional.

Funciones puras

Como se mencionó anteriormente, las funciones puras son funciones que:

- Siempre se evalúan con el mismo valor para la misma entrada.
- No tener efectos secundarios.

Resulta útil pensar en las funciones matemáticas en este contexto. En las matemáticas, las funciones dependen solo de sus argumentos y no tienen efectos secundarios. En la función matemática $f(x) = x + 1$, el valor de $f(x)$ depende solo del valor de x . Las funciones puras en la programación funcional son la misma manera.

Al escribir una función pura, la función debe depender solo de sus argumentos y no realizar ninguna acción que tenga como resultado un efecto secundario.

Este es un ejemplo de una función no pura porque depende del estado global y mutable:

```
let mutable value = 1  
  
let addOneToValue x = x + value
```

La función `addOneToValue` es claramente impura, porque `value` podría cambiarse en cualquier momento para tener un valor distinto de 1. Este patrón de función de un valor global se debe evitar en la programación

funcional.

Este es otro ejemplo de una función no pura, ya que realiza un efecto secundario:

```
let addOneToValue x =  
    printfn "x is %d" x  
    x + 1
```

Aunque esta función no depende de un valor global, escribe el valor de `x` en la salida del programa. Aunque no hay nada inherentemente incorrecto, esto significa que la función no es pura. Si otra parte del programa depende de algo externo al programa, como el búfer de salida, llamar a esta función puede afectar a la otra parte del programa.

La eliminación de la instrucción `printfn` hace que la función sea pura:

```
let addOneToValue x = x + 1
```

Aunque esta función no es intrínsecamente *mejor* que la versión anterior con la instrucción `printfn`, garantiza que toda esta función devuelve un valor. Llamar a esta función cualquier número de veces produce el mismo resultado: simplemente genera un valor. La capacidad de previsión proporcionada por la pureza es algo que los programadores funcionales están procurando.

Inmutabilidad

Por último, uno de los conceptos fundamentales de la programación funcional con tipo es la inmutabilidad. En F#, todos los valores son inmutables de forma predeterminada. Esto significa que no se pueden mutar en contexto a menos que los marque explícitamente como mutables.

En la práctica, el trabajo con valores inmutables significa que se cambia el enfoque a la programación de, "es necesario cambiar algo", a "es necesario generar un nuevo valor".

Por ejemplo, si se agrega 1 a un valor, se genera un nuevo valor, no se modifica el existente:

```
let value = 1  
let secondValue = value + 1
```

En F#, el código siguiente **no** muta la función `value`; en su lugar, realiza una comprobación de igualdad:

```
let value = 1  
value = value + 1 // Produce a 'bool' value!
```

Algunos lenguajes de programación funcionales no admiten la mutación. En F#, es compatible, pero no es el comportamiento predeterminado de los valores de.

Este concepto se extiende aún más a las estructuras de datos. En la programación funcional, las estructuras de datos inmutables como los conjuntos (y muchos más) tienen una implementación diferente de la que cabría esperar inicialmente. Conceptualmente, algo como agregar un elemento a un conjunto no cambia el conjunto, genera un *nuevo* conjunto con el valor agregado. En segundo plano, esto se logra con una estructura de datos diferente que permite realizar un seguimiento eficaz de un valor para que se pueda proporcionar como resultado la representación adecuada de los datos.

Este estilo de trabajo con valores y estructuras de datos es fundamental, ya que le obliga a tratar cualquier operación que modifique algo como si creara una nueva versión de ese elemento. Esto permite que elementos como la igualdad y la comparación sean coherentes en los programas.

Pasos siguientes

En la siguiente sección, se tratarán detalladamente las funciones y se explorarán las distintas formas en que se pueden usar en la programación funcional.

[Las funciones de primera clase](#) exploran las funciones en profundidad, lo que muestra cómo se pueden usar en varios contextos.

Información adicional

La serie [funcionalmente](#) es otro recurso fantástico para obtener información sobre la programación funcional F# con. Trata aspectos básicos de la programación funcional de forma pragmática y fácil de leer, mediante el uso F# de características para ilustrar los conceptos.

Funciones de primera clase

23/10/2019 • 31 minutes to read • [Edit Online](#)

Los lenguajes de programación funcional se caracterizan principalmente por tratar las funciones como valores de primera clase. El usuario podrá hacer con las funciones todo lo que se puede hacer con los valores de los otros tipos integrados y, además, con un esfuerzo comparable.

Entre los criterios más comunes para determinar si los valores son de primera clase se encuentran los siguientes:

- ¿Puede enlazar funciones a los identificadores? Es decir, ¿puede asignarles nombres?
- ¿Se pueden almacenar funciones en estructuras de datos, como en una lista?
- ¿Se puede pasar una función como argumento en una llamada de función?
- ¿Se puede devolver una función desde una llamada de función?

Las dos últimas medidas definen lo que se conoce como *operaciones de orden superior* o *funciones de orden superior*. Las funciones de orden superior aceptan otras funciones como argumentos y devuelven funciones como valores de llamadas de función. Estas operaciones son compatibles con los principales pilares de la programación funcional, a saber, las funciones de asignación y la composición de funciones.

Asignar un nombre al valor

Si una función es un valor de primera clase, debe ser posible asignarle un nombre al igual que en el caso de enteros, cadenas y otros tipos integrados. En la programación funcional, esto se denomina "enlazar un identificador a un valor". F# `let <identifier> = <value>` utiliza `let` enlaces para enlazar nombres a valores. En el código siguiente, se muestran dos ejemplos:

```
// Integer and string.  
let num = 10  
let str = "F#"
```

La asignación de un nombre a una función es así de sencilla. En el ejemplo siguiente se define una función denominada `squareIt` enlazando el identificador `squareIt` a la [expresión lambda](#) `fun n -> n * n`. La función `squareIt` tiene un parámetro, `n`, y devuelve el cuadrado de ese parámetro.

```
let squareIt = fun n -> n * n
```

F# proporciona la siguiente sintaxis más concisa para lograr el mismo resultado.

```
let squareIt2 n = n * n
```

En los siguientes ejemplos, se usa principalmente el primer estilo, `let <function-name> = <lambda-expression>`, para recalcar las similitudes entre la declaración de funciones y la declaración de otros tipos de valores. Sin embargo, todas las funciones con nombre también se pueden escribir mediante la sintaxis concisa. Algunos de los ejemplos se han escrito de ambas formas.

Almacenar el valor en una estructura de datos

Un valor de primera clase puede almacenarse en una estructura de datos. En el siguiente código, se muestran ejemplos en los se almacenan los valores en listas y tuplas.

```
// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]

// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )
```

Para comprobar que un nombre de función almacenado en una tupla se evalúa realmente como una función, en el siguiente ejemplo se usan los operadores `fst` y `snd` para extraer los dos primeros elementos de la tupla `funAndArgTuple`. El primer elemento de la tupla es `squareIt` y el segundo elemento es `num`. En un ejemplo anterior, se ha enlazado el identificador `num` al entero 10, un argumento válido para la función `squareIt`. La segunda expresión aplica el primer elemento de la tupla al segundo elemento de la tupla: `squareIt num`.

```
// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))
```

De forma similar, al igual que en el caso del identificador `num` y el entero 10, el identificador `squareIt` y la expresión lambda `fun n -> n * n` puede usarse indistintamente.

```
// Make a tuple of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))
```

Pasar el valor como un argumento

Si un lenguaje trata un valor como un valor de primera clase, se puede pasar dicho valor como argumento de una función. Por ejemplo, los enteros y cadenas se suelen pasar como argumentos. En el siguiente código, se muestran enteros y cadenas que se pasan como argumentos en F#.

```
// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)
```

Si una función es un valor de primera clase, dicha función también debe poder pasarse como un argumento. Recuerde que esta es la primera característica de las funciones de orden superior.

En el siguiente ejemplo, la función `applyIt` tiene dos parámetros, `op` y `arg`. Si envía una función con un parámetro a `op` y un argumento apropiado para la función a `arg`, la función devolverá el resultado de aplicar `op` a `arg`. En el siguiente ejemplo, el argumento de función y el argumento entero se envían de la misma manera: por su nombre.

```
// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)
```

La capacidad para enviar una función como un argumento a otra función subyace a las abstracciones comunes en los lenguajes de programación funcional, como las operaciones de asignación o de filtrado. Por ejemplo, una operación de asignación es una función de orden superior que captura el cálculo compartido por funciones que recorren una lista, realizan alguna operación con cada elemento y, a continuación, devuelven una lista con los resultados. Quizás se desee incrementar cada elemento de una lista de enteros, elevar cada elemento al cuadrado o cambiar a mayúsculas cada uno de los elementos de una lista de cadenas. La parte del cálculo propensa a

errores es el proceso recursivo que recorre la lista y compila una lista de los resultados que se van a devolver. Dicha parte se captura en la función de asignación. Lo único que hay que escribir para una aplicación concreta es la función que se desea aplicar a cada uno de los elementos de la lista (sumar, elevar al cuadrado, cambiar mayúscula a minúscula o viceversa). Dicha función se envía como argumento a la función de asignación, de la misma manera que se envía `squareIt` a `applyIt` en el ejemplo anterior.

F#proporciona métodos de mapa para la mayoría de los tipos de colección, incluidas [listas](#), [matrices](#) y [secuencias](#). En los siguientes ejemplos, se utilizan listas. La sintaxis es `List.map <the function> <the list>`.

```
// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot
```

Para obtener más información, vea [listas](#).

Devolver el valor de una llamada de función

Por último, si un lenguaje trata una función como valor de primera clase, dicha función debe poder devolverse como valor de una llamada de función, al igual que en el caso de otros tipos como enteros y cadenas.

Las siguientes llamadas de función devuelven enteros y los muestran.

```
// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)
```

La siguiente llamada de función devuelve una cadena.

```
// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()
```

La siguiente llamada de función, declarada en el propio código, devuelve un valor booleano. El valor mostrado es `True`.

```
System.Console.WriteLine((fun n -> n % 2 = 1) 15)
```

La capacidad para devolver una función como valor de una llamada de función es la segunda característica de las funciones de orden superior. En el siguiente ejemplo, `checkFor` se define como una función que toma un argumento, `item`, y devuelve una nueva función como su valor. La función devuelta toma una lista como

argumento, `lst`, y busca `item` en `lst`. Si encuentra `item`, la función devuelve `true`. Si no encuentra `item`, la función devuelve `false`. Como en la sección anterior, el código siguiente usa una función de lista proporcionada [List.exists](#) para buscar en la lista.

```
let checkFor item =  
    let functionToReturn = fun lst ->  
        List.exists (fun a -> a = item) lst  
    functionToReturn
```

En el siguiente código, se utiliza `checkFor` para crear una función que toma un argumento (una lista) y busca el número 7 en la lista.

```
// integerList and stringList were defined earlier.  
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]  
//let stringList = [ "one"; "two"; "three" ]  
  
// The returned function is given the name checkFor7.  
let checkFor7 = checkFor 7  
  
// The result displayed when checkFor7 is applied to integerList is True.  
System.Console.WriteLine(checkFor7 integerList)  
  
// The following code repeats the process for "seven" in stringList.  
let checkForSeven = checkFor "seven"  
  
// The result displayed is False.  
System.Console.WriteLine(checkForSeven stringList)
```

En el siguiente ejemplo, se utiliza el estatus de primera clase de las funciones en F# para declarar una función, `compose`, que devuelve una composición de dos argumentos de función.

```
// Function compose takes two arguments. Each argument is a function  
// that takes one argument of the same type. The following declaration  
// uses lambda expression syntax.  
let compose =  
    fun op1 op2 ->  
        fun n ->  
            op1 (op2 n)  
  
// To clarify what you are returning, use a nested let expression:  
let compose2 =  
    fun op1 op2 ->  
        // Use a let expression to build the function that will be returned.  
        let funToReturn = fun n ->  
            op1 (op2 n)  
        // Then just return it.  
        funToReturn  
  
// Or, integrating the more concise syntax:  
let compose3 op1 op2 =  
    let funToReturn = fun n ->  
        op1 (op2 n)  
    funToReturn
```

NOTE

Para obtener una versión más corta, vea la siguiente sección, "Funciones currificadas".

En el siguiente código, se envían dos funciones como argumentos a `compose` y las dos toman un solo argumento

del mismo tipo. El valor devuelto es una nueva función que es una composición de los dos argumentos de función.

```
// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)
```

NOTE

F# proporciona dos operadores, `<<` y `>>`, que realizan la composición de funciones. Por ejemplo,

`let squareAndDouble2 = doubleIt << squareIt` equivale a `let squareAndDouble = compose doubleIt squareIt` del ejemplo anterior.

En el siguiente ejemplo de cómo devolver una función como valor de una llamada de función, se crea un simple juego de adivinanzas. Para crear un juego, llame a `makeGame` y envíe para `target` el valor que el usuario debe adivinar. El valor devuelto de la función `makeGame` es una función que toma un argumento (la adivinanza) y notifica si el usuario ha respondido correctamente a la adivinanza.

```
let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
        if guess = target then
            System.Console.WriteLine("You win!")
        else
            System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game
```

El siguiente código llama a `makeGame`, enviando el valor `7` para `target`. El identificador `playGame` está enlazado a la expresión lambda devuelta. Por consiguiente, `playGame` es una función que toma como único argumento un valor de `guess`.


```

let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!

```

Funciones currificadas

Muchos de los ejemplos de la sección anterior se pueden escribir de forma más concisa aprovechando las ventajas de la currficación F# implícita en las declaraciones de función. La currficación es un proceso que consiste en transformar una función con varios parámetros en una serie de funciones incrustadas, cada una de las cuales tiene un solo parámetro. En F#, las funciones con más de un parámetro se currfican de manera inherente. Por ejemplo, la función `compose` que aparece en la sección anterior se puede escribir de manera concisa con tres parámetros, tal y como se indica a continuación.

```
let compose4 op1 op2 n = op1 (op2 n)
```

Sin embargo, el resultado es una función con un parámetro que devuelve una función con un parámetro que, a su vez, devuelve otra función con un parámetro, tal y como se muestra en `compose4curried`.

```

let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)

```

El acceso a esta función puede realizarse de varias maneras. En cada uno de los siguientes ejemplos, se devuelve y se muestra el número 18. Se puede reemplazar `compose4` con `compose4curried` en cualquiera de los ejemplos.

```

// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)

```

Para comprobar que la función se ejecuta igual que antes, recurre de nuevo a los casos de prueba originales.

```
let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)
```

NOTE

La currificación se puede restringir agrupando los parámetros en tuplas. Para obtener más información, vea "patrones de parámetro" en [parámetros y argumentos](#).

En el siguiente ejemplo, se utiliza la currificación implícita para escribir una versión más corta de `makeGame`. Los detalles referentes a cómo `makeGame` construye y devuelve la función `game` son menos explícitos en este formato, pero se pueden usar los casos de prueba originales para comprobar que el resultado es el mismo.

```
let makeGame2 target guess =
    if guess = target then
        System.Console.WriteLine("You win!")
    else
        System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'
```

Para obtener más información sobre currificación, vea "aplicación parcial de argumentos" en [funciones](#).

El identificador y la definición de función pueden usarse indistintamente

El nombre de variable `num` de los ejemplos anteriores se evalúa como el entero 10, y no es de extrañar que el entero 10 sea también válido en los casos en los que `num` es válido. Lo mismo se aplica a los identificadores de las funciones y sus valores: siempre que se pueda usar el nombre de la función, se podrá usar la expresión lambda enlazada al mismo.

En el siguiente ejemplo, se define una función `Boolean` denominada `isNegative` y, a continuación, se usan indistintamente el nombre y la definición de la función. En los tres ejemplos siguientes, se devuelve y se muestra `False`.

```
let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)
```

Para ir incluso un poco más lejos, reemplace `applyIt` por el valor al que está enlazada la función `applyIt`.

```
System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0) 10)
```

Las funciones son valores de primera clase en F#

En los ejemplos que figuran en las secciones anteriores, se muestra que las funciones en F# cumplen los criterios de valores de primera clase:

- Se puede enlazar un identificador a una definición de función.

```
let squareIt = fun n -> n * n
```

- Se puede almacenar una función en una estructura de datos.

```
let funTuple2 = ( BMICalculator, fun n -> n * n )
```

- Se puede pasar una función como argumento.

```
let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]
```

- Se puede devolver una función como valor de una llamada de función.

```
let checkFor item =
    let functionToReturn = fun lst ->
        List.exists (fun a -> a = item) lst
    functionToReturn
```

Para obtener más información F#acerca de, consulte la [F# referencia del lenguaje](#).

Ejemplo

DESCRIPCIÓN

El código siguiente contiene todos los ejemplos de este tema.

Código

```
// ** GIVE THE VALUE A NAME **

// Integer and string.
let num = 10
let str = "F#"
```

```

let squareIt = fun n -> n * n

let squareIt2 n = n * n

// ** STORE THE VALUE IN A DATA STRUCTURE **

// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]

// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )

// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100

```

```

// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))

// Make a list of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))

// ** PASS THE VALUE AS AN ARGUMENT **

// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)

// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)

// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot

```

```

// ** RETURN THE VALUE FROM A FUNCTION CALL **

// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)

// The following function call returns a string:

// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()

System.Console.WriteLine((fun n -> n % 2 = 1) 15)

let checkFor item =
    let functionToReturn = fun lst ->
        List.exists (fun a -> a = item) lst
    functionToReturn

// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)

// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expression syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
            op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
        op1 (op2 n)
    funToReturn

```

```

// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)

let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
        if guess = target then
            System.Console.WriteLine("You win!")
        else
            System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game

let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!

// ** CURRIED FUNCTIONS **

let compose4 op1 op2 n = op1 (op2 n)

let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)

// Access one layer at a time.

```

```

System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)


let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)


let makeGame2 target guess =
    if guess = target then
        System.Console.WriteLine("You win!")
    else
        System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'


// ** IDENTIFIER AND FUNCTION DEFINITION ARE INTERCHANGEABLE **


let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)


System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0) 10)


// ** FUNCTIONS ARE FIRST-CLASS VALUES IN F# **

//let squareIt = fun n -> n * n

let funTuple2 = ( BMICalculator, fun n -> n * n )

```



```
let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]

//let checkFor item =
//    let functionToReturn = fun lst ->
//        List.exists (fun a -> a = item) lst
//    functionToReturn
```

Vea también

- [Listas](#)
- [Tuplas](#)
- [Funciones](#)
- [let](#) [Enlaces](#)
- Expresiones lambda: [fun](#) [Palabra clave](#)

Programación asíncrona en F#

22/04/2020 • 24 minutes to read • [Edit Online](#)

La programación asíncrona es un mecanismo que es esencial para las aplicaciones modernas por diversas razones. Hay dos casos de uso principales que la mayoría de los desarrolladores encontrarán:

- Presentar un proceso de servidor que puede dar servicio a un número significativo de solicitudes entrantes simultáneas, al tiempo que se minimizan los recursos del sistema ocupados mientras el procesamiento de solicitudes espera entradas de sistemas o servicios externos a ese proceso
- Mantener una interfaz de usuario responsiva o un subproceso principal mientras se progresa simultáneamente en el trabajo en segundo plano

Aunque el trabajo en segundo plano a menudo implica la utilización de varios subprocesos, es importante tener en cuenta los conceptos de asincronía y multiproceso por separado. De hecho, son preocupaciones separadas, y una no implica la otra. En este artículo se describen los conceptos separados con más detalle.

Asincronía definida

El punto anterior - que la asincronía es independiente de la utilización de varios subprocesos - vale la pena explicar un poco más. Hay tres conceptos que a veces están relacionados, pero estrictamente independientes entre sí:

- Simultaneidad; cuando se ejecutan varios cálculos en períodos de tiempo superpuestos.
- Paralelismo; cuando varios cálculos o varias partes de un solo cálculo se ejecutan exactamente al mismo tiempo.
- Asincronía; cuando uno o más cálculos se pueden ejecutar por separado del flujo principal del programa.

Los tres son conceptos ortogonales, pero se pueden confundir fácilmente, especialmente cuando se utilizan juntos. Por ejemplo, es posible que deba ejecutar varios cálculos asíncronos en paralelo. Esta relación no significa que el paralelismo o la asincronía impliquen unos a otros.

Si considera la etimología de la palabra "asincrónico", hay dos piezas involucradas:

- "a", que significa "no".
- "sincrónico", que significa "al mismo tiempo".

Cuando se juntan estos dos términos, verá que "asincrónico" significa "no al mismo tiempo". Eso es todo. No hay ninguna implicación de simultaneidad o paralelismo en esta definición. Esto también es cierto en la práctica.

En términos prácticos, los cálculos asíncronos en F# están programados para ejecutarse independientemente del flujo principal del programa. Esta ejecución independiente no implica simultaneidad o paralelismo, ni implica que siempre se produce un cálculo en segundo plano. De hecho, los cálculos asíncronos pueden incluso ejecutarse sincrónicamente, dependiendo de la naturaleza del cálculo y el entorno en el que se ejecuta el cálculo.

La principal toma que debe tener es que los cálculos asíncronos son independientes del flujo del programa principal. Aunque hay pocas garantías sobre cuándo o cómo se ejecuta un cálculo asíncrono, hay algunos enfoques para orquestarlos y programarlos. En el resto de este artículo se exploran los conceptos básicos de la asincronía de F# y cómo usar los tipos, funciones y expresiones integradas en F#.

Conceptos principales

En F#, la programación asíncrona se centra en tres conceptos básicos:

- El `Async<'T>` tipo, que representa un cálculo asíncrono componible.

- Las `Async` funciones del módulo, que permiten programar el trabajo asíncrono, componer cálculos asíncronos y transformar resultados asíncronos.
- La `async { }` [expresión de cálculo](#), que proporciona una sintaxis conveniente para crear y controlar cálculos asíncronos.

Puede ver estos tres conceptos en el ejemplo siguiente:

```
open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn "File %s has %d bytes" fileName bytes.Length
    }

[<EntryPoint>]
let main argv =
    printTotalFileBytes "path-to-file.txt"
    |> Async.RunSynchronously

    Console.Read() |> ignore
    0
```

En el ejemplo, la `printTotalFileBytes` `string -> Async<unit>` función es de tipo `Async<unit>`. Llamar a la función no ejecuta realmente el cálculo asíncrono. En su lugar, devuelve un `Async<unit>` que actúa como una *especificación* del trabajo que se ejecuta de forma asíncrona. Llama `Async.AwaitTask` a su cuerpo, que convierte `ReadAllBytesAsync` el resultado de un tipo adecuado.

Otra línea importante es `Async.RunSynchronously` la llamada a `Async.AwaitTask`. Esta es una de las funciones de inicio del módulo `async` a las que tendrá que llamar si desea ejecutar realmente un cálculo asíncrono de F.

Esta es una diferencia fundamental con el `async` estilo de programación de C/Visual Basic. En F, los cálculos asíncronos se pueden considerar como **tareas frías**. Deben iniciarse explícitamente para ejecutarse. Esto tiene algunas ventajas, ya que le permite combinar y secuenciar el trabajo asíncrono mucho más fácilmente que en C o Visual Basic.

Combinar cálculos asíncronos

Este es un ejemplo que se basa en el anterior mediante la combinación de cálculos:

```

open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn "File %s has %d bytes" fileName bytes.Length
    }

[<EntryPoint>]
let main argv =
    argv
    |> Array.map printTotalFileBytes
    |> Async.Parallel
    |> Async.Ignore
    |> Async.RunSynchronously

0

```

Como puede ver, `main` la función tiene bastantes llamadas más realizadas. Conceptualmente, hace lo siguiente:

1. Transforme los argumentos `Async<unit>` de línea `Array.map` de comandos en cálculos con `.`
2. Cree `Async<'T[]>` un que programe `printTotalFileBytes` y ejecute los cálculos en paralelo cuando se ejecute.
3. Cree `Async<unit>` un que ejecute el cálculo paralelo e ignore su resultado.
4. Ejecute explícitamente el `Async.RunSynchronously` último cálculo con y bloquee hasta que se complete.

Cuando se ejecuta `printTotalFileBytes` este programa, se ejecuta en paralelo para cada argumento de línea de comandos. Dado que los cálculos asincrónicos se ejecutan independientemente del flujo del programa, no hay ningún orden en el que impriman su información y terminen de ejecutarse. Los cálculos se programarán en paralelo, pero su orden de ejecución no está garantizado.

Secuenciar cálculos asincrónicos

Dado `Async<'T>` que es una especificación de trabajo en lugar de una tarea que ya se está ejecutando, puede realizar transformaciones más complejas fácilmente. Este es un ejemplo que secuencia un conjunto de cálculos asincrónicos para que se ejecuten uno tras otro.

```

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn "File %s has %d bytes" fileName bytes.Length
    }

[<EntryPoint>]
let main argv =
    argv
    |> Array.map printTotalFileBytes
    |> Async.Sequential
    |> Async.Ignore
    |> Async.RunSynchronously
    |> ignore

```

Esto se `printTotalFileBytes` programará para ejecutar en `argv` el orden de los elementos de en lugar de programarlos en paralelo. Dado que el siguiente elemento no se programará hasta que el último cálculo haya terminado de ejecutarse, los cálculos se secuencian de forma que no haya superposición en su ejecución.

Funciones importantes del módulo asincrónico

Cuando se escribe código asincrónico en F, normalmente interactuará con un marco de trabajo que controla la programación de cálculos automáticamente. Sin embargo, este no siempre es el caso, por lo que es bueno aprender las diversas funciones de inicio para programar el trabajo asincrónico.

Debido a que los cálculos asincrónicos de F son una *especificación* de trabajo en lugar de una representación del trabajo que ya se está ejecutando, deben iniciarse explícitamente con una función de inicio. Hay muchas [funciones de inicio async](#) que son útiles en diferentes contextos. En la siguiente sección se describen algunas de las funciones de inicio más comunes.

Async.StartChild

Inicia un cálculo secundario dentro de un cálculo asincrónico. Esto permite que varios cálculos asincrónicos se ejecuten simultáneamente. El cálculo secundario comparte un token de cancelación con el cálculo primario. Si se cancela el cálculo primario, también se cancela el cálculo secundario.

Signature:

```
computation: Async<'T> * timeout: ?int -> Async<Async<'T>>
```

Cuándo usarlo:

- Si desea ejecutar varios cálculos asincrónicos simultáneamente en lugar de uno a la vez, pero no tenerlos programados en paralelo.
- Cuando desea vincular la duración de un cálculo secundario a la de un cálculo primario.

Qué tener en cuenta:

- Iniciar varios cálculos con `Async.StartChild` no es lo mismo que programarlos en paralelo. Si desea programar cálculos en `Async.Parallel` paralelo, utilice .
- La cancelación de un cálculo primario desencadenará la cancelación de todos los cálculos secundarios que inició.

Async.StartImmediate

Ejecuta un cálculo asincrónico y comienza inmediatamente en el subproceso actual del sistema operativo. Esto es útil si necesita actualizar algo en el subproceso que realiza la llamada durante el cálculo. Por ejemplo, si un cálculo asincrónico debe actualizar una `Async.StartImmediate` interfaz de usuario (por ejemplo, actualizar una barra de progreso), debe usarse.

Signature:

```
computation: Async<unit> * cancellationToken: ?Cancellation.Token -> unit
```

Cuándo usarlo:

- Cuando necesite actualizar algo en el subproceso que realiza la llamada en medio de un cálculo asincrónico.

Qué tener en cuenta:

- El código del cálculo asincrónico se ejecutará en cualquier subproceso en el que se programe. Esto puede ser problemático si ese subproceso es de alguna manera sensible, como un subproceso de interfaz de usuario. En tales `Async.StartImmediate` casos, es probable que no sea apropiado de usar.

Async.StartAsTask

Ejecuta un cálculo en el grupo de subprocesos. Devuelve `Task<TResult>` un que se completará en el estado correspondiente una vez que finalice el cálculo (produce el resultado, produce una excepción o se cancela). Si no se

proporciona ningún token de cancelación, se usa el token de cancelación predeterminado.

Signature:

```
computation: Async<'T> * taskCreationOptions: ?TaskCreationOptions * cancellationToken: ?CancellationToken -> Task<'T>
```

Cuándo usarlo:

- Cuando necesite llamar a una API de `Task<TResult>` .NET que espera que represente el resultado de un cálculo asíncronico.

Qué tener en cuenta:

- Esta llamada asignará `Task` un objeto adicional, que puede aumentar la sobrecarga si se utiliza con frecuencia.

Async.Parallel

Programa una secuencia de cálculos asíncronos que se ejecutarán en paralelo. El grado de paralelismo se puede ajustar/limitar `maxDegreesOfParallelism` opcionalmente especificando el parámetro.

Signature:

```
computations: seq<Async<'T>> * ?maxDegreesOfParallelism: int -> Async<'T[]>
```

Cuándo usarlo

- Si necesita ejecutar un conjunto de cálculos al mismo tiempo y no depende de su orden de ejecución.
- Si no necesita resultados de cálculos programados en paralelo hasta que se hayan completado todos.

Qué tener en cuenta:

- Solo puede acceder a la matriz resultante de valores una vez que todos los cálculos han finalizado.
- Los cálculos se ejecutarán siempre que terminen programados. Este comportamiento significa que no puede confiar en su orden de ejecución.

Async.Sequential

Programa una secuencia de cálculos asíncronos que se ejecutarán en el orden en que se pasan. Se ejecutará el primer cálculo, luego el siguiente, y así sucesivamente. No se ejecutarán cálculos en paralelo.

Signature:

```
computations: seq<Async<'T>> -> Async<'T[]>
```

Cuándo usarlo

- Si necesita ejecutar varios cálculos en orden.

Qué tener en cuenta:

- Solo puede acceder a la matriz resultante de valores una vez que todos los cálculos han finalizado.
- Los cálculos se ejecutarán en el orden en que se pasan a esta función, lo que puede significar que transcurrirá más tiempo antes de que se devuelvan los resultados.

Async.AwaitTask

Devuelve un cálculo asíncrono `Task<TResult>` que espera a que se complete el dado y devuelve su resultado como un `Async<'T>`

Signature:

```
task: Task<'T> -> Async<'T>
```

Cuándo usarlo:

- Cuando se consume una API de [Task<TResult>](#) .NET que devuelve un cálculo asincrónico dentro de F .

Qué tener en cuenta:

- Las excepciones se [AggregateException](#) ajustan siguiendo la convención de la biblioteca de tareas paralelas y este comportamiento es diferente de cómo se asincrónico de F - generalmente expone excepciones.

Async.Catch

Crea un cálculo asincrónico `Async<'T>` que ejecuta `Async<Choice<'T, exn>>` un determinado , devolviendo un archivo . Si el `Async<'T>` especificado se completa `Choice1of2` correctamente, se devuelve a con el valor resultante. Si se produce una excepción antes `Choice2of2` de que se complete, se devuelve a a con la excepción provocada. Si se utiliza en un cálculo asincrónico que se compone de muchos cálculos y uno de esos cálculos produce una excepción, el cálculo que abarca se detendrá por completo.

Signature:

```
computation: Async<'T> -> Async<Choice<'T, exn>>
```

Cuándo usarlo:

- Al realizar un trabajo asincrónico que puede producir un error con una excepción y desea controlar esa excepción en el llamador.

Qué tener en cuenta:

- Cuando se utilizan cálculos asincrónicos combinados o secuenciados, el cálculo que abarca se detendrá por completo si uno de sus cálculos "internos" produce una excepción.

Async.Ignore

Crea un cálculo asincrónico que ejecuta el cálculo especificado y omite su resultado.

Signature:

```
computation: Async<'T> -> Async<unit>
```

Cuándo usarlo:

- Cuando tiene un cálculo asincrónico cuyo resultado no es necesario. Esto es análogo `ignore` al código para el código no asincrónico.

Qué tener en cuenta:

- Si debe `Async.Ignore` utilizar porque desea `Async.Start` utilizar u otra `Async<unit>` función que requiera, considere si descartar el resultado está bien. Evite descartar los resultados solo para ajustarse a una firma de tipo.

Async.RunSynchronously

Ejecuta un cálculo asincrónico y espera su resultado en el subproceso que realiza la llamada. Esta llamada está bloqueando.

Signature:

```
computation: Async<'T> * timeout: ?int * cancellationToken: ?CancellationTokén -> 'T
```

Cuándo usarlo

- Si lo necesita, úselo solo una vez en una aplicación - en el punto de entrada para un ejecutable.
- Cuando no le importa el rendimiento y desea ejecutar un conjunto de otras operaciones asincrónicas a la vez.

Qué tener en cuenta:

- Al `Async.RunSynchronously` llamar, se bloquea el subproceso que realiza la llamada hasta que se completa la ejecución.

Async.Start

Inicia un cálculo asincrónico `unit` en el grupo de subprocesos que devuelve. No espera su resultado. Los cálculos `Async.Start` anidados iniciados con se inician independientemente del cálculo primario que los llamó. Su vida útil no está vinculada a ningún cálculo principal. Si se cancela el cálculo primario, no se cancela ningún cálculo secundario.

Signature:

```
computation: Async<unit> * cancellationToken: ?CancellationTokén -> unit
```

Utilícelo solo cuando:

- Tiene un cálculo asincrónico que no produce un resultado y/o requiere el procesamiento de uno.
- No es necesario saber cuándo se completa un cálculo asincrónico.
- No le importa en qué subproceso se ejecute un cálculo asincrónico.
- No es necesario tener en cuenta o notificar excepciones resultantes de la tarea.

Qué tener en cuenta:

- Las excepciones provocadas por `Async.Start` los cálculos iniciados con no se propagan al autor de la llamada. La pila de llamadas será completamente desenrollada.
- Cualquier trabajo (como `printfn` la `Async.Start` llamada) iniciado con no hará que el efecto ocurra en el subproceso principal de la ejecución de un programa.

Interoperar con .NET

Es posible que esté trabajando con una biblioteca de .NET o un código base de código que use programación asincrónica [async/await](#)-style. Debido a que la mayoría de las [Task<TResult>](#) [Task](#) bibliotecas de .NET usan `Async<'T>` los tipos y como sus abstracciones principales en lugar de `Task`, debe cruzar un límite entre estos dos enfoques de asincronía.

Cómo trabajar con .NET async y `Task<T>`

Trabajar con bibliotecas asincrónicas de [Task<TResult>](#) .NET y bases de código que usan (es decir, cálculos asincrónicos que tienen valores devueltos) es sencillo y tiene compatibilidad integrada con F#.

Puede utilizar `Async.AwaitTask` la función para esperar un cálculo asincrónico de .NET:


```
let getValueFromLibrary param =
    async {
        let! value = DotNetLibrary.GetValueAsync param |> Async.AwaitTask
        return value
    }
```

Puede utilizar `Async.StartAsTask` la función para pasar un cálculo asíncronico a un llamador de .NET:

```
let computationForCaller param =
    async {
        let! result = getAsyncResult param
        return result
    } |> Async.StartAsTask
```

Cómo trabajar con .NET async y `Task`

Para trabajar con las `Task` API que usan (es decir, cálculos asíncronicos de .NET que `Async<'T>` no `Task` devuelven un valor), es posible que deba agregar una función adicional que convierta un valor en :

```
module Async =
    // Async<unit> -> Task
    let startTaskFromAsyncUnit (comp: Async<unit>) =
        Async.StartAsTask comp :> Task
```

Ya hay `Async.AwaitTask` un que `Task` acepta un como entrada. Con esta y `startTaskFromAsyncUnit` la función definida `Task` anteriormente, puede iniciar y esperar tipos desde un cálculo asíncronico de F .

Relación con el multi-threading

Aunque el enhebrado se menciona a lo largo de este artículo, hay dos cosas importantes que debe recordar:

1. No hay afinidad entre un cálculo asíncronico y un subproceso, a menos que se inicie explícitamente en el subproceso actual.
2. La programación asíncronica en F no es una abstracción para multiproceso.

Por ejemplo, un cálculo puede ejecutarse realmente en el subproceso de su llamador, dependiendo de la naturaleza del trabajo. Un cálculo también podría "saltar" entre subprocesos, tomándolos prestados durante una pequeña cantidad de tiempo para realizar un trabajo útil entre períodos de "espera" (por ejemplo, cuando una llamada de red está en tránsito).

Aunque F- proporciona algunas capacidades para iniciar un cálculo asíncronico en el subproceso actual (o explícitamente no en el subproceso actual), la asincronía generalmente no está asociada a una estrategia de subprocesos determinada.

Vea también

- [El modelo de programación asíncronica de F](#)
- [Guía asincrona de F'com](#)
- [Para la diversión y la guía de programación asíncronica de beneficios](#)
- [Asíncronico en C- y F-: gotchas asíncronicos en C #](#)

Proveedores de tipos

23/07/2020 • 5 minutes to read • [Edit Online](#)

Un proveedor de tipos de F# es un componente que proporciona tipos, propiedades y métodos para usar en el programa. Los proveedores de tipos ofrecen lo que se conoce como **tipos proporcionados**, que se generan mediante el compilador de F# y se basan en un origen de datos externo.

Por ejemplo, un proveedor de tipos de F# para SQL puede generar tipos que representen tablas y columnas en una base de datos relacional. De hecho, esto es lo que hace el proveedor de tipos [SQLProvider](#).

Los tipos proporcionados dependen de los parámetros de entrada para un proveedor de tipos. Dicha entrada puede ser un origen de datos de muestra (por ejemplo, archivo de esquema JSON), una dirección URL que apunte directamente a un servicio externo o una cadena de conexión a un origen de datos. Un proveedor de tipos también permite garantizar que solo se expandan grupos de tipos a petición, es decir, que se expandan si el programa realmente hace referencia a los tipos. Esto permite la integración directa a petición de espacios de información a gran escala, tales como mercados de datos en línea de forma fuertemente tipada.

Proveedores de tipos generativos y de borrado

Hay dos clases de proveedores de tipos: generativos y de borrado.

Los proveedores de tipos generativos producen tipos que se pueden escribir como tipos de .NET en el ensamblado en el que se generan. Esto permite su consumo a partir de código en otros ensamblados. Esto significa que, en general, la representación tipada del origen de datos debe ser una factible para la representación con tipos de .NET.

Los proveedores de tipos de borrado producen tipos que solo se pueden consumir en el ensamblado o proyecto en el que se generan. Los tipos son efímeros, es decir, no se escriben en un ensamblado y no se pueden consumir mediante código en otros ensamblados. Pueden contener miembros *con retraso*, lo cual permite usar tipos proporcionados a partir de un espacio de información que sea potencialmente infinito. Son útiles para usar un subconjunto pequeño de un origen de datos grande e interconectado.

Proveedores de tipos usados habitualmente

Las siguientes bibliotecas de uso generalizado contienen proveedores de tipos para diferentes usos:

- [FSharp.Data](#) incluye proveedores de tipos para recursos y formatos de documento JSON, XML, CSV y HTML.
- [SQLProvider](#) proporciona acceso fuertemente tipado a bases de datos relacionales a través de la asignación de objetos y consultas LINQ de F# en dichos orígenes de datos.
- [FSharp.Data.SqlClient](#) tiene un conjunto de proveedores de tipos para la inserción comprobada en tiempo de compilación de T-SQL en F#.
- El [proveedor de tipos de Azure Storage](#) ofrece tipos para Azure Blobs, Tables y Queues, lo cual permite acceder a dichos recursos sin necesidad de especificar los nombres de los recursos como cadenas en todo el programa.
- [FSharp.Data.GraphQL](#) contiene **GraphQLProvider**, que proporciona tipos basados en un servidor GraphQL especificado mediante una dirección URL.

En los casos en los que sea necesario, se pueden [crear proveedores de tipos personalizados propios](#), o bien hacer referencia a proveedores de tipos creados por otros. Por ejemplo, suponga que su organización tiene un servicio de datos que proporciona un número elevado y creciente de conjuntos de datos con nombre, cada uno con su propio esquema de datos estable. Se puede elegir crear un proveedor de tipo que lea los esquemas y presente los últimos conjuntos de datos disponibles al programador de forma fuertemente tipada.

Vea también

- [Tutorial: Creación de un proveedor de tipos](#)
- [Referencia del lenguaje F#](#)

Tutorial: crear un proveedor de tipos

23/07/2020 • 65 minutes to read • [Edit Online](#)

El mecanismo de proveedores de tipos de F # es una parte importante de su compatibilidad con la programación enriquecida de información. Este tutorial le explica cómo crear proveedores de tipos, a la vez que le guía en el desarrollo de varios proveedores de tipo simples para ilustrar los conceptos básicos. Para obtener más información sobre el mecanismo de proveedores de tipos en F #, vea [proveedores de tipos](#).

El ecosistema de F # contiene una variedad de proveedores de tipo para los servicios de datos empresariales y de Internet que se usan habitualmente. Por ejemplo:

- [FSharp.Data](#) incluye proveedores de tipos para los formatos de documento JSON, XML, csv y HTML.
- [SQLProvider](#) proporciona acceso fuertemente tipado a las bases de datos SQL a través de una asignación de objetos y consultas LINQ de F # en estos orígenes de datos.
- [FSharp.Data.SqlClient](#) tiene un conjunto de proveedores de tipo para la incrustación comprobada en tiempo de compilación de T-SQL en F #.
- [FSharp.Data.TypeProviders](#) es un conjunto anterior de proveedores de tipo para su uso solo con .NET Framework programación para tener acceso a los servicios de datos SQL, Entity Framework, ODATA y WSDL.

En caso necesario, se pueden crear proveedores de tipos personalizados o se puede hacer referencia a proveedores de tipos creados por otros. Por ejemplo, una organización podría tener un servicio de datos que proporcionara un número elevado y creciente de conjuntos de datos con nombre, cada uno con su propio esquema de datos estable. Se puede crear un proveedor de tipos que lea los esquemas y presente los conjuntos de datos actuales al programador de una manera fuertemente tipada.

Antes de empezar

El mecanismo de proveedores de tipo está diseñado principalmente para insertar espacios de información de servicios y datos estables en la experiencia de programación de F#.

Este mecanismo no está diseñado para insertar espacios de información cuyo esquema cambie durante la ejecución del programa de forma relevante para la lógica del programa. El mecanismo tampoco está diseñado para la metaprogramación dentro del lenguaje, aunque ese dominio contenga algunas aplicaciones válidas. Debe utilizar este mecanismo solo en caso necesario y cuando el desarrollo de un proveedor de tipos produzca un valor muy alto.

Debe evitar escribir un proveedor de tipos cuando no hay un esquema disponible. Igualmente, debe evitar escribir un proveedor de tipo cuando una biblioteca de .NET normal (o incluso una existente) sería suficiente.

Antes de comenzar, debería hacerse las siguientes preguntas:

- ¿Tiene un esquema para su fuente de información? Si lo tiene, ¿cuál es la correspondencia entre los sistemas de tipos de F# y .NET?
- ¿Puede utilizar una API existente (dinámicamente tipada) como punto de partida para su implementación?
- ¿Usarán usted y su organización el proveedor de tipos lo suficiente como para hacer que valga la pena escribirlo? ¿Cubriría una biblioteca normal de .NET sus necesidades?
- ¿Cuánto cambiará el esquema?

- ¿Cambiará durante la escritura del código?
- ¿Cambiará entre las sesiones de escritura de código?
- ¿Cambiará durante la ejecución del programa?

Los proveedores de tipos son más adecuados en situaciones en las que el esquema es estable en runtime y durante el tiempo de vida del código compilado.

Un proveedor de tipos simple

Este ejemplo es `Samples.HelloWorldTypeProvider`, similar a los ejemplos del `examples` directorio del [SDK del proveedor de tipos de F#](#). El proveedor hace que esté disponible un "espacio de tipos" que contiene 100 tipos borrados, como muestra el código siguiente, en el que se usa la sintaxis de signatura de F# y se omiten los detalles de todos los tipos excepto `Type1`. Para obtener más información sobre los tipos borrados, consulte [detalles sobre los tipos de borrados proporcionados](#) más adelante en este tema.

```
namespace Samples.HelloWorldTypeProvider

type Type1 =
    /// This is a static property.
    static member StaticProperty : string

    /// This constructor takes no arguments.
    new : unit -> Type1

    /// This constructor takes one argument.
    new : data:string -> Type1

    /// This is an instance property.
    member InstanceProperty : int

    /// This is an instance method.
    member InstanceMethod : x:int -> char

    nested type NestedType =
        /// This is StaticProperty1 on NestedType.
        static member StaticProperty1 : string
        ...
        /// This is StaticProperty100 on NestedType.
        static member StaticProperty100 : string

type Type2 =
    ...

type Type100 =
    ...
```

Observe que el conjunto de tipos y miembros proporcionados se conoce de forma estática. Este ejemplo no aprovecha la capacidad de los proveedores para proporcionar tipos que dependen de un esquema. La implementación del proveedor de tipo se muestra en el código siguiente y sus detalles se tratan en secciones posteriores de este tema.

WARNING

Puede haber diferencias entre este código y los ejemplos en línea.

```

namespace Samples.FSharp.HelloWorldTypeProvider

open System
open System.Reflection
open ProviderImplementation.ProvidedTypes
open FSharp.Core.CompilerServices
open FSharp.Quotations

// This type defines the type provider. When compiled to a DLL, it can be added
// as a reference to an F# command-line compilation, script, or project.
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =

    // Inheriting from this type provides implementations of ITypeProvider
    // in terms of the provided types below.
    inherit TypeProviderForNamespaces(config)

    let namespaceName = "Samples.HelloWorldTypeProvider"
    let thisAssembly = Assembly.GetExecutingAssembly()

    // Make one provided type, called TypeN.
    let makeOneProvidedType (n:int) =
        ...
    // Now generate 100 types
    let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]

    // And add them to the namespace
    do this.AddNamespace(namespaceName, types)

[<assembly:TypeProviderAssembly>]
do()

```

Para usar este proveedor, abra una instancia independiente de Visual Studio, cree un script de F # y, a continuación, agregue una referencia al proveedor desde el script mediante #r como se muestra en el código siguiente:

```

#r @"..\bin\Debug\Samples.HelloWorldTypeProvider.dll"

let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")

let obj2 = Samples.HelloWorldTypeProvider.Type1("some other data")

obj1.InstanceProperty
obj2.InstanceProperty

[ for index in 0 .. obj1.InstanceProperty-1 -> obj1.InstanceMethod(index) ]
[ for index in 0 .. obj2.InstanceProperty-1 -> obj2.InstanceMethod(index) ]

let data1 = Samples.HelloWorldTypeProvider.Type1.NestedType.StaticProperty35

```

A continuación, busque los tipos en el espacio de nombres `Samples.HelloWorldTypeProvider` generado por el proveedor de tipos.

Antes de volver a compilar el proveedor, asegúrese de que se han cerrado todas las instancias de Visual Studio y de F# Interactive que están utilizando la DLL del proveedor. De lo contrario, aparecerá un error de compilación porque la DLL de salida estará bloqueada.

Para depurar este proveedor mediante instrucciones de impresión, cree un script que exponga un problema con el proveedor y, a continuación, utilice el código siguiente:

```

fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx

```

Para depurar este proveedor mediante Visual Studio, abra el Símbolo del sistema para desarrolladores de Visual Studio con credenciales administrativas y ejecute el siguiente comando:

```
devenv.exe /debugexe fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

Como alternativa, abra Visual Studio, abra el menú Depurar, elija `Debug/Attach to process...` y asocie a otro `devenv` proceso en el que esté editando el script. Con este método, le resultará más fácil centrarse en la lógica particular del proveedor de tipo escribiendo interactivamente expresiones en la segunda instancia (con IntelliSense completo y otras características).

Puede deshabilitar la depuración "Solo mi código" para identificar mejor los errores en el código generado. Para obtener información sobre cómo habilitar o deshabilitar esta característica, vea [navegar por el código con el depurador](#). Además, también puede establecer la detección de excepciones de primera oportunidad abriendo el `Debug` menú y, a continuación, eligiendo `Exceptions` las teclas `Ctrl + Alt + E` para abrir el `Exceptions` cuadro de diálogo. En ese cuadro de diálogo, en `Common Language Runtime Exceptions`, active la `Thrown` casilla.

Implementación del proveedor de tipos

En esta sección se muestran las etapas principales de la implementación del proveedor de tipos. En primer lugar, defina el tipo del proveedor de tipos personalizado:

```
[<TypeProvider>]  
type SampleTypeProvider(config: TypeProviderConfig) as this =
```

Este tipo debe ser público y debe marcarse con el atributo `TypeProvider` para que el compilador reconozca el proveedor de tipos cuando un proyecto independiente de F # haga referencia al ensamblado que contiene el tipo. El parámetro de *configuración* es opcional y, si está presente, contiene información de configuración contextual para la instancia del proveedor de tipo que crea el compilador de F #.

A continuación, implemente la interfaz `ITypeProvider`. En este caso, puede utilizar el tipo `TypeProviderForNamespaces` de la API `ProvidedTypes` como tipo base. Este tipo del asistente puede proporcionar una colección finita de espacios de nombres proporcionados anticipadamente, cada uno de los cuales contiene directamente un número finito de tipos fijos proporcionados anticipadamente. En este contexto, el proveedor genera *diligentemente* tipos incluso si no son necesarios o se usan.

```
inherit TypeProviderForNamespaces(config)
```

A continuación, defina los valores privados locales que especifican el espacio de nombres para los tipos proporcionados y busque el ensamblado propio del proveedor de tipos. Este ensamblado se utiliza más adelante como tipo primario lógico de los tipos borrados proporcionados.

```
let namespaceName = "Samples.HelloWorldTypeProvider"  
let thisAssembly = Assembly.GetExecutingAssembly()
```

A continuación, cree una función para proporcionar cada uno de los tipos `Type1... Type100`. Esta función se explica con más detalle más adelante en este tema.

```
let makeOneProvidedType (n:int) = ...
```

A continuación, genere los 100 tipos proporcionados:

```
let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]
```

Después, agregue los tipos como un espacio de nombres proporcionado:

```
do this.AddNamespace(namespaceName, types)
```

Finalmente, agregue un atributo de ensamblado que indique que está creando una DLL de proveedor de tipos:

```
[<assembly:TypeProviderAssembly>]  
do()
```

Proporcionar un tipo y sus miembros

La función `makeOneProvidedType` hace el trabajo real de proporcionar uno de los tipos.

```
let makeOneProvidedType (n:int) =  
...
```

En este paso se explica la implementación de esta función. En primer lugar, cree el tipo proporcionado (por ejemplo, `Type1`, cuando `n = 1`, o `Type57`, cuando `n = 57`).

```
// This is the provided type. It is an erased provided type and, in compiled code,  
// will appear as type 'obj'.  
let t = ProvidedTypeDefinition(thisAssembly, namespaceName,  
                               "Type" + string n,  
                               baseType = Some typeof<obj>)
```

Debe tener en cuenta los puntos siguientes:

- Este tipo proporcionado es un tipo borrado. Dado que indica que el tipo base es `obj`, las instancias aparecerán como valores de tipo `obj` en código compilado.
- Cuando especifique un tipo no anidado, deberá especificar también el ensamblado y el espacio de nombres. Para los tipos borrados, el ensamblado deberá ser el propio ensamblado del proveedor de tipos.

A continuación, agregue la documentación XML al tipo. Esta documentación se demora, es decir, se calcula a petición si el compilador host la necesita.

```
t.AddXmlDocDelayed (fun () -> sprintf "This provided type %s" ("Type" + string n))
```

A continuación, agregue una propiedad estática proporcionada al tipo:

```
let staticProp = ProvidedProperty(propertyName = "StaticProperty",  
                                   propertyType = typeof<string>,  
                                   isStatic = true,  
                                   getterCode = (fun args -> <@@ "Hello!" @@>))
```

Al obtener esta propiedad, siempre se evaluará como la cadena "Hello!". La función `GetterCode` de la propiedad utiliza una expresión de código delimitada de F# que representa el código que genera el compilador host para obtener la propiedad. Para obtener más información sobre las comillas, vea [expresiones de código delimitadas \(F#\)](#).

Agregue la documentación XML a la propiedad.


```
staticProp.AddXmlDocDelayed(fun () -> "This is a static property")
```

Ahora asocie la propiedad proporcionada al tipo proporcionado. Debe asociar un miembro proporcionado a un único tipo. De lo contrario, el miembro nunca será accesible.

```
t.AddMember staticProp
```

Ahora cree un constructor proporcionado que no reciba ningún parámetro.

```
let ctor = ProvidedConstructor(parameters = [ ],  
                               invokeCode = (fun args -> <@@ "The object data" :> obj @@>))
```

La función `InvokeCode` del constructor devuelve una expresión de código delimitada de F# que representa el código que el compilador host genera cuando se llama al constructor. Por ejemplo, puede usar el constructor siguiente:

```
new Type10()
```

Se creará una instancia del tipo proporcionado con los datos subyacentes "The object data". El código entre comillas incluye una conversión a `obj` porque ese tipo es el borrado de este tipo proporcionado (como se especificó al declarar el tipo proporcionado).

Agregue la documentación XML al constructor y agregue el constructor proporcionado al tipo proporcionado:

```
ctor.AddXmlDocDelayed(fun () -> "This is a constructor")  
  
t.AddMember ctor
```

Cree un segundo constructor proporcionado que tome un parámetro:

```
let ctor2 =  
    ProvidedConstructor(parameters = [ ProvidedParameter("data",typeof<string>) ],  
                        invokeCode = (fun args -> <@@ (%(args.[0]) : string) :> obj @@>))
```

La función `InvokeCode` del constructor devuelve de nuevo una expresión de código delimitada de F# que representa el código que el compilador host generó para una llamada al método. Por ejemplo, puede usar el constructor siguiente:

```
new Type10("ten")
```

Se crea una instancia del tipo proporcionado con los datos subyacentes "ten". Es posible que haya notado que la función `InvokeCode` devuelve una expresión de código delimitada. La entrada de esta función es una lista de expresiones, una por cada parámetro del constructor. En este caso, una expresión que representa el valor del único parámetro está disponible en `args.[0]`. El código de una llamada al constructor convierte el valor devuelto en el tipo borrado `obj`. Después de agregar el segundo constructor proporcionado al tipo, cree una propiedad de instancia proporcionada:

```

let instanceProp =
    ProvidedProperty(propertyName = "InstanceProperty",
        propertyType = typeof<int>,
        getterCode= (fun args ->
            <@@ ((%(args.[0]) : obj) :?)> string).Length @@>))
instanceProp.AddXmlDocDelayed(fun () -> "This is an instance property")
t.AddMember instanceProp

```

Al obtener esta propiedad, se devuelve la longitud de la cadena, que es el objeto de representación. La propiedad `GetterCode` devuelve una expresión de código delimitada de F# que especifica el código que genera el compilador host para obtener la propiedad. Al igual que la función `InvokeCode`, la función `GetterCode` devuelve una expresión de código delimitada. El compilador host llama a esta función con una lista de argumentos. En este caso, los argumentos incluyen solo la expresión única que representa la instancia en la que se llama al captador, a la que se puede tener acceso mediante `args.[0]`. La implementación de `GetterCode` después se inserta en el presupuesto de resultados en el tipo borrado `obj` y se usa una conversión para satisfacer el mecanismo del compilador para comprobar los tipos que el objeto es una cadena. La parte siguiente de `makeOneProvidedType` proporciona un método de instancia con un parámetro.

```

let instanceMeth =
    ProvidedMethod(methodName = "InstanceMethod",
        parameters = [ProvidedParameter("x", typeof<int>)],
        returnType = typeof<char>,
        invokeCode = (fun args ->
            <@@ ((%(args.[0]) : obj) :?)> string).Chars(%(args.[1]) : int) @@>))

instanceMeth.AddXmlDocDelayed(fun () -> "This is an instance method")
// Add the instance method to the type.
t.AddMember instanceMeth

```

Finalmente, cree un tipo anidado que contenga 100 propiedades anidadas. La creación de este tipo anidado y sus propiedades se demora, es decir, se calcula a petición.

```

t.AddMembersDelayed(fun () ->
    let nestedType = ProvidedTypeDefinition("NestedType", Some typeof<obj>)

    nestedType.AddMembersDelayed (fun () ->
        let staticPropsInNestedType =
            [
                for i in 1 .. 100 ->
                    let valueOfTheProperty = "I am string " + string i

                    let p =
                        ProvidedProperty(propertyName = "StaticProperty" + string i,
                            propertyType = typeof<string>,
                            isStatic = true,
                            getterCode= (fun args -> <@@ valueOfTheProperty @@>))

                    p.AddXmlDocDelayed(fun () ->
                        sprintf "This is StaticProperty%d on NestedType" i)

                    p
            ]

        staticPropsInNestedType

    [nestedType])

```

Detalles sobre los tipos proporcionados borrados

En el ejemplo de esta sección se proporcionan solo *tipos proporcionados borrados*, que son especialmente útiles

en las situaciones siguientes:

- Cuando se escribe un proveedor para un espacio de información que solo contiene datos y métodos.
- Cuando se escribe un proveedor en el que la semántica precisa de tipos en tiempo de ejecución no es fundamental para el uso práctico del espacio de información.
- Cuando se escribe un proveedor para un espacio de información que es tan grande y está tan interconectado que no es técnicamente factible generar tipos reales de .NET para el espacio de información.

En este ejemplo, se borra cada tipo proporcionado para el tipo `obj` y todos los usos del tipo aparecen como tipo `obj` en el código compilado. De hecho, los objetos subyacentes de estos ejemplos son cadenas, pero el tipo aparecerá como `System.Object` en el código compilado de .NET. Como con todos los usos del borrado de tipos, se puede utilizar la conversión boxing explícita, la conversión unboxing y la conversión para trastocar los tipos borrados. En este caso, puede producirse una excepción de conversión no válida cuando se utiliza el objeto. Un runtime de un proveedor puede definir su propio tipo de representación privado para ayudar a protegerse contra representaciones falsas. No se pueden definir tipos borrados en F#. Solo se pueden borrar los tipos proporcionados. Se deben entender las implicaciones, tanto prácticas como semánticas, de utilizar los tipos borrados para el proveedor de tipo o un proveedor que proporcione tipos borrados. Un tipo borrado no tiene un tipo real de .NET. Por consiguiente, no se puede hacer una reflexión precisa sobre el tipo y se podrían trastocar los tipos borrados si se utilizan conversiones en tiempo de ejecución y otras técnicas que dependen de semánticas exactas de tipos en tiempo de ejecución. El trastocamiento de tipos borrados frecuentemente da lugar a excepciones de conversión de tipos en tiempo de ejecución.

Elegir representaciones para los tipos proporcionados borrados

Para algunos usos de los tipos proporcionados borrados, no se requiere ninguna representación. Por ejemplo, el tipo proporcionado borrado podría contener únicamente propiedades y miembros estáticos y ningún constructor, por lo que ningún método ni propiedad devolvería ninguna instancia del tipo. Si puede tener acceso a instancias de un tipo proporcionado borrado, considere las preguntas siguientes:

¿Qué es el borrado de un tipo proporcionado?

- El borrado de un tipo proporcionado es el modo en que el tipo aparece en el código compilado de .NET.
- El borrado de una clase de un tipo proporcionado borrado siempre es el primer tipo base no borrado de la cadena de herencia del tipo.
- El borrado de una interfaz de un tipo de borrado proporcionado es siempre `System.Object`.

¿Qué son las representaciones de un tipo proporcionado?

- Se denomina representaciones al conjunto de objetos posibles para un tipo proporcionado borrado. En el ejemplo de este documento, las representaciones de todos los tipos proporcionados borrados `Type1..Type100` son siempre objetos de cadena.

Todas las representaciones de un tipo proporcionado deben ser compatibles con el borrado del tipo proporcionado. (De lo contrario, el compilador de F# generará un error para un uso del proveedor de tipos o se generará código no comprobable de .NET que no es válido. Un proveedor de tipos no es válido si devuelve código que proporciona una representación no válida).

Se puede elegir una representación para los objetos proporcionados mediante uno de los dos métodos siguientes, los cuales son muy comunes:

- Si lo que se hace es simplemente proporcionar un contenedor fuertemente tipado sobre un tipo existente de .NET, tiene sentido que el tipo borre ese tipo, use instancias de ese tipo como representaciones, o ambas cosas. Este enfoque es adecuado cuando la mayoría de los métodos existentes en ese tipo tienen sentido al usar la versión fuertemente tipada.

- Si lo que se desea es crear una API que difiera significativamente de cualquier API existente de .NET, tiene sentido crear tipos en tiempo de ejecución que constituyan la representación y el borrado de tipos para los tipos proporcionados.

El ejemplo de este documento utiliza cadenas como representaciones de los objetos proporcionados. Con frecuencia, puede ser adecuado utilizar otros objetos para las representaciones. Por ejemplo, se puede utilizar un diccionario como contenedor de propiedades:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new Dictionary<string,obj>()) :> obj @@>))
```

También se puede definir un tipo en el proveedor de tipos que se usará en runtime para formar la representación, junto con una o más operaciones en runtime:

```
type DataObject() =
    let data = Dictionary<string,obj>()
    member x.RuntimeOperation() = data.Count
```

Entonces los miembros proporcionados podrán construir instancias de este tipo de objeto:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new DataObject()) :> obj @@>))
```

En este caso, se puede (opcionalmente) utilizar este tipo como borrado de tipos especificando este tipo como `baseType` al construir `ProvidedTypeDefinition`:

```
ProvidedTypeDefinition(..., baseType = Some typeof<DataObject> )
...
ProvidedConstructor(..., InvokeCode = (fun args -> <@@ new DataObject() @@>), ...)
```

Lecciones principales

En la sección anterior se explicó cómo crear un proveedor de tipos de borrado simple que proporciona una serie de tipos, propiedades y métodos. En dicha sección se explicó también el concepto de borrado de tipos, incluidas algunas de las ventajas y desventajas de proporcionar tipos borrados desde un proveedor de tipos, y se explicaron las representaciones de los tipos borrados.

Un proveedor de tipos que usa parámetros estáticos

La capacidad de parametrizar los proveedores de tipo mediante datos estáticos permite muchos escenarios interesantes, incluso en los casos en que el proveedor no necesita tener acceso a ningún dato local o remoto. En esta sección, aprenderá algunas técnicas básicas para construir tales proveedores.

Proveedor de tipo de comprobación de expresiones regulares

Imagine que desea implementar un proveedor de tipos para expresiones regulares que contenga las bibliotecas [Regex](#) de .NET en una interfaz que proporcione las siguientes garantías en tiempo de compilación:

- Comprobar si una expresión regular es válida.
- Proporcionar propiedades con nombre en las coincidencias basadas en cualquier nombre de grupo de la expresión regular.

En esta sección se muestra cómo utilizar los proveedores de tipo para crear un tipo `RegexTyped` parametrizado por el patrón de expresiones regulares para proporcionar estas ventajas. El compilador notificará un error si el patrón

proporcionado no es válido y el proveedor de tipos puede extraer los grupos del patrón de modo que se pueda tener acceso a ellos mediante propiedades con nombre en las coincidencias. Cuando se diseña un proveedor de tipo, se debería considerar el aspecto que debería presentar su API expuesta para los usuarios finales y cómo se traducirá este diseño a código de .NET. El ejemplo siguiente muestra cómo usar una API como esta para obtener los componentes del código de área de un número de teléfono:

```
type T = RegexTyped< @"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)">
let reg = T()
let result = T.IsMatch("425-555-2345")
let r = reg.Match("425-555-2345").Group_AreaCode.Value //r equals "425"
```

El ejemplo siguiente muestra cómo convierte el proveedor de tipos estas llamadas:

```
let reg = new Regex(@"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)")
let result = reg.IsMatch("425-123-2345")
let r = reg.Match("425-123-2345").Groups["AreaCode"].Value //r equals "425"
```

Tenga en cuenta los siguientes puntos:

- El tipo estándar `Regex` representa el tipo parametrizado `RegexTyped`.
- El constructor `RegexTyped` produce una llamada al constructor `Regex` y le pasa el argumento de tipo estático para el patrón.
- Los resultados del método `Match` se representan mediante el tipo estándar `Match`.
- Cada grupo con nombre produce una propiedad proporcionada, y el acceso a la propiedad produce el uso de un indizador en la colección `Groups` de una coincidencia.

El código siguiente es la base de la lógica para implementar un proveedor como este y este ejemplo omite la adición de todos los miembros al tipo proporcionado. Para obtener información sobre cada miembro agregado, consulte la sección correspondiente más adelante en este tema. Para obtener el código completo, descargue el ejemplo del [paquete de ejemplo de F # 3,0](#) en el sitio web de codeplex.

```

namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

            match parameterValues with
            | [| :? string as pattern|] ->

                // Create an instance of the regular expression.
                //
                // This will fail with System.ArgumentException if the regular expression is not valid.
                // The exception will escape the type provider and be reported in client code.
                let r = System.Text.RegularExpressions.Regex(pattern)

                // Declare the typed regex provided type.
                // The type erasure of this type is 'obj', even though the representation will always be a Regex
                // This, combined with hiding the object methods, makes the IntelliSense experience simpler.
                let ty =
                    ProvidedTypeDefinition(
                        thisAssembly,
                        rootNamespace,
                        typeName,
                        baseType = Some baseTy)

                ...

                ty
            | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()

```

Tenga en cuenta los siguientes puntos:

- El proveedor de tipos toma dos parámetros estáticos: `pattern`, el patrón, que es obligatorio, y `options`, las opciones, que son opcionales (porque se proporciona un valor predeterminado).
- Después de proporcionar los argumentos estáticos, se crea una instancia de la expresión regular. Esta instancia inicia una excepción si la Regex no es correcta, y se notifica el error a los usuarios.
- El tipo que se devolverá cuando se den los argumentos se define dentro de la devolución de llamada `DefineStaticParameters`.
- Este código establece `HideObjectMethods` en true para que el uso de IntelliSense siga estando optimizado. Este atributo hace que los miembros `Equals`, `GetHashCode`, `Finalize` y `GetType` se supriman de listas de

IntelliSense para un objeto proporcionado.

- Se utiliza `obj` como tipo base del método, pero se utilizará un objeto `Regex` como representación en tiempo de ejecución de este tipo, como muestra el ejemplo siguiente.
- La llamada al constructor `Regex` inicia una excepción `ArgumentException` cuando una expresión regular no es válida. El compilador detecta esta excepción y envía un mensaje de error al usuario en tiempo de compilación o en el editor de Visual Studio. Esta excepción permite que las expresiones regulares se validen sin ejecutar una aplicación.

El tipo definido anteriormente todavía no es útil porque no contiene propiedades ni métodos significativos. En primer lugar, agregue un método `IsMatch` estático:

```
let isMatch =
    ProvidedMethod(
        methodName = "IsMatch",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = typeof<bool>,
        isStatic = true,
        invokeCode = fun args -> <@@ Regex.IsMatch(%args.[0], pattern) @@>)

isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified input string."
ty.AddMember isMatch
```

El código anterior define un método `IsMatch`, que toma una cadena como entrada y devuelve un valor `bool`. La única parte difícil es el uso del argumento `args` dentro de la definición de la función `InvokeCode`. En este ejemplo, `args` es una lista de expresiones de código delimitadas que representa los argumentos de este método. Si el método es un método de instancia, el primer argumento representa el argumento `this`. Sin embargo, para un método estático, los argumentos son simplemente los argumentos explícitos para el método. Observe que el tipo del valor de la expresión de código delimitada debe coincidir con el tipo del valor devuelto especificado (en este caso, `bool`). Observe también que este código utiliza el método `AddXmlDoc` para asegurarse de que el método proporcionado también tiene documentación útil, que se puede proporcionar con IntelliSense.

A continuación, agregue un método de instancia `Match`. Sin embargo, este método debe devolver un valor de un tipo `Match` proporcionado, de modo que se pueda tener acceso a los grupos de manera fuertemente tipada. Por ello, primero se declara el tipo `Match`. Como este tipo depende del patrón que se proporcionó como argumento estático, este tipo debe estar anidado dentro de la definición de tipos parametrizados:

```
let matchTy =
    ProvidedTypeDefinition(
        "MatchType",
        baseType = Some baseTy,
        hideObjectMethods = true)

ty.AddMember matchTy
```

A continuación, se agrega una propiedad al tipo `Match` de cada grupo. En tiempo de ejecución, una coincidencia se representa como un valor `Match`, por lo que la expresión de código delimitada que define la propiedad debe utilizar la propiedad indexada `Groups` para obtener el grupo pertinente.

```

for group in r.GetGroupNames() do
    // Ignore the group named 0, which represents all input.
    if group <> "0" then
        let prop =
            ProvidedProperty(
                propertyName = group,
                propertyType = typeof<Group>,
                getterCode = fun args -> <@@ ((%args.[0]:obj) :?> Match).Groups.[group] @@>)
                prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
            matchTy.AddMember prop

```

Una vez más, observe que se está agregando la documentación XML a la propiedad proporcionada. También observe que una propiedad puede leerse si se proporciona una función `GetterCode` y puede escribirse si se proporciona una función `SetterCode`, por lo que la propiedad resultante es de solo lectura.

Now you can create an instance method that returns a value of this `Match` type:

```

let matchMethod =
    ProvidedMethod(
        methodName = "Match",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = matchTy,
        invokeCode = fun args -> <@@ ((%args.[0]:obj) :?> Regex).Match(%args.[1]) :> obj @@>)

matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this regular expression"

ty.AddMember matchMeth

```

Como está creando un método de instancia, `args.[0]` representa la instancia de `RegexTyped` en la que se está llamando al método y `args.[1]` es el argumento de entrada.

Finalmente, proporcione un constructor para que se puedan crear instancias del tipo proporcionado.

```

let ctor =
    ProvidedConstructor(
        parameters = [],
        invokeCode = fun args -> <@@ Regex(pattern, options) :> obj @@>)

ctor.AddXmlDoc("Initializes a regular expression instance.")

ty.AddMember ctor

```

El constructor simplemente borra para la creación de una Regex estándar de .NET, a la cual también se le aplica la conversión box en un objeto porque `obj` es el borrado del tipo proporcionado. Con ese cambio, el uso de la API de ejemplo especificada previamente en el tema funciona como se esperaba. A continuación se muestra la versión final del código completo:

```

namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types.
    let thisAssembly = Assembly.GetExecutingAssembly()

```



```

let rootNamespace = "Samples.FSharp.RegexTypeProvider"
let baseTy = typeof<obj>
let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

do regexTy.DefineStaticParameters(
    parameters=staticParams,
    instantiationFunction=(fun typeName parameterValues ->

        match parameterValues with
        | [| :? string as pattern|] ->

            // Create an instance of the regular expression.

            let r = System.Text.RegularExpressions.Regex(pattern)

            // Declare the typed regex provided type.

            let ty =
                ProvidedTypeDefinition(
                    thisAssembly,
                    rootNamespace,
                    typeName,
                    baseType = Some baseTy)

            ty.AddXmlDoc "A strongly typed interface to the regular expression '%s'"

            // Provide strongly typed version of Regex.IsMatch static method.
            let isMatch =
                ProvidedMethod(
                    methodName = "IsMatch",
                    parameters = [ProvidedParameter("input", typeof<string>)],
                    returnType = typeof<bool>,
                    isStatic = true,
                    invokeCode = fun args -> <@@ Regex.IsMatch(%args.[0], pattern) @@>)

            isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified
input string"

            ty.AddMember isMatch

            // Provided type for matches
            // Again, erase to obj even though the representation will always be a Match
            let matchTy =
                ProvidedTypeDefinition(
                    "MatchType",
                    baseType = Some baseTy,
                    hideObjectMethods = true)

            // Nest the match type within parameterized Regex type.
            ty.AddMember matchTy

            // Add group properties to match type
            for group in r.GetGroupNames() do
                // Ignore the group named 0, which represents all input.
                if group <> "0" then
                    let prop =
                        ProvidedProperty(
                            propertyName = group,
                            propertyType = typeof<Group>,
                            getterCode = fun args -> <@@ ((%args.[0]:obj) :?) Match).Groups.[group] @@>)
                    prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
                    matchTy.AddMember(prop)

            // Provide strongly typed version of Regex.Match instance method.
            let matchMeth =
                ProvidedMethod(
                    methodName = "Match",

```

```

        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = matchTy,
        invokeCode = fun args -> <@@ ((%args.[0]:obj) :?)> Regex.Match(%args.[1]) :> obj @@>)
        matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this
regular expression"

        ty.AddMember matchMeth

// Declare a constructor.
let ctor =
    ProvidedConstructor(
        parameters = [],
        invokeCode = fun args -> <@@ Regex(pattern) :> obj @@>)

// Add documentation to the constructor.
ctor.AddXmlDoc "Initializes a regular expression instance"

ty.AddMember ctor

ty
| _ -> failwith "unexpected parameter values")

do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()

```

Lecciones principales

En esta sección se ha explicado cómo crear un proveedor de tipos que opera con sus parámetros estáticos. El proveedor comprueba el parámetro estático y proporciona operaciones basadas en su valor.

Un proveedor de tipos que está respaldado por datos locales

Con frecuencia se requiere que los proveedores de tipos muestren API basadas no solo en parámetros estáticos sino también en información procedente de sistemas locales o remotos. Esta sección trata sobre los proveedores de tipos basados en datos locales, como los archivos de datos locales.

Proveedor simple de archivos CSV

Como ejemplo sencillo, considere un proveedor de tipo para tener acceso a datos científicos con el formato de valores separados por comas (CSV). En esta sección se supone que los archivos CSV contienen una fila de encabezado seguida de datos en coma flotante, como se muestra en la tabla siguiente:

DISTANCIA (METROS)	TIEMPO (SEGUNDOS)
50.0	3.7
100.0	5.2
150.0	6.4

En esta sección se muestra cómo proporcionar un tipo que se puede usar para obtener filas con una propiedad `Distance` de tipo `float<meter>` y una propiedad `Time` de tipo `float<second>`. Para simplificar, se realizan las suposiciones siguientes:

- Los nombres de encabezado son de unidad menos o tienen el formato "nombre (unidad)" y no contienen comas.
- Las unidades son unidades internacionales (SI) del sistema como define el módulo [Microsoft.FSharp.Data.UnitSystems.Si.UnitNames \(F #\)](#).

- Las unidades son todas simples (por ejemplo, metro) en lugar de compuestas (por ejemplo, metros por segundo).
- Todas las columnas contienen datos en coma flotante.

Un proveedor más completo relajaría estas restricciones.

De nuevo, el primer paso es considerar qué aspecto debe tener la API. Dado un archivo `info.csv` con el contenido de la tabla anterior (en formato separado por comas), los usuarios del proveedor deberían poder escribir un código similar al del ejemplo siguiente:

```
let info = new MiniCsv<"info.csv">()
for row in info.Data do
let time = row.Time
printfn "%f" (float time)
```

En este caso, el compilador debería convertir estas llamadas en algo similar al ejemplo siguiente:

```
let info = new CsvFile("info.csv")
for row in info.Data do
let (time:float) = row.[1]
printfn "%f" (float time)
```

La conversión óptima requerirá que el proveedor de tipos defina un tipo `CsvFile` real en el ensamblado del proveedor de tipos. Los proveedores de tipos a veces se basan en algunos tipos y métodos del asistente para contener la lógica importante. Dado que las medidas se borran en tiempo de ejecución, se puede utilizar `float[]` como el tipo borrado para una fila. El compilador considerará que las distintas columnas contienen distintos tipos de medidas. Por ejemplo, la primera columna de nuestro ejemplo contiene el tipo `float<meter>` y la segunda contiene `float<second>`. Sin embargo, la representación borrada puede seguir siendo bastante simple.

En el ejemplo de código siguiente se muestra el núcleo de la implementación.

```
// Simple type wrapping CSV data
type CsvFile(filename) =
    // Cache the sequence of all data lines (all lines but the first)
    let data =
        seq {
            for line in File.ReadAllLines(filename) |> Seq.skip 1 ->
                line.Split(',') |> Array.map float
        }
    |> Seq.cache
    member _.Data = data

[<TypeProvider>]
type public MiniCsvProvider(cfg:TypeProviderConfig) as this =
    inherit TypeProviderForNamespaces(cfg)

    // Get the assembly and namespace used to house the provided types.
    let asm = System.Reflection.Assembly.GetExecutingAssembly()
    let ns = "Samples.FSharp.MiniCsvProvider"

    // Create the main provided type.
    let csvTy = ProvidedTypeDefinition(asm, ns, "MiniCsv", Some(typeof<obj>))

    // Parameterize the type by the file to use as a template.
    let filename = ProvidedStaticParameter("filename", typeof<string>)
    do csvTy.DefineStaticParameters([filename], fun tyName [| :? string as filename |] ->

        // Resolve the filename relative to the resolution folder.
        let resolvedFilename = Path.Combine(cfg.ResolutionFolder, filename))
```

```

// Get the first line from the file.
let headerLine = File.ReadLines(resolvedFilename) |> Seq.head

// Define a provided type for each row, erasing to a float[].
let rowTy = ProvidedTypeDefinition("Row", Some(typeof<float[]>))

// Extract header names from the file, splitting on commas.
// use Regex matching to get the position in the row at which the field occurs
let headers = Regex.Matches(headerLine, "[^,]+")

// Add one property per CSV field.
for i in 0 .. headers.Count - 1 do
    let headerText = headers.[i].Value

    // Try to decompose this header into a name and unit.
    let fieldName, fieldTy =
        let m = Regex.Match(headerText, @"(?<field>.+)\s((?<unit>.+)\s)")
        if m.Success then

            let unitName = m.Groups["unit"].Value
            let units = ProvidedMeasureBuilder.Default.SI unitName
            m.Groups["field"].Value, ProvidedMeasureBuilder.Default.AnnotateType(typeof<float>,
[units])

        else
            // no units, just treat it as a normal float
            headerText, typeof<float>

    let prop =
        ProvidedProperty(fieldName, fieldTy,
            getterCode = fun [row] -> <@@ (%row:float[]).[i] @@>)

    // Add metadata that defines the property's location in the referenced file.
    prop.AddDefinitionLocation(1, headers.[i].Index + 1, filename)
    rowTy.AddMember(prop)

// Define the provided type, erasing to CsvFile.
let ty = ProvidedTypeDefinition(asm, ns, tyName, Some(typeof<CsvFile>))

// Add a parameterless constructor that loads the file that was used to define the schema.
let ctor0 =
    ProvidedConstructor([],
        invokeCode = fun [] -> <@@ CsvFile(resolvedFilename) @@>)
ty.AddMember ctor0

// Add a constructor that takes the file name to load.
let ctor1 = ProvidedConstructor([ProvidedParameter("filename", typeof<string>)],
    invokeCode = fun [filename] -> <@@ CsvFile(%%filename) @@>)
ty.AddMember ctor1

// Add a more strongly typed Data property, which uses the existing property at runtime.
let prop =
    ProvidedProperty("Data", typedefof<seq<_>>.MakeGenericType(rowTy),
        getterCode = fun [csvFile] -> <@@ (%csvFile:CsvFile).Data @@>)
ty.AddMember prop

// Add the row type as a nested type.
ty.AddMember rowTy
ty)

// Add the type to the namespace.
do this.AddNamespace(ns, [csvTy])

```

Tenga en cuenta las siguientes observaciones sobre la implementación:

- Los constructores sobrecargados permiten leer el archivo original o uno que tenga un esquema idéntico. Este patrón es habitual al escribir un proveedor de tipo para orígenes de datos locales o remotos, y además

permite el uso de un archivo local como plantilla para los datos remotos.

- Puede utilizar el valor `TypeProviderConfig` que se pasa al constructor del proveedor de tipos para resolver nombres de archivo relativos.
- Se puede utilizar el método `AddDefinitionLocation` para definir la ubicación de las propiedades proporcionadas. Por lo tanto, si usa `Go To Definition` en una propiedad proporcionada, el archivo CSV se abrirá en Visual Studio.
- Se puede utilizar el tipo `ProvidedMeasureBuilder` para buscar las unidades del SI y generar los tipos `float<_>` pertinentes.

Lecciones principales

En esta sección se ha explicado cómo crear un proveedor de tipo para un origen de datos local con un esquema simple que está contenido en el propio origen de datos.

Ampliar conocimientos

En las secciones siguientes se incluyen sugerencias para ampliar conocimientos sobre el tema.

Un vistazo al código compilado de los tipos borrados

Para tener una idea de cómo se corresponde el uso del proveedor de tipos con el código emitido, revise la función siguiente mediante el proveedor `HelloWorldTypeProvider` utilizado anteriormente en este tema.

```
let function1 () =  
    let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")  
    obj1.InstanceProperty
```

A continuación se muestra una imagen del código resultante descompilado mediante ildasm.exe:

```
.class public abstract auto ansi sealed Module1  
extends [mscorlib]System.Object  
{  
    .custom instance void [FSharp.Core]Microsoft.FSharp.Core.CompilationMappingAttribute::ctor(valuetype [FSharp.Core]Microsoft.FSharp.Core.SourceConstructFlags)  
    = ( 01 00 07 00 00 00 00 00 )  
    .method public static int32 function1() cil managed  
    {  
        // Code size          24 (0x18)  
        .maxstack 3  
        .locals init ([0] object obj1)  
        IL_0000: nop  
        IL_0001: ldstr      "some data"  
        IL_0006: unbox.any [mscorlib]System.Object  
        IL_000b: stloc.0  
        IL_000c: ldloc.0  
        IL_000d: call      !!0 [FSharp.Core_2]Microsoft.FSharp.Core.LanguagePrimitives/IntrinsicFunctions::UnboxGeneric<string>(object)  
        IL_0012: callvirt instance int32 [mscorlib_3]System.String::get_Length()  
        IL_0017: ret  
    } // end of method Module1::function1  
  
} // end of class Module1
```

Como muestra el ejemplo, se han borrado todas las menciones del tipo `Type1` y la propiedad `InstanceProperty`, y quedan solo las operaciones para los tipos de tiempo de ejecución relacionados.

Diseño y convenciones de nomenclatura para los proveedores de tipos

Respete las convenciones siguientes al crear proveedores de tipos.

Proveedores de protocolos de conectividad En general, los nombres de la mayoría de los archivos dll de proveedor para los protocolos de datos y conectividad de servicio, como las conexiones de OData o SQL, deben acabar en `TypeProvider` o `TypeProviders` . Por ejemplo, utilice un nombre de DLL similar a la siguiente cadena:

```
Fabrikam.Management.BasicTypeProviders.dll
```

Asegúrese de que los tipos proporcionados son miembros del espacio de nombres correspondiente e indican el protocolo de conexión que se ha implementado:

```
Fabrikam.Management.BasicTypeProviders.WmiConnection<...>
Fabrikam.Management.BasicTypeProviders.DataProtocolConnection<...>
```

Proveedores de utilidades para la codificación general. Para un proveedor de tipos de utilidad, como el de las expresiones regulares, el proveedor de tipos puede ser parte de una biblioteca base, como se muestra en el ejemplo siguiente:

```
#r "Fabrikam.Core.Text.Utilities.dll"
```

En este caso, el tipo proporcionado aparecería en un punto adecuado según las convenciones normales de diseño de .NET:

```
open Fabrikam.Core.Text.RegexTyped

let regex = new RegexTyped<"a+b+a+b+">()
```

Orígenes de datos singleton. Algunos proveedores de tipos se conectan a un único origen de datos dedicado y solo proporcionan datos. En este caso, se debería colocar el sufijo `TypeProvider` y utilizar las convenciones normales de nomenclatura de .NET:

```
#r "Fabrikam.Data.Freebase.dll"

let data = Fabrikam.Data.Freebase.Astronomy.Asteroids
```

Para obtener más información, vea la convención de diseño `GetConnection` que se describe más adelante en este tema.

Patrones de diseño para los proveedores de tipos

En las secciones siguientes se describen los patrones de diseño que se pueden usar cuando se crean los proveedores de tipos.

El patrón de diseño `GetConnection`

La mayoría de los proveedores de tipos se deben escribir para que usen el patrón `GetConnection` utilizado por los proveedores de tipo en `FSharp.Data.TypeProviders.dll`, como se muestra en el ejemplo siguiente:

```
#r "Fabrikam.Data.WebDataStore.dll"

type Service = Fabrikam.Data.WebDataStore<...static connection parameters...>

let connection = Service.GetConnection(...dynamic connection parameters...)

let data = connection.Astronomy.Asteroids
```

Proveedores de tipos respaldados por datos y servicios remotos

Antes de crear un proveedor de tipos respaldado por datos y servicios remotos, se deben tener en cuenta varios

problemas que son inherentes a la programación conectada. Estos problemas incluyen los siguientes aspectos:

- Asignación de esquemas
- Vida e invalidación en presencia de un cambio de esquema
- Almacenamiento en caché de esquemas
- Implementaciones asíncronas de operaciones de acceso a datos
- Consultas admitidas, incluidas las consultas LINQ
- Credenciales y autenticación

Este tema no profundiza en estos problemas.

Técnicas adicionales de creación

Al escribir sus propios proveedores de tipos, quizá quiera utilizar las siguientes técnicas adicionales.

Crear tipos y miembros bajo demanda

La API de `ProvidedType` tiene versiones demoradas de `AddMember`.

```
type ProvidedType =  
    member AddMemberDelayed : (unit -> MemberInfo) -> unit  
    member AddMembersDelayed : (unit -> MemberInfo list) -> unit
```

Estas versiones se utilizan para crear espacios de tipos a petición.

Proporcionar tipos de matriz y crear instancias de tipos genéricos

Los miembros proporcionados (cuyas firmas incluyen tipos de matriz, tipos de `ByRef` y creaciones de instancias de tipos genéricos) usan el normal `MakeArrayType`, `MakePointerType` y `MakeGenericType` en cualquier instancia de `Type`, incluido `ProvidedTypeDefinitions`.

NOTE

En algunos casos, es posible que tenga que usar el ayudante en `ProvidedTypeBuilder.MakeGenericType`. Consulte la [documentación del SDK del proveedor de tipos](#) para obtener más detalles.

Proporcionar anotaciones de unidades de medida

La API `ProvidedTypes` proporciona asistentes para proporcionar anotaciones de medidas. Por ejemplo, para proporcionar el tipo `float<kg>`, utilice el código siguiente:

```
let measures = ProvidedMeasureBuilder.Default  
let kg = measures.SI "kilogram"  
let m = measures.SI "meter"  
let float_kg = measures.AnnotateType(typeof<float>, [kg])
```

Para proporcionar el tipo `Nullable<decimal<kg/m^2>>`, utilice el código siguiente:

```
let kgpm2 = measures.Ratio(kg, measures.Square m)  
let dkgpm2 = measures.AnnotateType(typeof<decimal>, [kgpm2])  
let nullableDecimal_kgpm2 = typedefof<System.Nullable<_>>.MakeGenericType [|dkgpm2 |]
```

Acceder a recursos locales del proyecto o del script

A cada instancia de un proveedor de tipo se le puede asignar un valor `TypeProviderConfig` durante la construcción.

Este valor contiene la "carpeta de resolución" para el proveedor (es decir, la carpeta del proyecto para la compilación o el directorio que contiene un script), la lista de ensamblados a los que se hace referencia y otra información.

Invalidación

Los proveedores pueden generar señales de invalidación para notificar al servicio del lenguaje F# que las suposiciones acerca del esquema pueden haber cambiado. Cuando se produce la invalidación, se repite una comprobación de tipos si el proveedor se hospeda en Visual Studio. Esta señal se omite cuando el proveedor se hospeda en F# Interactive o por el compilador de F# (fsc.exe).

Almacenar en caché la información del esquema

A menudo los proveedores deben almacenar en memoria caché el acceso a la información del esquema. Los datos almacenados en caché deben almacenarse utilizando un nombre de archivo que se da como parámetro estático o como datos de usuario. Un ejemplo de almacenamiento en caché de esquema es el parámetro `LocalSchemaFile` en los proveedores de tipos del ensamblado `FSharp.Data.TypeProviders`. En la implementación de estos proveedores, este parámetro estático ordena al proveedor de tipos que use la información del esquema del archivo local especificado en lugar de acceder al origen de datos en la red. Para utilizar la información del esquema almacenada en caché, también se debe establecer el parámetro estático `ForceUpdate` en `false`. Se puede usar una técnica similar para permitir el acceso a datos en línea y sin conexión.

Ensamblado de respaldo

Al compilar `.dll` un `.exe` archivo o, el archivo .dll de respaldo para los tipos generados se vincula estáticamente en el ensamblado resultante. Este vínculo se crea copiando las definiciones de tipos del lenguaje intermedio (IL) y cualquier recurso administrado del ensamblado de respaldo al ensamblado final. Cuando se usa F# Interactive, el archivo .dll de respaldo no se copia, sino que se carga directamente en el proceso de F# Interactive.

Excepciones y diagnósticos de proveedores de tipo

Todos los usos de todos los miembros de los tipos proporcionados pueden producir excepciones. En todos los casos, si un proveedor de tipos genera una excepción, el compilador host atribuye el error a un proveedor de tipos específico.

- Las excepciones de proveedores de tipos nunca deben producir errores internos del compilador.
- Los proveedores de tipo no pueden notificar advertencias.
- Cuando un proveedor de tipo se hospeda en el compilador de F#, un entorno de desarrollo de F# o F# Interactive, se detectan todas las excepciones de ese proveedor. La propiedad `Message` es siempre el texto del error y no aparece ningún seguimiento de pila. Si va a iniciar una excepción, puede iniciar los ejemplos siguientes: `System.NotSupportedException`, `System.IO.IOException`, `System.Exception`.

Proporcionar tipos generados

Hasta ahora, en este documento se ha explicado cómo proporcionar tipos borrados. También se puede usar el mecanismo de proveedores de tipo de F# para proporcionar tipos generados, que se agregan como definiciones de tipo reales de .NET en el programa del usuario. Se debe hacer referencia a los tipos proporcionados generados mediante una definición de tipo.

```
open Microsoft.FSharp.TypeProviders
```

```
type Service = ODataService<"http://services.odata.org/Northwind/Northwind.svc/">
```

El código del asistente `ProvidedTypes-0.2` que forma parte de la versión 3.0 de F# solo tiene compatibilidad limitada para proporcionar tipos generados. Los enunciados siguientes deben ser verdaderos para una definición de un tipo generado:

- `isErased` se debe establecer en `false`.

- El tipo generado se debe agregar a un recién construido `ProvidedAssembly()`, que representa un contenedor para fragmentos de código generados.
- El proveedor debe tener un ensamblado que tenga un archivo .dll de respaldo real de .NET con un archivo .dll coincidente en el disco.

Reglas y limitaciones

Al escribir proveedores de tipos, tenga en cuenta las siguientes reglas y limitaciones.

Los tipos proporcionados deben ser accesibles

Debe tenerse acceso a todos los tipos proporcionados desde los tipos no anidados. Los tipos no anidados se proporcionan en la llamada al constructor `TypeProviderForNamespaces` o en una llamada a `AddNamespace`. Por ejemplo, si el proveedor proporciona un tipo `StaticClass.P : T`, debe asegurarse de que T es un tipo no anidado o está anidado debajo de uno.

Por ejemplo, algunos proveedores tienen una clase estática como `DataTypes` que contiene estos tipos `T1, T2, T3, ...`. De lo contrario, el error indica que se ha encontrado una referencia al tipo T en el ensamblado A, pero el tipo no se encuentra en ese ensamblado. Si aparece este error, compruebe que se puede obtener acceso a todos los subtipos desde los tipos del proveedor. Nota: estos `T1, T2, T3...` tipos se conocen como tipos *sobre la marcha*. Recuerde colocarlos en un espacio de nombres accesible o en un tipo primario.

Limitaciones del mecanismo de proveedores de tipos

El mecanismo de proveedores de tipos de F# tiene las siguientes limitaciones:

- La infraestructura subyacente para los proveedores de tipos de F# no admite tipos genéricos proporcionados ni métodos genéricos proporcionados.
- El mecanismo no admite tipos anidados con parámetros estáticos.

Sugerencias de desarrollo

Puede que le resulten útiles las siguientes sugerencias durante el proceso de desarrollo:

Ejecutar dos instancias de Visual Studio

Puede desarrollar el proveedor de tipo en una instancia y probarlo en la otra porque el IDE de prueba tomará un bloqueo en el archivo .dll que evita que se recompile el proveedor de tipo. Por lo tanto, debe cerrar la segunda instancia de Visual Studio mientras se compila el proveedor en la primera y, a continuación, debe volver a abrir la segunda instancia después de compilar el proveedor.

Depurar proveedores de tipos mediante invocaciones de FSC. exe

Puede invocar proveedores de tipo mediante las herramientas siguientes:

- `fsc.exe` (el compilador de línea de comandos de F#)
- `fsi.exe` (el compilador interactivo de F#)
- `devenv.exe` (Visual Studio)

A menudo, lo más fácil es depurar los proveedores de tipo mediante `fsc.exe` en un archivo de script de prueba (por ejemplo, `script.fsx`). Puede iniciar un depurador desde el símbolo del sistema.

```
devenv /debugexe fsc.exe script.fsx
```

Puede usar `print-to-stdout` como registro.

Consulta también

- [Proveedores de tipos](#)
- [SDK del proveedor de tipos](#)

Seguridad del proveedor de tipos

23/10/2019 • 4 minutes to read • [Edit Online](#)

Los proveedores de tipos son ensamblados (DLL) al que hace referencia su F# proyecto o script que contienen código para conectarse a orígenes de datos externos y exponer esta información de tipo para el F# entorno de tipo. Normalmente, el código en ensamblados de referencia solo se ejecuta cuando se compila y, a continuación, ejecute el código (o en el caso de una secuencia de comandos, enviar el código para F# interactivo). Sin embargo, un ensamblado de proveedor de tipo se ejecutará dentro de Visual Studio cuando simplemente se puede examinar el código en el editor. Esto sucede porque los proveedores de tipos deben ejecutar para agregar información adicional en el editor, como información rápida sobre herramientas, las finalizaciones de IntelliSense y así sucesivamente. Como resultado, hay consideraciones de seguridad adicional para el tipo de ensamblados de proveedor, ya que se ejecutan automáticamente dentro del proceso de Visual Studio.

Cuadro de diálogo de advertencia de seguridad

Cuando se usa un ensamblado de proveedor de tipo concreto por primera vez, Visual Studio muestra un cuadro de diálogo de seguridad que le advierte que el proveedor de tipos está a punto de ejecutarse. Antes de Visual Studio carga el proveedor de tipos, ofrece la oportunidad de decidir si confiar en este proveedor determinado. Si confía en el origen del proveedor de tipo, a continuación, seleccione "confiar en este proveedor de tipo". Si no confía en el origen del proveedor de tipo, a continuación, seleccione "no confíe este proveedor de tipo". Confiar en el proveedor le permite ejecutar dentro de Visual Studio y proporcionar IntelliSense y generar características. Pero si el propio proveedor de tipo es malintencionado, ejecuta su código podría poner en peligro el equipo.

Si el proyecto contiene código que hace referencia a los proveedores de tipos que eligió en el cuadro de diálogo no debe confiar en, a continuación, en tiempo de compilación, el compilador notificará un error que indica que el proveedor de tipos no es de confianza. Los tipos que son dependientes en el proveedor de tipos de confianza se indican mediante subrayados ondulados rojos. Es seguro examinar el código en el editor.

Si decide cambiar la configuración de confianza directamente en Visual Studio, realice los pasos siguientes.

Para cambiar la configuración de confianza para los proveedores de tipos

1. En el **Tools** menú, seleccione **Options** y expanda el **F# Tools** nodo.
2. Seleccione **Type Providers**, en la lista de proveedores de tipos, seleccione la casilla de verificación para los proveedores de tipos que confía y desactive la casilla de verificación para aquellos que no confía.

Vea también

- [Proveedores de tipos](#)

Solución de problemas en proveedores de tipos

23/10/2019 • 4 minutes to read • [Edit Online](#)

En este tema se describe y proporciona posibles soluciones para los problemas que es más probable que encuentre al usar los proveedores de tipos.

Posibles problemas con los proveedores de tipos

Si se produce un problema cuando se trabaja con proveedores de tipos, puede revisar la siguiente tabla para las soluciones más comunes.

PROBLEMA	ACCIONES SUGERIDAS
Los cambios de esquema. Escribir proveedores funcionan mejor cuando el esquema de origen de datos es estable. Si agrega una tabla de datos o una columna o realizar otro cambio en ese esquema, el proveedor de tipos no reconoce automáticamente estos cambios.	Limpiar o recompilar el proyecto. Para limpiar el proyecto, elija compilar, Clean <i>ProjectName</i> en la barra de menús. Para volver a generar el proyecto, elija compilar, recompilar <i>ProjectName</i> en la barra de menús. Estas acciones restablecer todos los Estados del proveedor de tipo y forzar el proveedor para volver a conectarse al origen de datos y obtener información de esquema actualizado.
Error de conexión. La cadena de conexión o la dirección URL es correcta, la red está inactiva o el origen de datos o el servicio no está disponible.	Para un servicio web o servicio de OData, puede probar la dirección URL en Internet Explorer para comprobar si la dirección URL es correcta y el servicio está disponible. Una cadena de conexión de base de datos, puede usar las herramientas de conexión de datos de Explorador de servidores para comprobar si la cadena de conexión es válida y está disponible la base de datos. Después de restaurar la conexión, debe limpiar o recompilar el proyecto para que el proveedor de tipos se volverá a conectar a la red.
Credenciales no válidas. Debe tener permisos válidos para el servicio web o el origen de datos.	<p>Para una conexión de SQL, el nombre de usuario y la contraseña que se especifican en el archivo de configuración o la cadena de conexión deben ser válidos para la base de datos. Si usas la autenticación de Windows, debe tener acceso a la base de datos. El Administrador de base de datos puede identificar qué permisos que necesita para tener acceso a cada base de datos y cada elemento dentro de una base de datos.</p> <p>Para un servicio web o un servicio de datos, debe tener las credenciales adecuadas. La mayoría de los proveedores de tipos proporcionan un objeto DataContext, que contiene una propiedad de las credenciales que se puede establecer con el nombre de usuario correcto y la clave de acceso.</p>
Ruta de acceso no válida. Una ruta de acceso a un archivo no era válido.	Compruebe si la ruta de acceso es correcta y el archivo existe. Además, debe entrecomillar adecuadamente cualquier backslashes en la ruta de acceso o utilizar una cadena textual o cadenas delimitadas por comillas triples.

Vea también

- [Proveedores de tipos](#)

Programación interactiva con F#

04/11/2019 • 9 minutes to read • [Edit Online](#)

NOTE

En este artículo actualmente solo se describe la experiencia para Windows. Se reescribirá.

NOTE

El vínculo de la referencia de API le llevará a MSDN. La referencia de API de docs.microsoft.com no está completa.

F# Interactive (fsi.exe) se utiliza para ejecutar código de F# de manera interactiva en la consola o para ejecutar scripts de F#. En otras palabras, F# Interactive ejecuta un bucle REPL (del inglés Read, Evaluate, Print Loop - bucle Leer, Evaluar, Imprimir) para el lenguaje F#.

Para usar F# Interactive desde la consola, ejecute fsi.exe. Encontrará FSI.exe en:

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\<sku>\Common7\IDE\CommonExtensions\Microsoft\FSharp
```

donde `sku` es `Community`, `Professional` o `Enterprise`.

Para obtener más información sobre las opciones de línea de comandos disponibles, vea [Opciones de F# Interactive](#).

Para ejecutar F# Interactive a través de Visual Studio, puede hacer clic en el botón **F# Interactive** de la barra de herramientas o presionar las teclas **Ctrl+Alt+F**. De este modo, se abrirá la ventana interactiva, que es una ventana de herramientas en la que se ejecuta una sesión de F# Interactive. También puede seleccionar el código que quiere ejecutar en la ventana interactiva y presionar la combinación de teclas **ALT+ENTRAR**. F# Interactive se inicia en la ventana de herramientas con la etiqueta **F# Interactive**. Cuando use esta combinación de teclas, asegúrese de que la ventana del editor tiene el foco.

Tanto si usa la consola como si usa Visual Studio, aparece un símbolo del sistema y el intérprete espera una entrada por parte del usuario. Puede escribir código tal y como lo haría en un archivo de código fuente. Para compilar y ejecutar el código, escriba dos signos de punto y coma (;;) para finalizar una o varias líneas de entrada.

F# Interactive intenta compilar el código y, si lo logra, lo ejecuta e imprime en pantalla la signatura de los tipos y valores que compiló. Si se producen errores, el intérprete imprime en pantalla los mensajes de error.

El código escrito en una misma sesión tiene acceso a cualquier construcción escrita anteriormente, de modo que es posible crear programas. Un búfer extenso de la ventana de herramientas permite copiar el código en un archivo si es necesario.

Cuando F# Interactive se ejecuta en Visual Studio, lo hace de manera independiente del proyecto, de modo que, por ejemplo, no se pueden usar en F# Interactive las construcciones definidas en el proyecto a menos que se copie el código de dichas funciones en la ventana interactiva.

Si tiene un proyecto abierto que hace referencia a algunas bibliotecas, puede hacer referencia a ellas en F# Interactive a través del **Explorador de soluciones**. Para hacer referencia a una biblioteca en F# Interactive, expanda el nodo **Referencias**, abra el menú contextual de la biblioteca y seleccione **Enviar a F# Interactive**.

Puede controlar los argumentos (opciones) de la línea de comandos de F# Interactive ajustando la configuración.

En el menú **Herramientas**, seleccione **Opciones...** y, después, expanda **Herramientas de F#**. Las dos configuraciones que puede cambiar son las opciones de F# Interactive y la opción **F# Interactive de 64 bits**, que solo es relevante si ejecuta F# Interactive en un equipo de 64 bits. Este valor determina si desea ejecutar la versión de 64 bits dedicada de `fsi.exe` o de `fsianycpu.exe`, que usa la arquitectura del equipo para determinar si debe ejecutarse como un proceso de 32 o de 64 bits.

Scripting con F#

Los scripts usan la extensión de archivo `.fsx` o `.fsscript`. En lugar de compilar el código fuente y después ejecutar el ensamblado compilado, se puede ejecutar simplemente `fsi.exe` y especificar el nombre de archivo del script de código fuente de F#. F# Interactive lee el código y lo ejecuta en tiempo real.

Diferencias entre los entornos interactivo, compilado y de scripting

Al compilar código en F# Interactive, tanto si se ejecuta de forma interactiva como si ejecuta un script, se define el símbolo **INTERACTIVE**. Al compilar código en el compilador, se define el símbolo **COMPILED**. Por consiguiente, si el código debe ser diferente en modo interactivo y en modo compilado, se pueden usar las directivas de preprocesador de la compilación condicional para determinar cuál se va a usar.

Cuando se ejecutan scripts en F# Interactive, están disponibles algunas directivas que no están disponibles cuando se ejecuta el compilador. En la siguiente tabla se resumen las directivas que están disponibles cuando se usa F# Interactive.

DIRECTIVA	DESCRIPCIÓN
#help	Muestra información sobre las directivas disponibles.
#I	Especifica una ruta de búsqueda de ensamblado entre comillas.
#load	Lee un archivo de código fuente, lo compila y lo ejecuta.
#quit	Termina una sesión de F# Interactive.
#r	Hace referencia a un ensamblado.
#time ["on" "off"]	Por sí solo, #time activa y desactiva la presentación de información sobre el rendimiento. Cuando está habilitado, F# Interactive mide el tiempo real, el tiempo de CPU y la información sobre recolección de elementos no utilizados que se interpreta y ejecuta.

Al especificar los archivos o rutas de acceso en F# Interactive, se espera un literal de cadena. Por tanto, los archivos y las rutas de acceso deben estar entre comillas y se aplicarán los caracteres de escape habituales. Asimismo, puede usar el carácter `@` para hacer que F# Interactive interprete una cadena que contenga una ruta de acceso como una cadena textual. Esto hace que F# Interactive pase por alto cualquier carácter de escape.

Una de las diferencias entre el modo compilado y el modo interactivo es la manera en que se obtiene acceso a los argumentos de la línea de comandos. En modo compilado, use `System.Environment.GetCommandLineArgs`. En los scripts, use `fsi.CommandLineArgs`.

En el código siguiente se muestra cómo crear una función que lea los argumentos de la línea de comandos en un script y cómo hacer referencia a otro ensamblado desde un script. El primer archivo de código, **MyAssembly.fs**, contiene el código del ensamblado al que se hace referencia. Compile este archivo con la línea de comandos: `fsc -a MyAssembly.fs` y, después, ejecute el segundo archivo como un script con la línea de comandos: `fsi --exec`

file1.fsx test

```
// MyAssembly.fs
module MyAssembly
let myFunction x y = x + 2 * y
```

```
// file1.fsx
#r "MyAssembly.dll"

printfn "Command line arguments: "

for arg in fsi.CommandLineArgs do
    printfn "%s" arg

printfn "%A" (MyAssembly.myFunction 10 40)
```

La salida es la siguiente:

```
Command line arguments:
file1.fsx
test
90
```

Temas relacionados

TITLE	DESCRIPCIÓN
Opciones de F# Interactive	Describe la sintaxis de línea de comandos y las F# opciones de la Interactive, FSI. exe.
Referencia de la biblioteca interactiva de F#	Describe la funcionalidad de bibliotecas que está disponible cuando se ejecuta código en F# Interactive.

Novedades de la versión 4.7

19/03/2020 • 3 minutes to read • [Edit Online](#)

F 4.7 agrega varias mejoras al lenguaje f.

Introducción

F 4.7 está disponible en todas las distribuciones de .NET Core y las herramientas de Visual Studio. [Empiece a utilizar F](#) para obtener más información.

Versión de lenguaje

El compilador de F 4.7 presenta la capacidad de establecer la versión de idioma efectiva a través de una propiedad en el archivo de proyecto:

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

Puede establecerlo en `4.6` los `4.7` `latest` valores `preview` , , y . El valor predeterminado es `latest` .

Si lo establece `preview` en , el compilador activará todas las características de vista previa de F .

Rendimientos implícitos

Ya no es `yield` necesario aplicar la palabra clave en matrices, listas, secuencias o expresiones de cálculo donde se puede deducir el tipo. En el ejemplo siguiente, `yield` ambas expresiones requerían la instrucción para cada entrada anterior a F 4.7:

```
let s = seq { 1; 2; 3; 4; 5 }

let daysOfWeek includeWeekend =
[
    "Monday"
    "Tuesday"
    "Wednesday"
    "Thursday"
    "Friday"
    if includeWeekend then
        "Saturday"
        "Sunday"
]
```

Si introduce una `yield` sola palabra clave, `yield` todos los demás elementos también deben haberse aplicado a ella.

Los rendimientos implícitos no se activan `yield!` cuando se utilizan en una expresión que también se utiliza para hacer algo como aplanar una secuencia. Debe seguir utilizando `yield` hoy en día en estos casos.

Identificadores comodín

En el código de F que implica clases, el autoidentificador debe ser siempre explícito en las declaraciones de

miembro. Pero en los casos en que el autoidentificador nunca se utiliza, tradicionalmente ha sido convención usar un carácter de subrayado doble para indicar un autoidentificador sin nombre. Ahora puede utilizar un solo carácter de subrayado:

```
type C() =  
  member _M() = ()
```

Esto también `for` se aplica para los loops:

```
for _ in 1..10 do printfn "Hello!"
```

Relajaciones de sangría

Antes de F 4.7, los requisitos de sangría para el constructor principal y los argumentos de miembro estático requerían una sangría excesiva. Ahora solo requieren un único ámbito de sangría:

```
type OffsideCheck(a:int,  
  b:int, c:int,  
  d:int) = class end  
  
type C() =  
  static member M(a:int,  
    b:int, c:int,  
    d:int) = 1
```

Novedades de F# 4,6

05/02/2020 • 2 minutes to read • [Edit Online](#)

F#4,6 agrega varias mejoras en el F# lenguaje.

Primeros pasos

F#4,6 está disponible en todas las distribuciones de .NET Core y las herramientas de Visual Studio. Comience a usar para obtener más información. [F#](#)

Registros anónimos

Los [registros anónimos](#) son un nuevo F# tipo de tipo F# introducido en 4,6. Son agregados simples de valores con nombre que no es necesario declarar antes de su uso. Puede declararlos como Structs o tipos de referencia. Son tipos de referencia de forma predeterminada.

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

También se pueden declarar como Structs para cuando desee agrupar tipos de valor y funcionen en escenarios con distinción de rendimiento:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    struct {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

Son bastante eficaces y se pueden usar en numerosos escenarios. Obtenga más información en [registros anónimos](#).

Funciones de ValueOption

El tipo ValueOption agregado en F# 4,5 ahora tiene "paridad de función enlazada a módulo" con el tipo de opción. Algunos de los ejemplos más usados son los siguientes:

```
// Multiply a value option by 2 if it has value
let xOpt = ValueSome 99
let result = xOpt |> ValueOption.map (fun v -> v * 2)

// Reverse a string if it exists
let strOpt = ValueSome "Mirror image"
let reverse (str: string) =
    match str with
    | null
    | "" -> ValueNone
    | s ->
        str.ToCharArray()
        |> Array.rev
        |> string
        |> ValueSome

let reversedString = strOpt |> ValueOption.bind reverse
```

Esto permite usar ValueOption como opción en escenarios en los que tener un tipo de valor mejora el rendimiento.

Novedades de F # 4,5

23/07/2020 • 7 minutes to read • [Edit Online](#)

F # 4,5 agrega varias mejoras en el lenguaje F #. Muchas de estas características se agregaron juntas para que pueda escribir código eficaz en F #, al mismo tiempo que garantiza que este código es seguro. Esto significa agregar algunos conceptos al lenguaje y una cantidad significativa de análisis del compilador cuando se usan estas construcciones.

Introducción

F # 4,5 está disponible en todas las distribuciones de .NET Core y las herramientas de Visual Studio. Para obtener más información, [consulte Introducción a F #](#).

Structs de intervalo y de tipo ByRef

El [Span<T>](#) tipo introducido en .net Core le permite representar búferes en memoria de una manera fuertemente tipada, que ahora se permite en f # a partir de f # 4,5. En el ejemplo siguiente se muestra cómo se puede volver a usar una función que funciona en [Span<T>](#) con representaciones de búfer diferentes:

```
let safeSum (bytes: Span<byte>) =
    let mutable sum = 0
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes.[i]
    sum

// managed memory
let arrayMemory = Array.zeroCreate<byte>(100)
let arraySpan = new Span<byte>(arrayMemory)

safeSum(arraySpan) |> printfn "res = %d"

// native memory
let nativeMemory = Marshal.AllocHGlobal(100);
let nativeSpan = new Span<byte>(nativeMemory.ToPointer(), 100)

safeSum(nativeSpan) |> printfn "res = %d"
Marshal.FreeHGlobal(nativeMemory)

// stack memory
let mem = NativePtr.stackalloc<byte>(100)
let mem2 = mem |> NativePtr.toVoidPtr
let stackSpan = Span<byte>(mem2, 100)

safeSum(stackSpan) |> printfn "res = %d"
```

Un aspecto importante es que el intervalo y otros [Structs similares a ByRef](#) tienen un análisis estático muy rígido realizado por el compilador que restringen su uso de maneras que podría encontrar inesperado. Este es el equilibrio fundamental entre el rendimiento, la expresividad y la seguridad introducida en F # 4,5.

Byrefs renovado

Antes de F # 4,5, [Byrefs](#) en f # eran inseguros y desaparecían para numerosas aplicaciones. Los problemas de solidez en torno a byrefs se han solucionado en F # 4,5 y el mismo análisis estático realizado para las estructuras span y tipo ByRef también se ha aplicado.

inref<'t> y outref<'t>

Para representar la noción de un puntero administrado de solo lectura, de solo escritura y de lectura/escritura, F # 4,5 presenta los `inref<'T>` `outref<'T>` tipos, para representar los punteros de solo lectura y solo escritura, respectivamente. Cada uno tiene una semántica diferente. Por ejemplo, no se puede escribir en un `inref<'T>` :

```
let f (dt: inref<DateTime>) =  
    dt <- DateTime.Now // ERROR - cannot write to an inref!
```

De forma predeterminada, la inferencia de tipos deducirá punteros administrados como si `inref<'T>` estuvieran en línea con la naturaleza inmutable del código de F #, a menos que ya se haya declarado como mutable. Para hacer que se pueda escribir en algo, debe declarar un tipo como `mutable` antes de pasar su dirección a una función o miembro que lo manipule. Para obtener más información, consulte [Byrefs](#).

Estructuras readonly

A partir de F # 4,5, puede anotar un struct con [IsReadOnlyAttribute](#) como tal:

```
[<IsReadOnly; Struct>]  
type S(count1: int, count2: int) =  
    member x.Count1 = count1  
    member x.Count2 = count2
```

Esto no le permite declarar un miembro mutable en el struct y emite metadatos que permiten que F # y C# lo traten como de solo lectura cuando se consumen de un ensamblado. Para obtener más información, vea [Structs de solo lectura](#).

Punteros void

El `voidptr` tipo se agrega a F # 4,5, al igual que las funciones siguientes:

- `NativePtr.ofVoidPtr` para convertir un puntero void en un puntero int nativo
- `NativePtr.toVoidPtr` para convertir un puntero int nativo en un puntero void

Esto resulta útil al interoperar con un componente nativo que hace uso de punteros void.

La palabra clave `match!`

La `match!` palabra clave mejora la coincidencia de patrones cuando se encuentra dentro de una expresión de cálculo:

```
// Code that returns an asynchronous option
let checkBananaAsync (s: string) =
    async {
        if s = "banana" then
            return Some s
        else
            return None
    }

// Now you can use 'match!'
let funcWithString (s: string) =
    async {
        match! checkBananaAsync s with
        | Some bananaString -> printfn "It's banana!"
        | None -> printfn "%s" s
    }
```

Esto permite acortar el código que suele implicar opciones de combinación (u otros tipos) con expresiones de cálculo como Async. Para obtener más información, vea [Match!](#).

Requisitos de conversión relajada en las expresiones de matriz, lista y secuencia

La combinación de tipos en los que uno puede heredar de otro dentro de las expresiones de matriz, lista y secuencia solía haber requerido la conversión de cualquier tipo derivado a su tipo primario con `:>` o `upcast`. Esto ahora es relajado, como se muestra a continuación:

```
let x0 : obj list = [ "a" ] // ok pre-F# 4.5
let x1 : obj list = [ "a"; "b" ] // ok pre-F# 4.5
let x2 : obj list = [ yield "a" :> obj ] // ok pre-F# 4.5

let x3 : obj list = [ yield "a" ] // Now ok for F# 4.5, and can replace x2
```

Flexibilización de la sangría para las expresiones de matriz y lista

Antes de F # 4,5, era necesario aplicar sangría excesiva a las expresiones de matriz y lista cuando se pasan como argumentos a las llamadas a métodos. Esto ya no es necesario:

```
module NoExcessiveIndenting =
    System.Console.WriteLine(format="{0}", arg = [|
        "hello"
    |])
    System.Console.WriteLine([|
        "hello"
    |])
```

Referencia del lenguaje F#

08/01/2020 • 18 minutes to read • [Edit Online](#)

Esta sección es una referencia al F# lenguaje, un lenguaje de programación multiparadigma destinado a .net. El lenguaje F# admite los modelos de programación funcional, orientada a objetos e imperativa.

Tokens de F#

En la tabla siguiente se muestran temas de referencia que proporcionan tablas de palabras clave, símbolos y literales que se usan como tokens en F#.

TITLE	DESCRIPCIÓN
Referencia de palabras clave	Contiene vínculos a información sobre todas las palabras clave del lenguaje F#.
Referencia de símbolos y operadores	Contiene una tabla de símbolos y operadores que se usan en el lenguaje F#.
Literales	Describe la sintaxis de los valores literales en F# y cómo especificar la información de tipos para los literales de F#.

Conceptos del lenguaje F#

En la tabla siguiente se muestran los temas de referencia disponibles que describen los conceptos del lenguaje.

TITLE	DESCRIPCIÓN
Funciones	Las funciones son la unidad fundamental de la ejecución del programa en cualquier lenguaje de programación. Al igual que en otros lenguajes, una función de F# tiene un nombre, puede tener parámetros y tomar argumentos, y tiene un cuerpo. F# también admite construcciones de programación funcional como el tratamiento de las funciones como valores, el uso de funciones sin nombre en expresiones, la composición de funciones para crear nuevas funciones, funciones curricadas y la definición implícita de funciones mediante la aplicación parcial de argumentos de función.
Tipos en F#	Describe los tipos que se usan en F#, cómo se les asigna un nombre y se describen.
Inferencia de tipos	Describe cómo el compilador de F# realiza la inferencia de los tipos de valor, variables, parámetros y valores devueltos.
Generalización automática	Describe las construcciones genéricas en F#.

TITLE	DESCRIPCIÓN
Herencia	Describe la herencia, que se usa para modelar la relación de identidad o los subtipos en la programación orientada a objetos.
Miembros	Describe los miembros de los tipos de objeto de F#.
Parámetros y argumentos	Describe la compatibilidad con lenguajes para definir parámetros y pasar argumentos a las funciones, los métodos y las propiedades. Incluye información sobre cómo pasar por referencia.
Sobrecarga de operadores	Describe cómo sobrecargar los operadores aritméticos de un tipo de clase o registro, así como en el nivel global.
Conversiones	Describe la compatibilidad con las conversiones de tipos en F#.
Control de acceso	Describe el control de acceso en F#. El control de acceso se refiere a declarar qué clientes podrán usar determinados elementos de programa, tales como tipos, métodos, funciones, etc.
Coincidencia de patrones	Describe los patrones, que son las reglas para transformar los datos de entrada que se usan en todo el lenguaje F# para extraer datos de comparación con un patrón, descomponer datos en sus partes constituyentes o extraer información de los datos de varias maneras.
Patrones activos	Describe los patrones activos. Los patrones activos permiten definir particiones con nombre que subdividen los datos de entrada. Se pueden usar patrones activos para descomponer los datos de manera personalizada para cada partición.
Aserciones	Describe la expresión <code>assert</code> , que es una característica de depuración que se puede usar para probar una expresión. En caso de error en modo de depuración, una aserción genera un cuadro de diálogo de error del sistema.
Control de excepciones	Contiene información sobre la compatibilidad con el control de excepciones del lenguaje F#.
Atributos	Describe los atributos, que permiten aplicar metadatos a una construcción de programación.
Resource Management: The <code>use</code> Keyword (Administración de recursos: la palabra clave <code>use</code>)	Describe las palabras clave <code>use</code> y <code>using</code> , que permiten controlar la inicialización y la liberación de los recursos.
Espacios de nombres	Describe la compatibilidad con los espacios de nombres en F#. Un espacio de nombres permite organizar el código en áreas de funcionalidad relacionada, lo que permite adjuntar un nombre a una agrupación de elementos de programa.

TITLE	DESCRIPCIÓN
Módulos	Describe los módulos. Un módulo de F# es una agrupación de código de F#, como valores, tipos y valores de función, en un programa de F#. Agrupar el código en módulos ayuda a mantener junto el código relacionado y a evitar conflictos de nombres en los programas.
Declaraciones de importación: la palabra clave <code>open</code>	Describe cómo funciona <code>open</code> . Una declaración de importación especifica un módulo o un espacio de nombres a cuyos elementos se puede hacer referencia sin usar un nombre completo.
Firmas	Describe las firmas y los archivos de firma. Un archivo de signatura contiene información sobre las firmas públicas de un conjunto de elementos de programa F# como, por ejemplo, tipos, espacios de nombres y módulos. Puede usarse para especificar la accesibilidad de estos elementos de programa.
Documentación de XML	Describe la compatibilidad con la generación de archivos de documentación para comentarios de documentación XML, también denominados comentarios de barra diagonal triple. Se puede generar documentación a partir de los comentarios de código en F# exactamente igual que en los demás lenguajes .NET.
Sintaxis detallada	Describe la sintaxis para las construcciones de F# cuando no está habilitada la sintaxis ligera. La directiva <code>#light "off"</code> en la parte superior del archivo de código indica que se trata de la sintaxis detallada.

Tipos en F#

En la tabla siguiente se muestran los temas de referencia disponibles que describen los tipos admitidos por el lenguaje F#.

TITLE	DESCRIPCIÓN
Valores	Describe los valores, que son cantidades inmutables que tienen un tipo específico. Los valores pueden ser números enteros o de punto flotante, caracteres o texto, listas, secuencias, matrices, tuplas, uniones discriminadas, registros, tipos de clase o valores de función.
Tipos básicos	Describe los tipos básicos fundamentales que se usan en el F# lenguaje. También se proporcionan los tipos de .NET correspondientes y los valores máximos y mínimos para cada tipo.
Tipo unit	Describe el tipo <code>unit</code> , que indica la ausencia de un valor concreto. El tipo <code>unit</code> tiene solo un valor, que actúa como marcador de posición cuando no existe o no se necesita ningún otro valor.

TITLE	DESCRIPCIÓN
Cadenas	Describe las cadenas en F#. El tipo <code>string</code> representa texto inmutable como una secuencia de caracteres Unicode. <code>string</code> es un alias de <code>System.String</code> en .NET Framework.
Tuplas	Describe las tuplas, que son agrupaciones de valores sin nombre pero ordenados cuyos tipos pueden ser diferentes.
Tipos de colección F#	Información general de los tipos de colección funcionales de F#, incluidos los tipos de matrices, listas, secuencias (seq), asignaciones y conjuntos.
Listas	Describe las listas. En F#, una lista es una serie ordenada e inmutable de elementos del mismo tipo.
Opciones	Describe el tipo de opción. En F#, se usa una opción cuando un valor puede o no existir. Una opción tiene un tipo subyacente y puede contener un valor de ese tipo o no tener ningún valor.
Secuencias	Describe las secuencias. Una secuencia es una serie lógica de elementos del mismo tipo. Los elementos individuales de la secuencia únicamente se calculan si es necesario, de modo que la representación pueda ser menor que lo que indica el recuento literal de elementos.
Matrices	Describe las matrices. Las matrices son secuencias mutables de tamaño fijo y basadas en cero de elementos de datos consecutivos, todos del mismo tipo.
Registros	Describe los registros. Los registros representan agregados simples de valores con nombre, opcionalmente con miembros.
Uniones discriminadas	Describe las uniones discriminadas, que proporcionan compatibilidad con valores que pueden ser uno de los diversos casos con nombre, cada uno con valores y tipos posiblemente diferentes.
Enumeraciones	Describe las enumeraciones, que son tipos que tienen un conjunto definido de valores con nombre. Se pueden usar en lugar de los literales para que el código sea más fácil de leer y mantener.
Celdas de referencia	Describe las celdas de referencia, que son ubicaciones de almacenamiento que permiten crear variables mutables con semántica de referencias.
Abreviaturas de tipo	Describe las abreviaturas de tipo, que son nombres alternativos para los tipos.
Clases	Describe las clases, que son tipos que representan objetos que pueden tener propiedades, métodos y eventos.

TITLE	DESCRIPCIÓN
Estructuras	Describe las estructuras, que son tipos de objeto compactos que pueden ser más eficaces que una clase para los tipos que tienen una pequeña cantidad de datos y un comportamiento simple.
Interfaces	Describe las interfaces, que especifican conjuntos de miembros relacionados que otras clases implementan.
Clases abstractas	Describe las clases abstractas, que son clases que dejan algunos o todos los miembros sin implementar para que las clases derivadas puedan proporcionar las implementaciones.
Extensiones de tipo	Describe las extensiones de tipo, que permiten agregar nuevos miembros a un tipo de objeto definido previamente.
Tipos flexibles	Describe los tipos flexibles. Una anotación de tipo flexible es una indicación de que un parámetro, una variable o un valor tiene un tipo que es compatible con el tipo especificado, de tal forma que la compatibilidad viene determinada por la posición en una jerarquía orientada a objetos de clases o interfaces.
Delegados	Describe los delegados, que representan una llamada de función como un objeto.
Unidades de medida	Describe las unidades de medida. En F#, los valores de punto flotante pueden tener unidades de medida asociadas, que se suelen usar para indicar la longitud, el volumen, la masa, etc.
Proveedores de tipos	Describe los proveedores de tipos y proporciona vínculos a tutoriales sobre el uso de los proveedores de tipos integrados para tener acceso a bases de datos y servicios web.

Expresiones de F#

En la tabla siguiente se muestran los temas que describen las expresiones de F#.

TITLE	DESCRIPCIÓN
Expresiones condicionales: <code>if...then...else</code>	Describe la expresión <code>if...then...else</code> , que ejecuta diferentes ramas de código y también se evalúa como un valor distinto según la expresión booleana especificada.
Expresiones de coincidencia	Describe la expresión <code>match</code> , que proporciona control de rama basado en la comparación de una expresión con un conjunto de patrones.
Bucles: expresión <code>for...to</code>	Describe la expresión <code>for...to</code> , que se usa para recorrer en iteración un bucle sobre un intervalo de valores de una variable de bucle.

TITLE	DESCRIPCIÓN
Bucles: expresión <code>for...in</code>	Describe la expresión <code>for...in</code> , una construcción de bucle que se usa para recorrer en iteración las coincidencias de un patrón en una colección enumerable como una expresión de intervalo, una secuencia, una lista, una matriz u otra construcción que admita la enumeración.
Bucles: expresión <code>while...do</code>	Describe la expresión <code>while...do</code> , que se usa para realizar la ejecución iterativa (en bucle) mientras se cumple una condición de prueba especificada.
Expresiones de objeto	Describe las expresiones de objeto, que son las que crean nuevas instancias de un tipo de objeto anónimo creado dinámicamente que se basa en un tipo base, una interfaz o un conjunto de interfaces existente.
Expresiones diferidas	Describe las expresiones diferidas, que son cálculos que no se evalúan inmediatamente, pero se evalúan en su lugar cuando el resultado es realmente necesario.
Expresiones de cálculo	Describe las expresiones de cálculo en F#, que proporcionan una sintaxis adecuada para escribir cálculos que se pueden secuenciar y combinar mediante enlaces y construcciones de flujo de control. Se pueden usar para proporcionar una sintaxis apropiada para los <i>monads</i> , una característica de programación funcional que se puede usar para administrar datos, controles y efectos secundarios en programas funcionales. Uno de los tipos de expresión de cálculo, el flujo de trabajo asíncronico, proporciona compatibilidad con los cálculos asíncronos y en paralelo. Para más información, vea Flujos de trabajo asíncronos .
Flujos de trabajo asíncronos	Describe los flujos de trabajo asíncronos, una característica del lenguaje que permite escribir código asíncrono de manera muy similar a cómo se escribe código sincrónico.
Expresiones de código delimitadas	Describe las expresiones de código delimitadas, una característica del lenguaje que permite generar expresiones de código de F# y trabajar con ellas mediante programación.
Expresiones de consulta	Describe las expresiones de consulta, una característica del lenguaje que implementa LINQ para F# y permite escribir consultas para un origen de datos o una colección enumerable.

Construcciones admitidas por el compilador

En la tabla siguiente se muestran los temas que describen las construcciones especiales admitidas por el compilador.

TEMA	DESCRIPCIÓN
------	-------------

TEMA	DESCRIPCIÓN
Opciones del compilador	Describe las opciones de línea de comandos para el compilador de F#.
Directivas de compilador	Describe las directivas de procesador y las directivas de compilador.
Identificadores de línea, archivo y ruta de acceso de origen	Describe los identificadores <code>__LINE__</code> , <code>__SOURCE_DIRECTORY__</code> y <code>__SOURCE_FILE__</code> , que son valores integrados que permiten obtener acceso al número de línea y al nombre de directorio y archivo de origen en el código.

Referencia de palabras clave

27/03/2020 • 15 minutes to read • [Edit Online](#)

En este tema se proporcionan vínculos a información acerca de todas las palabras clave del lenguaje F.

Tabla de palabras clave de F

En la tabla siguiente se muestran todas las palabras clave de F en orden alfabético, junto con breves descripciones y vínculos a temas relevantes que contienen más información.

PALABRA CLAVE	VÍNCULO	DESCRIPCIÓN
<code>abstract</code>	Miembros Clases abstractas	Indica un método que no tiene ninguna implementación en el tipo en el que se declara o que es virtual y tiene una implementación predeterminada.
<code>and</code>	let Enlaces Registros Miembros Restricciones	Se utiliza en enlaces y registros mutuamente recursivos, en declaraciones de propiedad y con varias restricciones en parámetros genéricos.
<code>as</code>	Clases Coincidencia de modelos	Se utiliza para asignar al objeto de clase actual un nombre de objeto. También se utiliza para dar un nombre a un patrón completo dentro de una coincidencia de patrón.
<code>assert</code>	Aserciones	Se utiliza para comprobar el código durante la depuración.
<code>base</code>	Clases Herencia	Se utiliza como el nombre del objeto de clase base.
<code>begin</code>	Sintaxis detallada	En la sintaxis detallada, indica el inicio de un bloque de código.
<code>class</code>	Clases	En la sintaxis detallada, indica el inicio de una definición de clase.
<code>default</code>	Miembros	Indica una implementación de un método abstracto; junto con una declaración de método abstracto para crear un método virtual.
<code>delegate</code>	Delegados	Se utiliza para declarar un delegado.

PALABRA CLAVE	VÍNCULO	DESCRIPCIÓN
<code>do</code>	do (Enlaces) Bucles: expresión <code>for...to</code> Bucles: expresión <code>for...in</code> Bucles: expresión <code>while...do</code>	Se utiliza en construcciones de bucle o para ejecutar código imperativo.
<code>done</code>	Sintaxis detallada	En la sintaxis detallada, indica el final de un bloque de código en una expresión de bucle.
<code>downcast</code>	Conversiones	Se utiliza para convertir a un tipo que es inferior en la cadena de herencia.
<code>downto</code>	Bucles: expresión <code>for...to</code>	En <code>for</code> una expresión, se utiliza al contar en sentido inverso.
<code>elif</code>	Expresiones condicionales: <code>if...then...else</code>	Se utiliza en la bifurcación condicional. Una forma <code>else if</code> corta de .
<code>else</code>	Expresiones condicionales: <code>if...then...else</code>	Se utiliza en la bifurcación condicional.
<code>end</code>	Estructuras Uniones discriminadas Registros Extensiones de tipo Sintaxis detallada	En las definiciones de tipo y las extensiones de tipo, indica el final de una sección de definiciones de miembro. En la sintaxis detallada, se usa para especificar <code>begin</code> el final de un bloque de código que comienza con la palabra clave.
<code>exception</code>	Control de excepciones Tipos de excepción	Se utiliza para declarar un tipo de excepción.
<code>extern</code>	Funciones externas	Indica que un elemento de programa declarado se define en otro binario o ensamblado.
<code>false</code>	Tipos primitivos	Se utiliza como literal booleano.
<code>finally</code>	Excepciones: <code>try...finally</code> La expresión	Se utiliza <code>try</code> junto con para introducir un bloque de código que se ejecuta independientemente de si se produce una excepción.
<code>fixed</code>	Corregido	Se utiliza para "pin" un puntero en la pila para evitar que se recopile la basura.

PALABRA CLAVE	VÍNCULO	DESCRIPCIÓN
<code>for</code>	Bucles: expresión <code>for...to</code> Bucles: expresión <code>for...in</code>	Se utiliza en construcciones de bucle.
<code>fun</code>	Expresiones lambda: la <code>fun</code> palabra clave	Se utiliza en expresiones lambda, también conocidas como funciones anónimas.
<code>function</code>	Expresiones de coincidencia Expresiones Lambda: La palabra clave divertida	Se utiliza como una <code>fun</code> alternativa más <code>match</code> corta a la palabra clave y una expresión en una expresión lambda que tiene coincidencia de patrones en un único argumento.
<code>global</code>	Espacios de nombres	Se utiliza para hacer referencia al espacio de nombres de .NET de nivel superior.
<code>if</code>	Expresiones condicionales: <code>if...then...else</code>	Se utiliza en construcciones de bifurcación condicional.
<code>in</code>	Bucles: expresión <code>for...in</code> Sintaxis detallada	Se utiliza para expresiones de secuencia y, en sintaxis detallada, para separar expresiones de enlaces.
<code>inherit</code>	Herencia	Se utiliza para especificar una clase base o una interfaz base.
<code>inline</code>	Funciones Funciones insertadas	Se utiliza para indicar una función que debe integrarse directamente en el código del autor de la llamada.
<code>interface</code>	Interfaces	Se utiliza para declarar e implementar interfaces.
<code>internal</code>	Control de acceso	Se utiliza para especificar que un miembro está visible dentro de un ensamblado, pero no fuera de él.
<code>lazy</code>	Expresiones diferidas	Se utiliza para especificar una expresión que se va a realizar solo cuando se necesita un resultado.
<code>let</code>	let Enlaces	Se utiliza para asociar o enlazar un nombre a un valor o función.
<code>let!</code>	Flujos de trabajo asincrónicos Expresiones de cálculo	Se utiliza en flujos de trabajo asincrónicos para enlazar un nombre al resultado de un cálculo asincrónico o, en otras expresiones de cálculo, se usa para enlazar un nombre a un resultado, que es del tipo de cálculo.
<code>match</code>	Expresiones de coincidencia	Se utiliza para bifurcar comparando un valor con un patrón.

PALABRA CLAVE	VÍNCULO	DESCRIPCIÓN
<code>match!</code>	Expresiones de cálculo	Se utiliza para inlinear una llamada a una expresión de cálculo y coincidencia de patrón en su resultado.
<code>member</code>	Miembros	Se utiliza para declarar una propiedad o método en un tipo de objeto.
<code>module</code>	Módulos	Se utiliza para asociar un nombre a un grupo de tipos, valores y funciones relacionados para separarlo lógicamente de otro código.
<code>mutable</code>	Enlaces let	Se utiliza para declarar una variable, es decir, un valor que se puede cambiar.
<code>namespace</code>	Espacios de nombres	Se utiliza para asociar un nombre a un grupo de tipos y módulos relacionados, para separarlo lógicamente de otro código.
<code>new</code>	Constructores Restricciones	<p>Se utiliza para declarar, definir o invocar un constructor que crea o puede crear un objeto.</p> <p>También se utiliza en restricciones de parámetros genéricos para indicar que un tipo debe tener un constructor determinado.</p>
<code>not</code>	Referencia de símbolos y operadores Restricciones	En realidad no es una palabra clave. Sin <code>not struct</code> embargo, en combinación se utiliza como una restricción de parámetro genérico.
<code>null</code>	Valores NULL Restricciones	<p>Indica la ausencia de un objeto.</p> <p>También se utiliza en restricciones de parámetros genéricos.</p>
<code>of</code>	Uniones discriminadas Delegados Tipos de excepción	Se utiliza en uniones discriminadas para indicar el tipo de categorías de valores y en las declaraciones de delegado y excepción.
<code>open</code>	Declaraciones de importación: la palabra clave <code>open</code>	Se utiliza para hacer que el contenido de un espacio de nombres o módulo esté disponible sin calificación.
<code>or</code>	Referencia de símbolos y operadores Restricciones	<p>Se utiliza con condiciones <code>or</code> booleanas como un operador booleano. Equivalente a <code> </code>.</p> <p>También se utiliza en restricciones de miembro.</p>

PALABRA CLAVE	VÍNCULO	DESCRIPCIÓN
<code>override</code>	Miembros	Se utiliza para implementar una versión de un método abstracto o virtual que difiere de la versión base.
<code>private</code>	Control de acceso	Restringe el acceso a un miembro al código del mismo tipo o módulo.
<code>public</code>	Control de acceso	Permite el acceso a un miembro desde fuera del tipo.
<code>rec</code>	Funciones	Se utiliza para indicar que una función es recursiva.
<code>return</code>	Flujos de trabajo asincrónicos Expresiones de cálculo	Se utiliza para indicar un valor que se va a proporcionar como resultado de una expresión de cálculo.
<code>return!</code>	Expresiones de cálculo Flujos de trabajo asincrónicos	Se utiliza para indicar una expresión de cálculo que, cuando se evalúa, proporciona el resultado de la expresión de cálculo contenedora.
<code>select</code>	Expresiones de consulta	Se utiliza en expresiones de consulta para especificar qué campos o columnas se van a extraer. Tenga en cuenta que se trata de una palabra clave contextual, lo que significa que en realidad no es una palabra reservada y solo actúa como una palabra clave en el contexto adecuado.
<code>static</code>	Miembros	Se utiliza para indicar un método o propiedad que se puede llamar sin una instancia de un tipo o un miembro de valor que se comparte entre todas las instancias de un tipo.
<code>struct</code>	Estructuras Tuplas Restricciones	Se utiliza para declarar un tipo de estructura. Se utiliza para especificar una tupla struct. También se utiliza en restricciones de parámetros genéricos. Se utiliza para la compatibilidad con OCaml en las definiciones de módulo.
<code>then</code>	Expresiones condicionales: <code>if...then...else</code> Constructores	Se utiliza en expresiones condicionales. También se utiliza para realizar efectos secundarios después de la construcción del objeto.
<code>to</code>	Bucles: expresión <code>for...to</code>	Se <code>for</code> utiliza en bucles para indicar un rango.

PALABRA CLAVE	VÍNCULO	DESCRIPCIÓN
<code>true</code>	Tipos primitivos	Se utiliza como literal booleano.
<code>try</code>	Excepciones: expresión try...with Excepciones: expresión try...finally	Se utiliza para introducir un bloque de código que podría generar una excepción. Utilizado junto <code>with</code> <code>finally</code> con o .
<code>type</code>	Tipos en F# Clases Registros Estructuras Enumeraciones Uniones discriminadas Abreviaturas de tipo Unidades de medida	Se utiliza para declarar una clase, registro, estructura, unión discriminada, tipo de enumeración, unidad de medida o abreviatura de tipo.
<code>upcast</code>	Conversiones	Se utiliza para convertir a un tipo que es superior en la cadena de herencia.
<code>use</code>	Administración de <code>use</code> recursos: La palabra clave	Se utiliza <code>let</code> en lugar <code>Dispose</code> de para los valores que requieren ser llamados para liberar recursos.
<code>use!</code>	Expresiones de cálculo Flujos de trabajo asincrónicos	Se utiliza <code>let!</code> en lugar de en flujos <code>Dispose</code> de trabajo asincrónicos y otras expresiones de cálculo para los valores que requieren ser llamados para liberar recursos.
<code>val</code>	Campos explícitos: la <code>val</code> palabra clave Firmas Miembros	Se utiliza en una firma para indicar un valor, o en un tipo para declarar un miembro, en situaciones limitadas.
<code>void</code>	Tipos primitivos	Indica el tipo <code>void</code> .NET. Se utiliza al interoperar con otros lenguajes .NET.
<code>when</code>	Restricciones	Se utiliza para condiciones booleanas (<i>cuando protege</i>) en coincidencias de patrones y para introducir una cláusula de restricción para un parámetro de tipo genérico.
<code>while</code>	Bucles: expresión <code>while...do</code>	Presenta una construcción de bucle.

PALABRA CLAVE	VÍNCULO	DESCRIPCIÓN
<code>with</code>	Expresiones de coincidencia Expresiones de objeto Expresiones de registro de copia y actualización Extensiones de tipo Excepciones: <code>try...with</code> La expresión	Se utiliza <code>match</code> junto con la palabra clave en expresiones de coincidencia de patrones. También se usa en expresiones de objeto, expresiones de copia de registros y extensiones de tipo para introducir definiciones de miembro e introducir controladores de excepciones.
<code>yield</code>	Listas, Matrices, Secuencias	Se utiliza en una expresión de lista, matriz o secuencia para generar un valor para una secuencia. Normalmente se puede omitir, ya que está implícito en la mayoría de las situaciones.
<code>yield!</code>	Expresiones de cálculo Flujos de trabajo asíncronos	Se utiliza en una expresión de cálculo para anexar el resultado de una expresión de cálculo determinada a una colección de resultados para la expresión de cálculo contenedora.

Los siguientes tokens se reservan en F- porque son palabras clave en el idioma OCaml:

- `asr`
- `land`
- `lor`
- `lsl`
- `lsr`
- `lxor`
- `mod`
- `sig`

Si utiliza `--mlcompatibility` la opción del compilador, las palabras clave anteriores están disponibles para su uso como identificadores.

Los siguientes tokens se reservan como palabras clave para la futura expansión del lenguaje F-

- `atomic`
- `break`
- `checked`
- `component`
- `const`
- `constraint`
- `constructor`
- `continue`
- `eager`
- `event`
- `external`
- `functor`
- `include`

- `method`
- `mixin`
- `object`
- `parallel`
- `process`
- `protected`
- `pure`
- `sealed`
- `tailcall`
- `trait`
- `virtual`
- `volatile`

Vea también

- [Referencia del lenguaje f](#)
- [Referencia de símbolos y operadores](#)
- [Opciones del compilador](#)

Referencia de símbolos y operadores

04/11/2019 • 18 minutes to read • [Edit Online](#)

NOTE

Los vínculos de la referencia de API de este artículo le llevarán a MSDN. La referencia de API de docs.microsoft.com no está completa.

En este tema se incluye una tabla de símbolos y operadores que se utilizan en el lenguaje F#.

Tabla de símbolos y operadores

En la siguiente tabla se describen los símbolos utilizados en el lenguaje F#, se incluyen vínculos a temas que proporcionan más información y se facilita una breve descripción de algunos de los usos del símbolo. Los símbolos están ordenados según el orden del juego de caracteres ASCII.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
!	Celdas de referencia Expresiones de cálculo	<ul style="list-style-type: none">Desreferencia una celda de referencia.Después de una palabra clave, indica una versión modificada del comportamiento de la misma, controlado por un flujo de trabajo.
!=	No disponible.	<ul style="list-style-type: none">No se utiliza en F#. Utilice <code><></code> para las operaciones de desigualdad.
"	Literales Cadenas	<ul style="list-style-type: none">Delimita una cadena de texto.
"""	Cadenas	Delimita una cadena de texto literal. Se diferencia de <code>@"..."</code> en que puede indicar un carácter de comilla mediante el uso de comillas simples en la cadena.
#	Directivas de compilador Tipos flexibles	<ul style="list-style-type: none">Prefija una directiva de compilador o preprocesador, como <code>#light</code>.Cuando se usa con un tipo, indica un <i>tipo flexible</i>, que hace referencia a un tipo o a cualquiera de sus tipos derivados.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
\$	No hay más información disponible.	<ul style="list-style-type: none"> Se utiliza internamente para determinados nombres de variable y función generados por el compilador.
%	Operadores aritméticos Expresiones de código delimitadas	<ul style="list-style-type: none"> Calcula el resto entero. Se utiliza para ensamblar expresiones en expresiones de código delimitadas con tipo.
%%	Expresiones de código delimitadas	<ul style="list-style-type: none"> Se utiliza para ensamblar expresiones en expresiones de código delimitadas sin tipo.
%?	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Calcula el resto entero, cuando el lado derecho es un tipo que acepta valores NULL.
&	Expresiones de coincidencia	<ul style="list-style-type: none"> Calcula la dirección de un valor mutable para usarlo al interoperar con otros lenguajes. Se utiliza en los patrones AND.
&&	Operadores booleanos	<ul style="list-style-type: none"> Calcula la operación AND booleana.
&&&	Operadores bit a bit	<ul style="list-style-type: none"> Calcula la operación AND bit a bit.
,	Literales Generalización automática	<ul style="list-style-type: none"> Delimita un literal de carácter único. Indica un parámetro de tipo genérico.
`...`	No hay más información disponible.	<ul style="list-style-type: none"> Delimita un identificador que de otro modo no sería un identificador válido, como una palabra clave de lenguaje.
()	Tipo unit	<ul style="list-style-type: none"> Representa el valor único del tipo de unidad.
(...)	Tuplas Sobrecarga de operadores	<ul style="list-style-type: none"> Indica el orden en que se evalúan las expresiones. Delimita una tupla. Se utiliza en las definiciones de operador.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
<code>(*...*)</code>		<ul style="list-style-type: none"> Delimita un comentario que podría abarcar varias líneas.
<code>(...)</code>	Patrones activos	<ul style="list-style-type: none"> Delimita un modelo activo. También se denominan <i>delimitadores de modelo activo</i>.
<code>*</code>	Operadores aritméticos Tuplas Unidades de medida	<ul style="list-style-type: none"> Cuando se utiliza como un operador binario, multiplica los lados izquierdo y derecho. En los tipos, indica el emparejamiento en una tupla. Se utiliza en unidades de medida de tipos.
<code>*?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Multiplica los lados izquierdo y derecho, si el lado derecho es un tipo que acepta valores NULL.
<code>**</code>	Operadores aritméticos	<ul style="list-style-type: none"> Calcula la operación de exponenciación (<code>x ** y</code> significa <code>x</code> elevado a la potencia de <code>y</code>).
<code>+</code>	Operadores aritméticos	<ul style="list-style-type: none"> Si se utiliza como un operador binario, suma los lados izquierdo y derecho. Cuando se utiliza como un operador unario, indica una cantidad positiva. (Formalmente, genera el mismo valor sin modificar el signo).
<code>+?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Suma los lados izquierdo y derecho, si el lado derecho es un tipo que acepta valores NULL.
<code>,</code>	Tuplas	<ul style="list-style-type: none"> Separa los elementos de una tupla o los parámetros de tipo.
<code>-</code>	Operadores aritméticos	<ul style="list-style-type: none"> Si se utiliza como un operador binario, resta el lado derecho del lado izquierdo. Si se utiliza como un operador unario, realiza una operación de negación.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
<code>-?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> • Resta el lado derecho del lado izquierdo, si el lado derecho es un tipo que acepta valores NULL.
<code>-></code>	Funciones Expresiones de coincidencia	<ul style="list-style-type: none"> • En tipos de función, delimita los argumentos y valores devueltos. • Genera una expresión (en expresiones de secuencia); equivalente a la palabra clave <code>yield</code>. • Utilizado en expresiones de coincidencia.
<code>.</code>	Miembros Tipos primitivos	<ul style="list-style-type: none"> • Tiene acceso a un miembro y separa los nombres individuales de un nombre completo. • Especifica un separador decimal en números de punto flotante.
<code>..</code>	Bucles: expresión <code>for...in</code>	<ul style="list-style-type: none"> • Especifica un intervalo.
<code>.. ..</code>	Bucles: expresión <code>for...in</code>	<ul style="list-style-type: none"> • Especifica un intervalo y un incremento.
<code>.[...]</code>	Matrices	<ul style="list-style-type: none"> • Tiene acceso a un elemento de matriz.
<code>/</code>	Operadores aritméticos Unidades de medida	<ul style="list-style-type: none"> • Divide el lado izquierdo (numerador) entre el lado derecho (denominador). • Se utiliza en unidades de medida de tipos.
<code>/?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> • Divide el lado izquierdo entre el lado derecho, si el lado derecho es un tipo que acepta valores NULL.
<code>//</code>		<ul style="list-style-type: none"> • Indica el principio de una sola línea de comentario.
<code>///</code>	Documentación de XML	<ul style="list-style-type: none"> • Indica un comentario XML.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
:	Funciones	<ul style="list-style-type: none"> En una anotación de tipo, separa un parámetro o nombre de miembro de su tipo.
::	Listas Expresiones de coincidencia	<ul style="list-style-type: none"> Crea una lista. El elemento del lado izquierdo antecede a la lista en el lado derecho. Se utiliza en la coincidencia de patrones para separar las partes de una lista.
:=	Celdas de referencia	<ul style="list-style-type: none"> Asigna un valor a una celda de referencia.
:	Conversiones	<ul style="list-style-type: none"> Convierte un tipo en otro que está más arriba en la jerarquía.
?:	Expresiones de coincidencia	<ul style="list-style-type: none"> Devuelve <code>true</code> si el valor coincide con el tipo especificado (incluido si es un subtipo); de lo contrario, devuelve <code>false</code> (operador de prueba de tipo).
?:>	Conversiones	<ul style="list-style-type: none"> Convierte un tipo en otro que está por debajo en la jerarquía.
;	Sintaxis detallada Listas Registros	<ul style="list-style-type: none"> Separa expresiones (se usa principalmente en sintaxis detallada). Separa elementos de una lista. Separa los campos de un registro.
<	Operadores aritméticos	<ul style="list-style-type: none"> Calcula la operación “menor que”.
<?	Operadores que aceptan valores NULL	Calcula la operación “menor que”, si el lado derecho es un tipo que acepta valores NULL.
<<	Funciones	<ul style="list-style-type: none"> Compone dos funciones en orden inverso; la segunda se ejecuta primero (operador de composición hacia atrás).

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
<<<	Operadores bit a bit	<ul style="list-style-type: none"> Desplaza hacia la izquierda los bits de la cantidad del lado izquierdo, y lo hace en el número de bits especificado en el lado derecho.
<-	Valores	<ul style="list-style-type: none"> Asigna un valor a una variable.
<...>	Generalización automática	<ul style="list-style-type: none"> Delimita los parámetros de tipo.
<>	Operadores aritméticos	<ul style="list-style-type: none"> Devuelve <code>true</code> si el lado izquierdo no es igual que el lado derecho; de lo contrario, devuelve <code>false</code>.
<>?	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Calcula la operación “no es igual a” cuando el lado derecho es un tipo que acepta valores NULL.
<=	Operadores aritméticos	<ul style="list-style-type: none"> Devuelve <code>true</code> si el lado izquierdo es menor o igual que el lado derecho; de lo contrario, devuelve <code>false</code>.
<=?	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Calcula la operación “menor o igual que” cuando el lado derecho es un tipo que acepta valores NULL.
<	Funciones	<ul style="list-style-type: none"> Pasa el resultado de la expresión de la derecha a la función de la izquierda (operador de canalización hacia atrás).
<	Operadores.(<)<'T1','T2','U'> Función	<ul style="list-style-type: none"> Pasa la tupla de dos argumentos del lado derecho a la función del lado izquierdo.
<	Operadores.(<)<'T1','T2','T3','U'> Función	<ul style="list-style-type: none"> Pasa la tupla de tres argumentos del lado derecho a la función del lado izquierdo.
<@...@>	Expresiones de código delimitadas	<ul style="list-style-type: none"> Delimita una expresión de código con tipos.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
<code><@...@></code>	Expresiones de código delimitadas	<ul style="list-style-type: none"> Delimita una expresión de código sin tipos.
<code>=</code>	Operadores aritméticos	<ul style="list-style-type: none"> Devuelve <code>true</code> si el lado izquierdo es igual que el lado derecho; de lo contrario, devuelve <code>false</code>.
<code>=?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Calcula la operación "es igual a" cuando el lado derecho es un tipo que acepta valores NULL.
<code>==</code>	No disponible.	<ul style="list-style-type: none"> No se utiliza en F#. Utilice <code>=</code> para operaciones de igualdad.
<code>></code>	Operadores aritméticos	<ul style="list-style-type: none"> Devuelve <code>true</code> si el lado izquierdo es mayor que el lado derecho; de lo contrario, devuelve <code>false</code>.
<code>>?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Calcula la operación "mayor que" cuando el lado derecho es un tipo que acepta valores NULL.
<code>>></code>	Funciones	<ul style="list-style-type: none"> Compone dos funciones (operador de composición hacia delante).
<code>>>></code>	Operadores bit a bit	<ul style="list-style-type: none"> Desplaza hacia la derecha los bits de la cantidad del lado izquierdo, y lo hace en el número de posiciones especificado en el lado derecho.
<code>>=</code>	Operadores aritméticos	<ul style="list-style-type: none"> Devuelve <code>true</code> si el lado izquierdo es mayor o igual que el lado derecho; de lo contrario, devuelve <code>false</code>.
<code>>=?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Calcula la operación "mayor o igual que" cuando el lado derecho es un tipo que acepta valores NULL.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
<code>?</code>	Parámetros y argumentos	<ul style="list-style-type: none"> Especifica un argumento opcional. Se usa como operador para método dinámico y llamadas de propiedad. Debe proporcionar su propia implementación.
<code>? ... <- ...</code>	No hay más información disponible.	<ul style="list-style-type: none"> Se usa como operador para establecer propiedades dinámicas. Debe proporcionar su propia implementación.
<code>?>=</code> , <code>?></code> , <code>?<=</code> , <code>?<</code> , <code>?=</code> , <code>?<></code> , <code>?+</code> , <code>?-</code> , <code>?*</code> , <code>?/</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Equivalente a los operadores correspondientes sin el sufijo ?; en la izquierda hay un tipo que acepta valores NULL.
<code>>=?</code> , <code>>?</code> , <code><=?</code> , <code><?</code> , <code>=?</code> , <code><>?</code> , <code>+?</code> , <code>-?</code> , <code>*?</code> , <code>/?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Equivalente a los operadores correspondientes sin el sufijo ?; en la derecha hay un tipo que acepta valores NULL.
<code>?>=?</code> , <code>?>?</code> , <code>?<=?</code> , <code>?<?</code> , <code>?=?</code> , <code>?<>?</code> , <code>?+?</code> , <code>?-?</code> , <code>?*?</code> , <code>?/?</code>	Operadores que aceptan valores NULL	<ul style="list-style-type: none"> Equivalente a los operadores correspondientes sin los signos de interrogación circundantes, donde ambos lados son tipos que aceptan valores NULL.
<code>@</code>	Listas Cadenas	<ul style="list-style-type: none"> Concatena dos listas. Cuando se coloca delante de un literal de cadena, indica que la cadena debe interpretarse literalmente, sin interpretación alguna de los caracteres de escape.
<code>[...]</code>	Listas	<ul style="list-style-type: none"> Delimita los elementos de una lista.
<code>[...]</code>	Matrices	<ul style="list-style-type: none"> Delimita los elementos de una matriz.
<code>[<...>]</code>	Atributos	<ul style="list-style-type: none"> Delimita un atributo.
<code>\</code>	Cadenas	<ul style="list-style-type: none"> Produce el escape del carácter siguiente; se utiliza en los literales de carácter y de cadena.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
<code>^</code>	Parámetros de tipo resueltos estáticamente Cadenas	<ul style="list-style-type: none"> Especifica los parámetros de tipo que deben resolverse en tiempo de compilación, no en runtime. Concatena cadenas.
<code>^^^</code>	Operadores bit a bit	<ul style="list-style-type: none"> Calcula la operación OR exclusiva bit a bit.
<code>_</code>	Expresiones de coincidencia Genéricos	<ul style="list-style-type: none"> Indica un patrón de carácter comodín. Especifica un parámetro genérico anónimo.
<code>`</code>	Generalización automática	<ul style="list-style-type: none"> Se utiliza internamente para indicar un parámetro de tipo genérico.
<code>{...}</code>	Secuencias Registros	<ul style="list-style-type: none"> Delimita expresiones de secuencia y expresiones de cálculo. Se utiliza en definiciones de registro.
<code> </code>	Expresiones de coincidencia	<ul style="list-style-type: none"> Delimita casos de coincidencia individuales, casos de unión discriminada individuales y valores de la enumeración.
<code> </code>	Operadores booleanos	<ul style="list-style-type: none"> Calcula la operación OR booleana.
<code> </code>	Operadores bit a bit	<ul style="list-style-type: none"> Calcula la operación OR bit a bit.
<code> ></code>	Funciones	<ul style="list-style-type: none"> Pasa el resultado del lado izquierdo a la función del lado derecho (operador de canalización hacia delante).
<code> ></code>	Operadores.(>)<'T1','T2','U'> Función	<ul style="list-style-type: none"> Pasa la tupla de dos argumentos del lado izquierdo a la función del lado derecho.
<code> ></code>	Operadores.(>)<'T1','T2','T3','U'> Función	<ul style="list-style-type: none"> Pasa la tupla de tres argumentos del lado izquierdo a la función del lado derecho.

SÍMBOLO U OPERADOR	VÍNCULOS	DESCRIPCIÓN
<code>~~</code>	Sobrecarga de operadores	<ul style="list-style-type: none"> Se utiliza para declarar una sobrecarga para el operador unario de negación.
<code>~~~</code>	Operadores bit a bit	<ul style="list-style-type: none"> Calcula la operación NOT bit a bit.
<code>~-</code>	Sobrecarga de operadores	<ul style="list-style-type: none"> Se utiliza para declarar una sobrecarga para el operador unario menos.
<code>~+</code>	Sobrecarga de operadores	<ul style="list-style-type: none"> Se utiliza para declarar una sobrecarga para el operador unario más.

Precedencia de operadores

En la siguiente tabla se muestra, de menor a mayor, el orden de precedencia de operadores y otras palabras clave de expresiones del lenguaje F#. También se indica la asociatividad, si corresponde.

"?"	ASOCIATIVIDAD
<code>as</code>	Derecho
<code>when</code>	Derecho
<code> </code> (canalización)	Izquierdo
<code>;</code>	Derecho
<code>let</code>	No asociativo
<code>function</code> , <code>fun</code> , <code>match</code> , <code>try</code>	No asociativo
<code>if</code>	No asociativo
<code>not</code>	Derecho
<code>-></code>	Derecho
<code>:=</code>	Derecho
<code>,</code>	No asociativo
<code>or</code> , <code> </code>	Izquierdo
<code>&</code> , <code>&&</code>	Izquierdo

"??"	ASOCIATIVIDAD
:>, :?>	Derecho
< OP, > OP, =, op, & OP, & (incluyendo <<<, >>>, , &&&)	Izquierdo
^ op (incluyendo ^^^)	Derecho
::	Derecho
:?	No asociativo
- op, + op	Se aplica a los usos de infijo de estos símbolos
* op, / op, % op	Izquierdo
** op	Derecho
f x (aplicación de función)	Izquierdo
(coincidencia de patrones)	Derecho
operadores de prefijo (+ op, - op, %, %% , & , && , ! op, ~ op)	Izquierdo
.	Izquierdo
f(x)	Izquierdo
f< tipos >	Izquierdo

F# es compatible con la sobrecarga de operadores personalizados. Esto significa que puede definir sus propios operadores. En la tabla anterior, *op* puede ser cualquier secuencia válida (posiblemente vacía) de caracteres de operador, ya sean integrados o definidos por el usuario. Por lo tanto, puede utilizar esta tabla para determinar qué secuencia de caracteres se utilizará para que un operador personalizado logre alcanzar el nivel deseado de precedencia. Los caracteres `.` iniciales se omiten cuando el compilador determina la precedencia.

Vea también

- [Referencia del lenguaje F#](#)
- [Sobrecarga de operadores](#)

Operadores aritméticos

23/10/2019 • 7 minutes to read • [Edit Online](#)

En este tema se describen los operadores aritméticos que F# están disponibles en el lenguaje.

Resumen de operadores aritméticos binarios

En la tabla siguiente se resumen los operadores aritméticos binarios que están disponibles para los tipos enteros y de punto flotante con conversión unboxing.

OPERADOR BINARIO	NOTAS
<code>+</code> (suma, más)	Desactivada. Posible condición de desbordamiento cuando se suman números y la suma supera el valor absoluto máximo admitido por el tipo.
<code>-</code> (resta, menos)	Desactivada. Posible condición de subdesbordamiento cuando se restan tipos sin signo o cuando los valores de punto flotante son demasiado pequeños para representarlos mediante el tipo.
<code>*</code> (multiplicación, veces)	Desactivada. Posible condición de desbordamiento cuando se multiplican los números y el producto supera el valor absoluto máximo admitido por el tipo.
<code>/</code> (división, dividido por)	La división por cero produce DivideByZeroException para los tipos enteros. En el caso de los tipos de punto flotante, la división por cero le proporciona los valores <code>+Infinity</code> o <code>-Infinity</code> de punto flotante especiales. También hay una posible condición de subdesbordamiento cuando un número de punto flotante es demasiado pequeño para representarlo mediante el tipo.
<code>%</code> (resto, REM)	Devuelve el resto de una operación de división. El signo del resultado es el mismo que el del primer operando.
<code>**</code> (exponenciación, a la potencia de)	Posible condición de desbordamiento cuando el resultado supera el valor absoluto máximo para el tipo. El operador de exponenciación solo funciona con tipos de punto flotante.

Resumen de operadores aritméticos unarios

En la tabla siguiente se resumen los operadores aritméticos unarios que están disponibles para los tipos enteros y de punto flotante.

UNARIO (OPERADOR)	NOTAS
<code>+</code> positivo	Se puede aplicar a cualquier expresión aritmética. No cambia el signo del valor.

UNARIO (OPERADOR)	NOTAS
<code>-</code> (negación, negativo)	Se puede aplicar a cualquier expresión aritmética. Cambia el signo del valor.

El comportamiento en caso de desbordamiento o subdesbordamiento para los tipos enteros es ajustar. Esto produce un resultado incorrecto. El desbordamiento de enteros es un problema potencialmente grave que puede contribuir a problemas de seguridad cuando no se escribe el software en su cuenta. Si esto supone un problema para su aplicación, considere la posibilidad de usar los `Microsoft.FSharp.Core.Operators.Checked` operadores comprobados en.

Resumen de operadores de comparación binaria

En la tabla siguiente se muestran los operadores de comparación binarios que están disponibles para los tipos enteros y de punto flotante. Estos operadores devuelven valores `bool` de tipo.

Los números de punto flotante nunca deben compararse directamente para comprobar si son iguales, porque la representación de punto flotante de IEEE no admite una operación de igualdad exacta. Dos números que se pueden comprobar fácilmente para ser iguales mediante la inspección del código puede tener realmente representaciones de bits diferentes.

OPERADOR	NOTAS
<code>=</code> (igualdad, es igual a)	No es un operador de asignación. Se usa solo para la comparación. Este es un operador genérico.
<code>></code> (mayor que)	Este es un operador genérico.
<code><</code> (menor que)	Este es un operador genérico.
<code>>=</code> (mayor o igual que)	Este es un operador genérico.
<code><=</code> (menor o igual que)	Este es un operador genérico.
<code><></code> (no igual)	Este es un operador genérico.

Operadores genéricos y sobrecargados

Todos los operadores descritos en este tema se definen en el espacio de nombres **Microsoft.FSharp.Core.Operators**. Algunos de los operadores se definen mediante parámetros de tipo resueltos estáticamente. Esto significa que hay definiciones individuales para cada tipo específico que funciona con ese operador. Todos los operadores aritméticos y bit a bit binarios y binarios se encuentran en esta categoría. Los operadores de comparación son genéricos y, por tanto, funcionan con cualquier tipo, no solo con tipos aritméticos primitivos. Los tipos de registro y Unión discriminada tienen sus propias implementaciones personalizadas generadas por F# el compilador. Los tipos de clase usan [Equals](#) el método.

Los operadores genéricos son personalizables. Para personalizar las funciones de comparación, [Equals](#) invalide para proporcionar su propia comparación de igualdad personalizada [IComparable](#), a continuación, implemente. La [System.IComparable](#) interfaz tiene un método único, el [CompareTo](#) método.

Operadores e inferencia de tipos

El uso de un operador en una expresión restringe la inferencia de tipos en dicho operador. Además, el uso de

operadores evita la generalización automática, porque el uso de operadores implica un tipo aritmético. En ausencia de otra información, el F# compilador deduce `int` como el tipo de expresiones aritméticas. Puede invalidar este comportamiento especificando otro tipo. Por lo tanto `int`, los tipos de argumento `function1` y el tipo de valor devuelto de en el código siguiente se deducen como, pero `float` los tipos de `function2` se deducen como.

```
// x, y and return value inferred to be int
// function1: int -> int -> int
let function1 x y = x + y

// x, y and return value inferred to be float
// function2: float -> float -> float
let function2 (x: float) y = x + y
```

Vea también

- [Referencia de símbolos y operadores](#)
- [Sobrecarga de operadores](#)
- [Operadores bit a bit](#)
- [Operadores booleanos](#)

Operadores booleanos

23/10/2019 • 2 minutes to read • [Edit Online](#)

Este tema describe la compatibilidad con los operadores booleanos en la F# lenguaje.

Resumen de operadores booleanos

En la tabla siguiente se resume los operadores booleanos que están disponibles en el F# lenguaje. Es el único tipo admitido por estos operadores el `bool` tipo.

OPERADOR	DESCRIPCIÓN
<code>not</code>	Negación booleana
<code> </code>	OR booleano
<code>&&</code>	AND Boolean

Realizan el booleano operadores AND y OR *la evaluación de cortocircuito*, es decir, evalúa la expresión a la derecha del operador solo cuando sea necesario determinar el resultado global de la expresión. La segunda expresión de la `&&` operador se evalúa solo si la primera expresión se evalúa como `true`; la segunda expresión de la `||` operador se evalúa solo si la primera expresión se evalúa como `false`.

Vea también

- [Operadores bit a bit](#)
- [Operadores aritméticos](#)
- [Referencia de símbolos y operadores](#)

Operadores bit a bit

23/10/2019 • 3 minutes to read • [Edit Online](#)

En este tema se describe los operadores bit a bit que están disponibles en el F# lenguaje.

Resumen de los operadores bit a bit

En la tabla siguiente se describe los operadores bit a bit que se admiten para los tipos enteros con conversión unboxing en los F# lenguaje.

OPERADOR	NOTAS
<code>&&&</code>	Operador AND bit a bit. Bits del resultado tienen el valor 1 si y solo si los bits correspondientes de ambos operandos de origen son 1.
<code> </code>	Operador OR bit a bit. Bits del resultado tienen el valor 1 si alguno de los bits correspondientes en el origen de los operandos son 1.
<code>^^^</code>	Bit a bit operador OR exclusivo. Bits del resultado tienen el valor 1 si y solo si los bits en los operandos de origen tienen valores distintos.
<code>~~~</code>	Operador de negación bit a bit. Esto es un operador unario y genera un resultado en el que todos los bits 0 en el operando de origen se convierten en bits 1 y todos los bits 1 se convierten en bits 0.
<code><<<</code>	Bit a bit el operador de desplazamiento a la izquierda. El resultado es el primer operando con bits desplazado a la izquierda el número de bits en el segundo operando. No se giran bits desplazados fuera de la posición más significativa en el menos significativo. Los bits menos significativos se rellenan con ceros. El tipo del segundo argumento es <code>int32</code> .
<code>>>></code>	Bit a bit el operador de desplazamiento a la derecha. El resultado es el primer operando con bits desplazado a la derecha el número de bits en el segundo operando. No se giran bits desplazados el menos significativo en la posición más significativa. Para los tipos sin signo, los bits más significativos se rellenan con ceros. Para los tipos con signo con valores negativos, los bits más significativos se rellenan con los. El tipo del segundo argumento es <code>int32</code> .

Los tipos siguientes pueden utilizarse con operadores bit a bit: `byte`, `sbyte`, `int16`, `uint16`, `int32 (int)`, `uint32`, `int64`, `uint64`, `nativeint`, y `unativeint`.

Vea también

- [Referencia de símbolos y operadores](#)
- [Operadores aritméticos](#)

- [Operadores booleanos](#)

Operadores que aceptan valores NULL

04/11/2019 • 6 minutes to read • [Edit Online](#)

Los operadores que aceptan valores NULL son operadores aritméticos o de comparación binarios que funcionan con tipos aritméticos que aceptan valores NULL en uno o ambos lados. Los tipos que aceptan valores NULL se producen con frecuencia cuando se trabaja con datos de orígenes como bases de datos que permiten valores NULL en lugar de valores reales. Los operadores que aceptan valores NULL se utilizan con frecuencia en expresiones de consulta. Además de los operadores que aceptan valores NULL para operaciones aritméticas y de comparación, los operadores de conversión se pueden usar para realizar conversiones entre tipos que aceptan valores NULL. También hay versiones que aceptan valores NULL de ciertos operadores de consulta.

Tabla de operadores que aceptan valores NULL

En la tabla siguiente se enumeran los operadores que F# aceptan valores NULL que se admiten en el lenguaje.

ACEPTA VALORES NULL A LA IZQUIERDA	ACEPTA VALORES NULL A LA DERECHA	AMBOS LADOS ACEPTAN VALORES NULL
? > =	> =?	? > =?
? >	? >?	? >?
? < =	< =?	? < =?
? <	? <?	? <?
? =	=?	? =?
? < >	< >?	? < >?
? +	+?	? +?
? -	-?	? -?
? *	*?	? *?
? /	/?	? /?
? %	%?	? %?

Comentarios

Los operadores que aceptan valores NULL se incluyen en el módulo [NullableOperators](#) en el espacio de nombres [Microsoft.FSharp.Linq](#). El tipo de los datos que aceptan valores NULL es `System.Nullable<'T>`.

En las expresiones de consulta, los tipos que aceptan valores NULL se producen cuando se seleccionan datos de un origen de datos que permite valores NULL en lugar de valores. En una base de datos de SQL Server, cada columna de datos de una tabla tiene un atributo que indica si se permiten valores NULL. Si se permiten valores NULL, los datos devueltos por la base de datos pueden contener valores NULL que no pueden representarse mediante un tipo de datos primitivo como `int`, `float`, etc. Por lo tanto, los datos se devuelven como un

`System.Nullable<int>` en lugar de `int` y `System.Nullable<float>` en lugar de `float`. El valor real se puede obtener de un objeto `System.Nullable<T>` mediante la propiedad `Value`, y puede determinar si un objeto `System.Nullable<T>` tiene un valor llamando al método `HasValue`. Otro método útil es el método `System.Nullable<T>.GetValueOrDefault`, que permite obtener el valor o un valor predeterminado del tipo adecuado. El valor predeterminado es alguna forma de valor "cero", como 0, 0,0 o `false`.

Los tipos que aceptan valores NULL se pueden convertir en tipos primitivos que no aceptan valores NULL mediante los operadores de conversión habituales, como `int` o `float`. También es posible convertir de un tipo que acepta valores NULL a otro tipo que acepta valores NULL mediante los operadores de conversión para tipos que aceptan valores NULL. Los operadores de conversión adecuados tienen el mismo nombre que los estándar, pero se encuentran en un módulo independiente, el módulo que [acepta valores NULL](#) en el espacio de nombres [Microsoft.FSharp.Linq](#). Normalmente, se abre este espacio de nombres cuando se trabaja con expresiones de consulta. En ese caso, puede usar los operadores de conversión que aceptan valores NULL agregando el prefijo `Nullable.` al operador de conversión adecuado, tal y como se muestra en el código siguiente.

```
open Microsoft.FSharp.Linq

let nullableInt = new System.Nullable<int>(10)

// Use the Nullable.float conversion operator to convert from one nullable type to another nullable type.
let nullableFloat = Nullable.float nullableInt

// Use the regular non-nullable float operator to convert to a non-nullable float.
printfn "%f" (float nullableFloat)
```

El resultado es `10.000000`

Los operadores de consulta en campos de datos que aceptan valores NULL, como `sumByNullable`, también existen para su uso en expresiones de consulta. Los operadores de consulta para tipos que no aceptan valores NULL no son compatibles con tipos que aceptan valores NULL, por lo que debe usar la versión que acepta valores NULL del operador de consulta adecuado cuando se trabaja con valores de datos que aceptan valores NULL. Para obtener más información, vea [expresiones de consulta](#).

En el ejemplo siguiente se muestra el uso de operadores que aceptan F# valores NULL en una expresión de consulta. La primera consulta muestra cómo se escribiría una consulta sin un operador que acepta valores NULL; la segunda consulta muestra una consulta equivalente que utiliza un operador que acepta valores NULL. En el contexto completo, incluido cómo configurar la base de datos para usar este código de ejemplo, vea [Tutorial: acceso a un SQL Database mediante proveedores de tipos](#).


```

open System
open System.Data
open System.Data.Linq
open Microsoft.FSharp.Data.TypeProviders
open Microsoft.FSharp.Linq

[<Generate>]
type dbSchema = SqlConnection<"Data Source=MYSERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = dbSchema.GetDataContext()

query {
    for row in db.Table2 do
        where (row.TestData1.HasValue && row.TestData1.Value > 2)
        select row
} |> Seq.iter (fun row -> printfn "%d %s" row.TestData1.Value row.Name)

query {
    for row in db.Table2 do
        // Use a nullable operator ?>
        where (row.TestData1 ?> 2)
        select row
} |> Seq.iter (fun row -> printfn "%d %s" (row.TestData1.GetValueOrDefault()) row.Name)

```

Vea también

- [Proveedores de tipos](#)
- [Expresiones de consulta](#)

Funciones

23/07/2020 • 16 minutes to read • [Edit Online](#)

Las funciones son la unidad fundamental de la ejecución del programa en cualquier lenguaje de programación. Al igual que en otros lenguajes, una función de F# tiene un nombre, puede tener parámetros y tomar argumentos, y tiene un cuerpo. F# también admite construcciones de programación funcional como el tratamiento de las funciones como valores, el uso de funciones sin nombre en expresiones, la composición de funciones para crear nuevas funciones, funciones curricadas y la definición implícita de funciones mediante la aplicación parcial de argumentos de función.

Las funciones se definen mediante la palabra clave `let`, o bien, si la función es recursiva, la combinación de palabras clave `let rec`.

Sintaxis

```
// Non-recursive function definition.  
let [inline] function-name parameter-list [ : return-type ] = function-body  
// Recursive function definition.  
let rec function-name parameter-list = recursive-function-body
```

Comentarios

El *nombre de la función* es un identificador que representa la función. La *lista de parámetros* consta de parámetros sucesivos separados por espacios. Se puede especificar un tipo explícito para cada parámetro, tal como se describe en la sección Parámetros. Si no se especifica un tipo de argumento concreto, el compilador intenta deducir el tipo del cuerpo de la función. El *cuerpo de la función* consta de una expresión. La expresión que forma el cuerpo de la función suele ser una expresión compuesta formada por varias expresiones que culminan en una expresión final que es el valor devuelto. El *tipo de valor devuelto* es un signo de dos puntos seguido de un tipo y es opcional. Si no se especifica explícitamente el tipo del valor devuelto, el compilador determina el tipo de valor devuelto a partir de la expresión final.

Una definición de función simple es similar a la siguiente:

```
let f x = x + 1
```

En el ejemplo anterior, el nombre de función es `f`, el argumento es `x`, que tiene el tipo `int`, el cuerpo de la función es `x + 1` y el valor devuelto es de tipo `int`.

Las funciones se pueden marcar como `inline`. Para más información sobre `inline`, vea [Inline Functions](#) (Funciones insertadas).

Ámbito

En cualquier nivel de ámbito distinto al ámbito de módulo, no es un error volver a usar un nombre de función o valor. Si se vuelve a usar un nombre, el último nombre declarado prevalece sobre el declarado anteriormente. Pero en el ámbito de nivel superior en un módulo, los nombres deben ser únicos. Por ejemplo, el código siguiente produce un error cuando aparece en el ámbito de módulo, pero no cuando aparece dentro de una función:

```
let list1 = [ 1; 2; 3]
// Error: duplicate definition.
let list1 = []
let function1 =
  let list1 = [1; 2; 3]
  let list1 = []
  list1
```

Pero el código siguiente es aceptable en cualquier nivel de ámbito:

```
let list1 = [ 1; 2; 3]
let sumPlus x =
  // OK: inner list1 hides the outer list1.
  let list1 = [1; 5; 10]
  x + List.sum list1
```

Parámetros

Los nombres de los parámetros aparecen después del nombre de función. Se puede especificar un tipo para un parámetro, como se muestra en el ejemplo siguiente:

```
let f (x : int) = x + 1
```

Si se especifica un tipo, sigue al nombre del parámetro y se separa del nombre mediante dos puntos. Si se omite el tipo del parámetro, el compilador infiere el tipo de parámetro. Por ejemplo, en la siguiente definición de función, se deduce que el argumento `x` es del tipo `int` porque `1` es de tipo `int`.

```
let f x = x + 1
```

Pero el compilador intentará que la función sea lo más genérica posible. Por ejemplo, observe el código siguiente:

```
let f x = (x, x)
```

La función crea una tupla a partir de un argumento de cualquier tipo. Dado que no se especifica el tipo, la función se puede usar con cualquier tipo de argumento. Para más información, vea [Automatic Generalization](#) (Generalización automática).

Cuerpos de función

El cuerpo de una función puede contener definiciones de variables locales y funciones. Estas variables y funciones están en ámbito en el cuerpo de la función actual, pero no fuera de ella. Una vez habilitada la opción de sintaxis ligera, se debe usar sangría para indicar que es una definición de un cuerpo de función, como se muestra en el ejemplo siguiente:

```
let cylinderVolume radius length =
  // Define a local value pi.
  let pi = 3.14159
  length * pi * radius * radius
```

Para más información, vea [Code Formatting Guidelines](#) (Instrucciones de formato de código) y [Verbose Syntax](#) (Sintaxis detallada).

Valores devueltos

El compilador usa la expresión final en el cuerpo de una función para determinar el tipo y el valor devuelto. Es posible que el compilador deduzca el tipo de la expresión final a partir las expresiones anteriores. En la función `cylinderVolume`, como se muestra en la sección anterior, el tipo de `pi` se determina a partir del tipo del literal `3.14159` como `float`. El compilador usa el tipo de `pi` para determinar el tipo de la expresión `h * pi * r * r` como `float`. Por tanto, el tipo de valor devuelto global de la función es `float`.

Para especificar explícitamente el valor devuelto, escriba el código de esta forma:

```
let cylinderVolume radius length : float =  
    // Define a local value pi.  
    let pi = 3.14159  
    length * pi * radius * radius
```

Tal como se escribe el código anterior, el compilador aplica `float` a toda la función. Si también quiere aplicarlo a los tipos de parámetro, use el código siguiente:

```
let cylinderVolume (radius : float) (length : float) : float
```

Llamar a una función

Las funciones se llaman especificando el nombre de la función seguido de un espacio y, después, los argumentos separados por espacios. Por ejemplo, para llamar a la función `cylinderVolume` y asignar el resultado al valor `vol`, se escribe el código siguiente:

```
let vol = cylinderVolume 2.0 3.0
```

Aplicación parcial de argumentos

Si se proporcionan menos argumentos que los especificados, se crea una nueva función que espera los argumentos restantes. Este método de control de argumentos se conoce como *currificación* y es una característica de los lenguajes de programación funcionales como F#. Por ejemplo, supongamos que está trabajando con dos tamaños de canalización: una tiene un radio de 2,0 y la otra tiene un radio de 3,0. Se podrían crear funciones que determinen el volumen de canalización de esta forma:

```
let smallPipeRadius = 2.0  
let bigPipeRadius = 3.0  
  
// These define functions that take the length as a remaining  
// argument:  
  
let smallPipeVolume = cylinderVolume smallPipeRadius  
let bigPipeVolume = cylinderVolume bigPipeRadius
```

Después, se proporcionaría el argumento adicional según sea necesario para las distintas longitudes de canalización de los dos tamaños diferentes:

```
let length1 = 30.0
let length2 = 40.0
let smallPipeVol1 = smallPipeVolume length1
let smallPipeVol2 = smallPipeVolume length2
let bigPipeVol1 = bigPipeVolume length1
let bigPipeVol2 = bigPipeVolume length2
```

Funciones recursivas

Las *funciones recursivas* son funciones que se llaman a sí mismas. Requieren que se especifique la palabra clave **rec** después de la palabra clave **let**. La función recursiva se invoca desde el interior del cuerpo de la función de la misma forma que se invocaría cualquier llamada de función. La siguiente función recursiva calcula el número n de Fibonacci. La secuencia de números de Fibonacci se conoce desde la antigüedad y es una secuencia en la que cada número sucesivo es la suma de los dos números anteriores en la secuencia.

```
let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)
```

Es posible que algunas funciones recursivas desborden la pila del programa o tengan un rendimiento ineficaz si no se escriben con cuidado y con el conocimiento de determinadas técnicas especiales, como el uso de acumuladores y continuaciones.

Valores de función

En F#, todas las funciones se consideran valores, de hecho, se conocen como *valores de función*. Dado que las funciones son valores, se pueden usar como argumentos de otras funciones o en otros contextos donde se usan los valores. El siguiente ejemplo muestra una función que toma un valor de función como argumento:

```
let apply1 (transform : int -> int) y = transform y
```

El tipo de un valor de función se especifica mediante el token `->`. En el lado izquierdo de este token está el tipo del argumento y , en el lado derecho, el valor devuelto. En el ejemplo anterior, `apply1` es una función que toma una función `transform` como argumento, donde `transform` es una función que toma un entero y devuelve otro entero. En el código siguiente se muestra cómo usar `apply1`:

```
let increment x = x + 1

let result1 = apply1 increment 100
```

El valor de `result` será 101 después de ejecutar el código anterior.

Si hay varios argumentos, se separan por sucesivos tokens `->`, como se muestra en el ejemplo siguiente:

```
let apply2 (f: int -> int -> int) x y = f x y

let mul x y = x * y

let result2 = apply2 mul 10 20
```

El resultado es 200.

Expresiones lambda

Una *expresión lambda* es una función sin nombre. En los ejemplos anteriores, en lugar de definir las funciones con nombre **increment** y **mul**, se podrían usar expresiones lambda de esta forma:

```
let result3 = apply1 (fun x -> x + 1) 100

let result4 = apply2 (fun x y -> x * y ) 10 20
```

Las expresiones lambda se definen mediante la palabra clave `fun`. Una expresión lambda es similar a una definición de función, salvo que en lugar del token `=` se usa el token `->` para separar la lista de argumentos del cuerpo de la función. Al igual que en una definición de función normal, se pueden deducir o especificar explícitamente los tipos de argumento, y el tipo de valor devuelto de la expresión lambda se deduce del tipo de la última expresión en el cuerpo. Para obtener más información, vea [Expresiones lambda: Palabra clave `fun`](#).

Composición de funciones y canalización

En F#, las funciones se pueden componer a partir de otras funciones. La composición de dos funciones **función1** y **función2** es otra función que representa la aplicación de **función1** seguida de la aplicación de **función2**:

```
let function1 x = x + 1
let function2 x = x * 2
let h = function1 >> function2
let result5 = h 100
```

El resultado es 202.

La canalización permite encadenar llamadas a funciones como operaciones sucesivas. La canalización funciona de la siguiente manera:

```
let result = 100 |> function1 |> function2
```

El resultado es 202 de nuevo.

Los operadores de composición toman dos funciones y devuelven una función. Por el contrario, los operadores de canalización toman una función y un argumento, y devuelven un valor. En el ejemplo de código siguiente se muestra la diferencia entre los operadores de canalización y composición mostrando las diferencias en las firmas de función y el uso.

```
// Function composition and pipeline operators compared.

let addOne x = x + 1
let timesTwo x = 2 * x

// Composition operator
// ( >> ) : ('T1 -> 'T2) -> ('T2 -> 'T3) -> 'T1 -> 'T3
let Compose2 = addOne >> timesTwo

// Backward composition operator
// ( << ) : ('T2 -> 'T3) -> ('T1 -> 'T2) -> 'T1 -> 'T3
let Compose1 = addOne << timesTwo

// Result is 5
let result1 = Compose1 2

// Result is 6
let result2 = Compose2 2

// Pipelining
// Pipeline operator
// ( |> ) : 'T1 -> ('T1 -> 'U) -> 'U
let Pipeline2 x = addOne x |> timesTwo

// Backward pipeline operator
// ( <| ) : ('T -> 'U) -> 'T -> 'U
let Pipeline1 x = addOne <| timesTwo x

// Result is 5
let result3 = Pipeline1 2

// Result is 6
let result4 = Pipeline2 2
```

Sobrecargar funciones

Los métodos de un tipo se pueden sobrecargar, pero no las funciones. Para más información, vea [Métodos](#).

Vea también

- [Valores](#)
- [Referencia del lenguaje F#](#)

Enlaces let

23/10/2019 • 9 minutes to read • [Edit Online](#)

Un *enlace* asocia un identificador con un valor o una función. La `let` palabra clave se usa para enlazar un nombre a un valor o una función.

Sintaxis

```
// Binding a value:  
let identifier-or-pattern [: type] =expressionbody-expression  
// Binding a function value:  
let identifier parameter-list [: return-type ] =expressionbody-expression
```

Comentarios

La `let` palabra clave se usa en expresiones de enlace para definir valores o valores de función para uno o más nombres. La forma más sencilla de la `let` expresión enlaza un nombre a un valor simple, como se indica a continuación.

```
let i = 1
```

Si separa la expresión del identificador usando una nueva línea, debe aplicar sangría a cada línea de la expresión, como en el código siguiente.

```
let someVeryLongIdentifier =  
  // Note indentation below.  
  3 * 4 + 5 * 6
```

En lugar de simplemente un nombre, se puede especificar un patrón que contenga nombres, por ejemplo, una tupla, tal y como se muestra en el código siguiente.

```
let i, j, k = (1, 2, 3)
```

El *cuerpo-expresión* es la expresión en la que se usan los nombres. La expresión de cuerpo aparece en su propia línea, con la sangría que se alinea exactamente con el primer carácter de `let` la palabra clave:

```
let result =  
  
  let i, j, k = (1, 2, 3)  
  
  // Body expression:  
  i + 2*j + 3*k
```

Un `let` enlace puede aparecer en el nivel de módulo, en la definición de un tipo de clase o en ámbitos locales, como en una definición de función. No es necesario que un enlace en el nivel superior de un módulo o de un tipo de clase tenga una expresión de cuerpo, sino que en otros niveles de ámbito, se requiere la expresión del cuerpo. Los nombres enlazados se pueden usar después del punto de definición, pero no en ningún momento antes de que aparezca el enlace, como se muestra en el código siguiente.


```
// Error:
printfn "%d" x
let x = 100
// OK:
printfn "%d" x
```

Enlaces de función

Los enlaces de función siguen las reglas de los enlaces de valor, excepto que los enlaces de función incluyen el nombre de la función y los parámetros, como se muestra en el código siguiente.

```
let function1 a =
    a + 1
```

En general, los parámetros son patrones, como un patrón de tupla:

```
let function2 (a, b) = a + b
```

Una `let` expresión de enlace se evalúa como el valor de la última expresión. Por lo tanto, en el ejemplo de código siguiente, `result` el valor de se `100 * function3 (1, 2)` calcula a partir de, `300` que se evalúa como.

```
let result =
    let function3 (a, b) = a + b
    100 * function3 (1, 2)
```

Para obtener más información, vea [Funciones](#).

Anotaciones de tipo

Puede especificar los tipos para los parámetros incluyendo dos puntos (:) seguido de un nombre de tipo, todo entre paréntesis. También puede especificar el tipo del valor devuelto anexando los dos puntos y el tipo después del último parámetro. Las anotaciones de tipo completo para `function1`, con enteros como tipos de parámetro, serían como se indica a continuación.

```
let function1 (a: int) : int = a + 1
```

Cuando no hay ningún parámetro de tipo explícito, se usa la inferencia de tipos para determinar los tipos de parámetros de las funciones. Esto puede incluir la generalización automática del tipo de un parámetro para que sea genérico.

Para obtener más información, vea [generalización automática](#) e inferencia de [tipos](#).

Enlaces let en clases

Un `let` enlace puede aparecer en un tipo de clase, pero no en un tipo de estructura o de registro. Para usar un enlace `let` en un tipo de clase, la clase debe tener un constructor `Primary`. Los parámetros de constructor deben aparecer después del nombre de tipo en la definición de clase. Un `let` enlace en un tipo de clase define los campos privados y los miembros de ese tipo de clase `do` y, junto con los enlaces del tipo, crea el código para el constructor principal para el tipo. En los siguientes ejemplos de código se `MyClass` muestra una clase `field1` con `field2` campos privados y.

```
type MyClass(a) =  
  let field1 = a  
  let field2 = "text"  
  do printfn "%d %s" field1 field2  
  member this.F input =  
    printfn "Field1 %d Field2 %s Input %A" field1 field2 input
```

Los ámbitos de `field1` y `field2` se limitan al tipo en el que se declaran. Para obtener más información, vea [let](#) [enlaces en clases](#) y [clases](#).

Parámetros de tipo en los enlaces Let

Un `let` enlace en el nivel de módulo, en un tipo o en una expresión de cálculo puede tener parámetros de tipo explícitos. Un enlace Let en una expresión, como dentro de una definición de función, no puede tener parámetros de tipo. Para más información, vea [Genéricos](#).

Atributos en enlaces Let

Los atributos se pueden aplicar a los enlaces `let` de nivel superior de un módulo, tal y como se muestra en el código siguiente.

```
[<Obsolete>  
let function1 x y = x + y
```

Ámbito y accesibilidad de los enlaces Let

El ámbito de una entidad declarada con un enlace Let está limitado a la parte del ámbito contenedor (por ejemplo, una función, un módulo, un archivo o una clase) después de que aparezca el enlace. Por lo tanto, se puede decir que un enlace Let introduce un nombre en un ámbito. En un módulo, se puede acceder a un valor o función enlazado a los clientes de un módulo siempre que se pueda acceder al módulo, ya que los enlaces Let de un módulo se compilan en funciones públicas del módulo. Por el contrario, los enlaces Let de una clase son privados para la clase.

Normalmente, las funciones de los módulos deben calificarse con el nombre del módulo cuando se usan en el código de cliente. Por ejemplo, si un módulo `Module1` tiene una función `function1`, los usuarios especificarán `Module1.function1` que hagan referencia a la función.

Los usuarios de un módulo pueden usar una declaración de importación para hacer que las funciones de ese módulo estén disponibles para su uso sin que el nombre del módulo los califique. En el ejemplo que se acaba de mencionar, los usuarios del módulo pueden, en ese caso, abrir el módulo mediante `Module1` la declaración de importación `function1` Open y, a continuación, hacer referencia directamente a.

```
module Module1 =  
  let function1 x = x + 1.0  
  
module Module2 =  
  let function2 x =  
    Module1.function1 x  
  
open Module1  
  
let function3 x =  
  function1 x
```

Algunos módulos tienen el atributo [RequireQualifiedAccess](#), lo que significa que las funciones que exponen

deben calificarse con el nombre del módulo. Por ejemplo, el F# módulo List tiene este atributo.

Para obtener más información sobre los módulos y el control de acceso, consulte [módulos](#) y [Access Control](#).

Vea también

- [Funciones](#)
- `let` [Bindings in Classes](#) (Enlaces `let` en clases)

do (Enlaces)

23/10/2019 • 2 minutes to read • [Edit Online](#)

Un `do` enlace se utiliza para ejecutar código sin definir una función o un valor. Además, los enlaces `do` se pueden usar en las clases, vea [do enlaces en clases](#).

Sintaxis

```
[ attributes ]  
[ do ]expression
```

Comentarios

Use un `do` enlace cuando desee ejecutar código independientemente de una definición de función o valor. La expresión de un `do` enlace debe devolver `unit`. El código de un enlace de `do` nivel superior se ejecuta cuando se inicializa el módulo. La palabra `do` clave es opcional.

Los atributos se pueden aplicar a un enlace de `do` nivel superior. Por ejemplo, si el programa utiliza la interoperabilidad com, es posible que `STAThread` desee aplicar el atributo a su programa. Para ello, puede usar un atributo en un `do` enlace, tal y como se muestra en el código siguiente.

```
open System  
open System.Windows.Forms  
  
let form1 = new Form()  
form1.Text <- "XYZ"  
  
[<STAThread>]  
do  
    Application.Run(form1)
```

Vea también

- [Referencia del lenguaje F#](#)
- [Funciones](#)

Expresiones lambda: Palabra clave Fun (F#)

23/10/2019 • 2 minutes to read • [Edit Online](#)

La `fun` palabra clave se usa para definir una expresión lambda, es decir, una función anónima.

Sintaxis

```
fun parameter-list -> expression
```

Comentarios

La *lista de parámetros* consta normalmente de nombres y, opcionalmente, tipos de parámetros. En general, la *lista de parámetros* puede estar formada por F# patrones. Para obtener una lista completa de los posibles patrones, consulte [coincidencia de patrones](#). Entre las listas de parámetros válidos se incluyen los ejemplos siguientes.

```
// Lambda expressions with parameter lists.
fun a b c -> ...
fun (a: int) b c -> ...
fun (a : int) (b : string) (c:float) -> ...

// A lambda expression with a tuple pattern.
fun (a, b) -> ...

// A lambda expression with a list pattern.
fun head :: tail -> ...
```

La *expresión* es el cuerpo de la función, la última expresión de la que genera un valor devuelto. Entre los ejemplos de expresiones lambda válidas se incluyen los siguientes:

```
fun x -> x + 1
fun a b c -> printfn "%A %A %A" a b c
fun (a: int) (b: int) (c: int) -> a + b * c
fun x y -> let swap (a, b) = (b, a) in swap (x, y)
```

Uso de expresiones lambda

Las expresiones lambda son especialmente útiles cuando se desea realizar operaciones en una lista o en otra colección y se desea evitar el trabajo adicional de definir una función. Muchas F# funciones de la biblioteca toman los valores de función como argumentos y puede ser especialmente conveniente usar una expresión lambda en esos casos. En el código siguiente se aplica una expresión lambda a los elementos de una lista. En este caso, la función anónima agrega 1 a todos los elementos de una lista.

```
let list = List.map (fun i -> i + 1) [1;2;3]
printfn "%A" list
```

Vea también

- [Funciones](#)

Funciones recursivas: La palabra clave REC

23/10/2019 • 2 minutes to read • [Edit Online](#)

La `rec` palabra clave se usa junto con `let` la palabra clave para definir una función recursiva.

Sintaxis

```
// Recursive function:
let rec function-nameparameter-list =
    function-body

// Mutually recursive functions:
let rec function1-nameparameter-list =
    function1-body
and function2-nameparameter-list =
    function2-body
...
```

Comentarios

Las funciones recursivas, las funciones que se llaman a sí mismas, F# se identifican explícitamente en el lenguaje. Esto hace que el identificador que se está definiendo esté disponible en el ámbito de la función.

En el código siguiente se muestra una función recursiva que calcula el número^{de Fibonacci} n .

```
let rec fib n =
    if n <= 2 then 1
    else fib (n - 1) + fib (n - 2)
```

NOTE

En la práctica, el código como el anterior es una pérdida de memoria y tiempo de procesador porque implica el recálculo de valores calculados previamente.

Los métodos son implícitamente recursivos dentro del tipo; no es necesario agregar la `rec` palabra clave. Los enlaces `Let` dentro de las clases no son implícitamente recursivos.

Funciones mutuamente recursivas

A veces, las funciones son mutuamente recursivas, lo que significa que las llamadas forman un círculo, donde una función llama a otra que, a su vez, llama a la primera, con cualquier número de llamadas entre. Debe definir estas funciones juntas en un `let` enlace, utilizando la `and` palabra clave para vincularlas.

En el ejemplo siguiente se muestran dos funciones mutuamente recursivas.

```
let rec Even x =  
  if x = 0 then true  
  else Odd (x-1)  
and Odd x =  
  if x = 0 then false  
  else Even (x-1)
```

Vea también

- [Funciones](#)

Punto de entrada

23/10/2019 • 2 minutes to read • [Edit Online](#)

En este tema se describe el método que se usa para establecer el punto de F# entrada en un programa.

Sintaxis

```
[<EntryPoint>]  
let-function-binding
```

Comentarios

En la sintaxis anterior, *Let-function-Binding* es la definición de una función en un `let` enlace.

El punto de entrada a un programa que se compila como un archivo ejecutable es donde se inicia la ejecución formalmente. Para especificar el punto de entrada a F# una aplicación, aplique `EntryPoint` el atributo a la función `main` del programa. Esta función (creada mediante un `let` enlace) debe ser la última función del último archivo compilado. El último archivo compilado es el último archivo del proyecto o el último archivo que se pasa a la línea de comandos.

La función de punto de entrada `string array -> int` tiene el tipo. Los argumentos proporcionados en la línea de comandos se pasan `main` a la función en la matriz de cadenas. El primer elemento de la matriz es el primer argumento; el nombre del archivo ejecutable no se incluye en la matriz, como en otros lenguajes. El valor devuelto se usa como código de salida para el proceso. Cero suele indicar Success; los valores distintos de cero indican un error. No hay ninguna Convención para el significado específico de códigos de retorno distintos de cero; los significados de los códigos de retorno son específicos de la aplicación.

En el ejemplo siguiente se muestra una `main` función simple.

```
[<EntryPoint>]  
let main args =  
    printfn "Arguments passed to function : %A" args  
    // Return 0. This indicates success.  
    0
```

Cuando este código se ejecuta con la línea `EntryPoint.exe 1 2 3` de comandos, el resultado es el siguiente.

```
Arguments passed to function : [|"1"; "2"; "3"|]
```

Punto de entrada implícito

Cuando un programa no tiene ningún atributo `EntryPoint` que indique explícitamente el punto de entrada, los enlaces de nivel superior del último archivo que se va a compilar se usan como punto de entrada.

Vea también

- [Funciones](#)
- [Enlaces let](#)

Funciones externas

23/10/2019 • 2 minutes to read • [Edit Online](#)

En este tema F# se describe la compatibilidad con lenguajes para llamar a funciones en código nativo.

Sintaxis

```
[<DllImport( arguments )>]  
extern declaration
```

Comentarios

En la sintaxis anterior, *arguments* representa los `System.Runtime.InteropServices.DllImportAttribute` argumentos que se proporcionan al atributo. El primer argumento es una cadena que representa el nombre del archivo DLL que contiene esta función, sin la extensión. dll. Se pueden proporcionar argumentos adicionales para cualquiera de las propiedades públicas de la `System.Runtime.InteropServices.DllImportAttribute` clase, como la Convención de llamada.

Suponga que tiene un archivo C++ dll nativo que contiene la siguiente función exportada.

```
#include <stdio.h>  
extern "C" void __declspec(dllexport) HelloWorld()  
{  
    printf("Hello world, invoked by F#\n");  
}
```

Puede llamar a esta función desde F# mediante el código siguiente.

```
open System.Runtime.InteropServices  
  
module InteropWithNative =  
    [DllImport(@"C:\bin\nativedll", CallingConvention = CallingConvention.Cdecl)]  
    extern void HelloWorld()  
  
InteropWithNative.HelloWorld()
```

La interoperabilidad con código nativo se conoce como invocación de *plataforma* y es una característica de CLR. Para más información, consulte [Interoperating with Unmanaged Code](#) (Interoperar con código no administrado) La información de esa sección es aplicable a F#.

Vea también

- [Funciones](#)

Funciones insertadas

23/10/2019 • 3 minutes to read • [Edit Online](#)

Las *funciones insertadas* son funciones que se integran directamente en el código de llamada.

Usar funciones insertadas

Cuando se usan parámetros de tipo estático, cualquier función que esté parametrizada por parámetros de tipo debe estar alineada. Esto garantiza que el compilador puede resolver estos parámetros de tipo. Cuando se usan parámetros de tipo genérico normales, no hay ninguna restricción.

Aparte de habilitar el uso de restricciones de miembro, las funciones insertadas pueden ser útiles para optimizar el código. Sin embargo, el uso excesivo de las funciones insertadas puede hacer que el código sea menos resistente a los cambios en las optimizaciones del compilador y la implementación de las funciones de la biblioteca. Por esta razón, debe evitar el uso de funciones insertadas para la optimización a menos que haya intentado todas las demás técnicas de optimización. Hacer que una función o un método insertado a veces puede mejorar el rendimiento, pero no siempre es el caso. Por lo tanto, también debe usar medidas de rendimiento para comprobar que la realización de cualquier función insertada en realidad tiene un efecto positivo.

El `inline` modificador se puede aplicar a las funciones en el nivel superior, en el nivel de módulo o en el nivel de método de una clase.

En el ejemplo de código siguiente se muestra una función insertada en el nivel superior, un método de instancia insertado y un método estático insertado.

```
let inline increment x = x + 1
type WrapInt32() =
    member inline this.IncrementByOne(x) = x + 1
    static member inline Increment(x) = x + 1
```

Funciones insertadas e inferencia de tipos

La presencia de `inline` afecta a la inferencia de tipos. Esto se debe a que las funciones insertadas pueden tener parámetros de tipo resueltos estáticamente, mientras que las funciones no insertadas no pueden. En el ejemplo de código siguiente se muestra `inline` un caso en el que resulta útil porque se está usando una función que tiene un parámetro de `float` tipo resuelto estáticamente, el operador de conversión.

```
let inline printAsFloatingPoint number =
    printfn "%f" (float number)
```

Sin el `inline` modificador, la inferencia de tipos fuerza a la función a tomar un tipo específico, `int` en este caso. Pero con el `inline` modificador, la función también se deduce para tener un parámetro de tipo resuelto estáticamente. Con el `inline` modificador, se deduce que el tipo es el siguiente:

```
^a -> unit when ^a : (static member op_Explicit : ^a -> float)
```

Esto significa que la función acepta cualquier tipo que admita una conversión a **float**.

Vea también

- Funciones
- Restricciones
- Parámetros de tipo resueltos estáticamente

Valores

23/10/2019 • 6 minutes to read • [Edit Online](#)

Los valores de F# son cantidades que tienen un tipo específico. Los valores pueden ser números enteros o de punto flotante, caracteres o texto, listas, secuencias, matrices, tuplas, uniones discriminadas, registros, tipos de clase o valores de función.

Enlace de un valor

El término *enlace* significa asociar un nombre a una definición. La palabra clave `let` enlaza un valor, como en los ejemplos siguientes:

```
let a = 1
let b = 100u
let str = "text"

// A function value binding.

let f x = x + 1
```

El tipo de un valor se infiere de la definición. Para un tipo primitivo, como un número entero o de punto flotante, el tipo se determina a partir del tipo del literal. Por tanto, en el ejemplo anterior, el compilador infiere que el tipo de `b` es `unsigned int`, mientras que el compilador infiere que el tipo de `a` es `int`. El tipo de un valor de función se determina a partir del valor devuelto en el cuerpo de la función. Para más información sobre los tipos de valor de función, vea [Funciones](#). Para más información sobre los tipos literales, vea [Literals](#) (Literales).

De forma predeterminada, el compilador no emite diagnósticos sobre los enlaces no utilizados. Para recibir estos mensajes, habilite la advertencia 1182 en el archivo del proyecto o al invocar al compilador (vea `--warnon` en [Opciones del compilador](#)).

¿Por qué inmutables?

Los valores inmutables son valores que no se pueden cambiar durante el transcurso de la ejecución de un programa. Si está habituado a usar lenguajes como C++, Visual Basic o C#, le resultará sorprendente que F# dé prioridad a los valores inmutables y no a las variables, a las que se pueden asignar nuevos valores durante la ejecución de un programa. Los datos inmutables son un elemento importante de la programación funcional. En un entorno multiproceso, resulta complicado administrar las variables mutables compartidas, dado que pueden modificarlas muchos subprocesos diferentes. Además, con las variables mutables, a veces puede resultar difícil saber si existe la posibilidad de que una variable haya cambiado cuando se pasa a otra función.

En los lenguajes funcionales puros, no hay variables y las funciones se comportan estrictamente como funciones matemáticas. Cuando el código de un lenguaje de procedimientos usa una asignación de variable para modificar un valor, el código equivalente de un lenguaje funcional tiene un valor inmutable que es la entrada, una función inmutable y distintos valores inmutables como salida. Esta rigidez matemática permite un razonamiento más estricto sobre el comportamiento del programa. Y este razonamiento más estricto es lo que permite a los compiladores comprobar el código de una manera más rigurosa y optimizar con mayor eficacia, además de hacer que a los desarrolladores de software les resulte más fácil entender y escribir código correcto. Por tanto, es probable que el código funcional sea más fácil de depurar que el código de procedimientos ordinario.

Aunque F# no es un lenguaje funcional puro, admite plenamente la programación funcional. El uso de valores inmutables es una práctica correcta porque permite que el código se beneficie de un aspecto importante de la

programación funcional.

Variables mutables

Se puede usar la palabra clave `mutable` para especificar una variable que se puede cambiar. En F#, en general las variables mutables deben tener un ámbito limitado, ya sea como campo de un tipo o bien como valor local. Las variables mutables con un ámbito limitado son más fáciles de controlar y es menos probable que se modifiquen de manera incorrecta.

Se puede asignar un valor inicial a una variable mutable mediante la palabra clave `let` de la misma manera que se define un valor. Pero la diferencia reside en que, posteriormente, se pueden asignar nuevos valores a las variables mutables mediante el operador `<-`, como en el ejemplo siguiente.

```
let mutable x = 1
x <- x + 1
```

Los valores `mutable` marcados se pueden promover `'a ref` automáticamente a si se capturan mediante un cierre, incluidos los formularios que crean `seq` cierres, como los generadores. Si desea recibir una notificación cuando esto suceda, habilite la advertencia 3180 en el archivo del proyecto o al invocar al compilador.

Temas relacionados

TÍTULO	DESCRIPCIÓN
Enlaces let	Proporciona información sobre el uso <code>let</code> de la palabra clave para enlazar nombres a valores y funciones.
Funciones	Proporciona información general sobre las funciones en F#.

Vea también

- [Valores NULL](#)
- [Referencia del lenguaje F#](#)

Valores NULL

23/10/2019 • 5 minutes to read • [Edit Online](#)

En este tema se describe cómo se utiliza el valor F#null en.

Valor null

El valor NULL no se utiliza normalmente en F# para valores o variables. Sin embargo, null aparece como un valor anómalo en determinadas situaciones. Si se define un tipo en F#, no se permite NULL como valor normal a menos que el atributo [AllowNullLiteral](#) se aplique al tipo. Si un tipo se define en algún otro lenguaje de .NET, NULL es un valor posible y, al interoperar con estos tipos, el F# código podría encontrar valores NULL.

Para un tipo definido en F# y utilizado estrictamente desde F#, la única forma de crear un valor null utilizando directamente F# la biblioteca es usar `unchecked . default` o `array. zerocreate ()`. Sin embargo, para F# un tipo que se usa desde otros lenguajes .net, o si está usando ese tipo con una API que no está escrita F#en, como la .NET Framework, se pueden producir valores NULL.

Puede usar el `option` tipo en F# cuando pueda usar una variable de referencia con un valor null posible en otro lenguaje .net. En lugar de NULL, con un F# `option` tipo, se usa el valor `None` de la opción si no hay ningún objeto. El valor `Some(obj)` de la opción se usa con `obj` un objeto cuando hay un objeto. Para obtener más información, vea [Opciones](#). Tenga en cuenta que todavía `null` puede empaquetar un valor en una opción si, para `Some x`, `x` es `null`. Por este motivo, es importante usar `None` cuando un valor es `null`.

La `null` palabra clave es una palabra clave válida F# en el lenguaje y tiene que usarlo al trabajar con .NET Framework API u otras API escritas en otro lenguaje .net. Las dos situaciones en las que podría necesitar un valor null son cuando se llama a una API de .NET y se pasa un valor NULL como argumento, y cuando se interpreta el valor devuelto o un parámetro de salida desde una llamada al método .NET.

Para pasar un valor null a un método .net, simplemente use la `null` palabra clave en el código de llamada. En el siguiente ejemplo código se muestra cómo hacerlo.

```
open System

// Pass a null value to a .NET method.
let ParseDateTime (str: string) =
    let (success, res) = DateTime.TryParse(str, null, System.Globalization.DateTimeStyles.AssumeUniversal)
    if success then
        Some(res)
    else
        None
```

Para interpretar un valor null que se obtiene de un método .NET, use la coincidencia de patrones si es posible. En el ejemplo de código siguiente se muestra cómo usar la coincidencia de `ReadLine` patrones para interpretar el valor null que se devuelve cuando intenta leer más allá del final de un flujo de entrada.

```
// Open a file and create a stream reader.
let fileStream1 =
    try
        System.IO.File.OpenRead("TextFile1.txt")
    with
        | :? System.IO.FileNotFoundException -> printfn "Error: TextFile1.txt not found."; exit(1)

let streamReader = new System.IO.StreamReader(fileStream1)

// ProcessNextLine returns false when there is no more input;
// it returns true when there is more input.
let ProcessNextLine nextLine =
    match nextLine with
    | null -> false
    | inputString ->
        match ParseDateTime inputString with
        | Some(date) -> printfn "%s" (date.ToLocalTime().ToString())
        | None -> printfn "Failed to parse the input."
        true

// A null value returned from .NET method ReadLine when there is
// no more input.
while ProcessNextLine (streamReader.ReadLine()) do ()
```

Los valores NULL F# de los tipos también se pueden generar de otras maneras, como cuando se `Array.zeroCreate` usa, que `Unchecked.defaultof` llama a. Debe tener cuidado con este código para mantener encapsulados los valores NULL. En una biblioteca diseñada únicamente para F#, no es necesario comprobar si hay valores NULL en cada función. Si está escribiendo una biblioteca para interoperar con otros lenguajes de .net, es posible que tenga que agregar comprobaciones para los parámetros `ArgumentNullException` de entrada NULL y producir una C# , tal como se hace en o Visual Basic código.

Puede usar el código siguiente para comprobar si un valor arbitrario es NULL.

```
match box value with
| null -> printf "The value is null."
| _ -> printf "The value is not null."
```

Vea también

- [Valores](#)
- [Expresiones de coincidencia](#)

Literales

29/10/2019 • 4 minutes to read • [Edit Online](#)

NOTE

Los vínculos de referencia de la API de este artículo le llevarán a MSDN (por ahora).

En este tema se proporciona una tabla que muestra cómo especificar el tipo de un literal F#.en.

Tipos literales

En la tabla siguiente se muestran los tipos F#literales en. Los caracteres que representan dígitos en notación hexadecimal no distinguen mayúsculas de minúsculas; los caracteres que identifican el tipo distinguen mayúsculas de minúsculas.

TYPE	DESCRIPCIÓN	SUFIJO O PREFIJO	EJEMPLOS
sbyte	entero de 8 bits con signo	s	<code>86y</code> <code>0b00000101y</code>
byte	número natural de 8 bits sin signo	uy	<code>86uy</code> <code>0b00000101uy</code>
int16	entero de 16 bits con signo	s	<code>86s</code>
uint16	número natural de 16 bits sin signo	us	<code>86us</code>
int	entero de 32 bits con signo	l o ninguno	<code>86</code>
int32			<code>86l</code>
uint	unsigned 32-número natural de bits	u o UL	<code>86u</code>
uint32			<code>86ul</code>
nativeint (puntero nativo a un número natural con signo	n	<code>123n</code>
unativeint (puntero nativo como un número natural sin signo	CEPE	<code>0x00002D3Fun</code>
int64	entero de 64 bits con signo	L	<code>86L</code>
uint64	unsigned 64-número natural de bits	UL	<code>86UL</code>

usar mayúsculas iniciales al definir los literales.

```
[<Literal>]
let SomeJson = ""{"numbers":[1,2,3,4,5]}""

[<Literal>]
let Literal1 = "a" + "b"

[<Literal>]
let FileLocation = __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let Literal2 = 1 ||| 64

[<Literal>]
let Literal3 = System.IO.FileAccess.Read ||| System.IO.FileAccess.Write
```

Comentarios

Las cadenas Unicode pueden contener codificaciones explícitas que puede especificar mediante `\u` seguido de una codificación de código hexadecimal de 16 bits (0000-FFFF) o UTF-32 que puede especificar mediante `\u` seguido de un código hexadecimal de 32 bits que representa cualquier Unicode. punto de código (00000000-0010FFFF).

No se permite el uso de otros operadores bit a bit distintos de `|||`.

Enteros en otras bases

Los enteros de 32 bits con signo también se pueden especificar en formato hexadecimal, octal o binario mediante un prefijo `0x`, `0o` o `0b` respectivamente.

```
let numbers = (0x9F, 0o77, 0b1010)
// Result: numbers : int * int * int = (159, 63, 10)
```

Caracteres de subrayado en literales numéricos

Puede separar los dígitos con el carácter de subrayado (`_`).

```
let value = 0xDEAD_BEEF

let valueAsBits = 0b1101_1110_1010_1101_1011_1110_1110_1111

let exampleSSN = 123_456_7890
```

Vea también

- [Clase Core. Literalattribute](#) (

Tipos en F#

08/01/2020 • 9 minutes to read • [Edit Online](#)

En este tema se describen los tipos que se F# usan en F# y cómo se denominan y describen los tipos.

Resumen de F# tipos

Algunos tipos se consideran *tipos primitivos*, como el tipo booleano `bool` y los tipos enteros y de punto flotante de varios tamaños, entre los que se incluyen los tipos de bytes y caracteres. Estos tipos se describen en [tipos primitivos](#).

Otros tipos que se integran en el lenguaje incluyen tuplas, listas, matrices, secuencias, registros y uniones discriminadas. Si tiene experiencia con otros lenguajes de .NET y está F#aprendiendo, debe leer los temas de cada uno de estos tipos. En la sección [temas relacionados](#) de este tema se incluyen vínculos a más información sobre estos tipos. Estos F#tipos específicos de son compatibles con los estilos de programación que son comunes a los lenguajes de programación funcionales. Muchos de estos tipos tienen módulos asociados en la F# biblioteca que admiten operaciones comunes en estos tipos.

El tipo de una función incluye información sobre los tipos de parámetro y el tipo de valor devuelto.

La .NET Framework es el origen de los tipos de objeto, los tipos de interfaz, los tipos de delegado y otros. Puede definir sus propios tipos de objeto igual que en cualquier otro lenguaje .NET.

Además, F# el código puede definir alias, que son *abreviaturas de tipo* con nombre, que son nombres alternativos para los tipos. Puede usar las abreviaturas de tipo cuando el tipo cambie en el futuro y desee evitar cambiar el código que depende del tipo. O bien, puede usar una abreviatura de tipo como nombre descriptivo para un tipo que pueda facilitar la lectura y comprensión del código.

F#proporciona tipos de colección útiles que están diseñados pensando en la programación funcional. El uso de estos tipos de colección le ayuda a escribir código que es más funcional en estilo. Para obtener más información, vea [F# tipos de colección](#).

Sintaxis para tipos

En F# el código, a menudo tiene que escribir los nombres de los tipos. Cada tipo tiene una forma sintáctica y estas formas sintácticas se utilizan en anotaciones de tipo, declaraciones de método abstractas, declaraciones de delegado, firmas y otras construcciones. Siempre que declare una nueva construcción de programa en el intérprete, el intérprete imprime el nombre de la construcción y la sintaxis de su tipo. Esta sintaxis puede ser simplemente un identificador de un tipo definido por el usuario o un identificador integrado como para `int` o `string`, pero para tipos más complejos, la sintaxis es más compleja.

En la tabla siguiente se muestran los aspectos de la F# sintaxis de tipos para los tipos.

TIPO DE	SINTAXIS DE TIPOS	EJEMPLOS
tipo primitivo	<i>type-name</i>	<code>int</code> <code>float</code> <code>string</code>

TIPO DE	SINTAXIS DE TIPOS	EJEMPLOS
tipo de agregado (clase, estructura, Unión, registro, enumeración, etc.)	<i>type-name</i>	<code>System.DateTime</code> <code>Color</code>
abreviatura de tipo	<i>type-abbreviation-name</i>	<code>bigint</code>
tipo completo	<i>namespaces.type-name</i> o <i>modules.type-name</i> o <i>namespaces.modules.type-name</i>	<code>System.IO.StreamWriter</code>
matriz	<i>nombre de tipo</i> [] o <i>matriz de nombres de tipo</i>	<code>int[]</code> <code>array<int></code> <code>int array</code>
matriz bidimensional	<i>type-name</i> [,]	<code>int[,]</code> <code>float[,]</code>
matriz tridimensional	<i>nombre de tipo</i> [,,]	<code>float[,,,]</code>
tuple	<i>Type-nombre1 * Type-nombre2 ...</i>	Por ejemplo, <code>(1, 'b', 3)</code> tiene el tipo <code>int * char * int</code>
tipo genérico	<i>type-parameter generic-type-name</i> o <i>generic-type-name</i> < <i>type-parameter-list</i> >	<code>'a list</code> <code>list<'a></code> <code>Dictionary<'key, 'value></code>
tipo construido (un tipo genérico que tiene un argumento de tipo específico proporcionado)	<i>type-argument generic-type-name</i> o <i>generic-type-name</i> < <i>type-argument-list</i> >	<code>int option</code> <code>string list</code> <code>int ref</code> <code>option<int></code> <code>list<string></code> <code>ref<int></code> <code>Dictionary<int, string></code>
tipo de función que tiene un parámetro único	<i>parameter-type1 -> return-type</i>	Función que toma un <code>int</code> y devuelve un <code>string</code> tiene el tipo <code>int -> string</code>

TIPO DE	SINTAXIS DE TIPOS	EJEMPLOS
tipo de función que tiene varios parámetros	<i>parámetro-type1 -> parámetro-tipo2 -> ...-> tipo de valor devuelto</i>	Una función que toma un <code>int</code> y un <code>float</code> y devuelve una <code>string</code> tiene el tipo <code>int -> float -> string</code>
función de orden superior como parámetro	<i>(tipo de función)</i>	<code>List.map</code> tiene el tipo <code>('a -> 'b) -> 'a list -> 'b list</code>
delegate	Delegado de <i>tipo de función</i>	<code>delegate of unit -> int</code>
tipo flexible	<i>nombre del tipo de #</i>	<code>#System.Windows.Forms.Control</code> <code>#seq<int></code>

Temas relacionados

TEMA	DESCRIPCIÓN
Tipos primitivos	Describe los tipos simples integrados, como los tipos enteros, el tipo booleano y los tipos de caracteres.
Tipo unit	Describe el tipo de <code>unit</code> , un tipo que tiene un valor y que se indica mediante (); equivalente a <code>void</code> en C# y <code>Nothing</code> en Visual Basic.
Tuplas	Describe el tipo de tupla, un tipo que consta de valores asociados de cualquier tipo agrupados en pares, tripas, cuádruplas, etc.
Opciones	Describe el tipo de opción, un tipo que puede tener un valor o estar vacío.
Listas	Describe las listas, que son una serie ordenada e inmutable de elementos del mismo tipo.
Matrices	Describe las matrices, que son conjuntos ordenados de elementos mutables del mismo tipo que ocupan un bloque de memoria contiguo y tienen un tamaño fijo.
Secuencias	Describe el tipo de secuencia, que representa una serie lógica de valores; los valores individuales se calculan solo según sea necesario.
Registros	Describe el tipo de registro, un agregado pequeño de valores con nombre.
Uniones discriminadas	Describe el tipo de Unión discriminada, un tipo cuyos valores pueden ser cualquiera de un conjunto de tipos posibles.
Funciones	Describe los valores de función.

TEMA	DESCRIPCIÓN
Clases	Describe el tipo de clase, un tipo de objeto que corresponde a un tipo de referencia de .NET. Los tipos de clase pueden contener miembros, propiedades, interfaces implementadas y un tipo base.
Estructuras	Describe el tipo de <code>struct</code> , un tipo de objeto que corresponde a un tipo de valor de .NET. Normalmente, el tipo de <code>struct</code> representa un agregado pequeño de datos.
Interfaces	Describe los tipos de interfaz, que son tipos que representan un conjunto de miembros que proporcionan ciertas funciones, pero que no contienen datos. Un tipo de interfaz debe ser implementado por un tipo de objeto para que sea útil.
Delegados	Describe el tipo de delegado, que representa una función como un objeto.
Enumeraciones	Describe los tipos de enumeración, cuyos valores pertenecen a un conjunto de valores con nombre.
Atributos	Describe los atributos, que se usan para especificar los metadatos de otro tipo.
Tipos de excepción	Describe las excepciones, que especifican la información de error.

Inferencia de tipos

23/10/2019 • 5 minutes to read • [Edit Online](#)

En este tema se describe F# cómo el compilador deduce los tipos de valores, variables, parámetros y valores devueltos.

Inferencia de tipos en general

La idea de la inferencia de tipos es que no es necesario especificar los tipos de F# construcciones excepto cuando el compilador no puede deducir el tipo de forma concluyente. Omitir la información de tipo explícita no F# significa que se trata de un lenguaje con establecimiento dinámico de F# tipos o que los valores de están débilmente tipados. F# es un lenguaje de tipo estático, lo que significa que el compilador deduce un tipo exacto para cada construcción durante la compilación. Si no hay información suficiente para que el compilador deduzca los tipos de cada construcción, debe proporcionar información de tipo adicional, normalmente agregando anotaciones de tipo explícito en algún lugar del código.

Inferencia de tipos de parámetro y de valor devuelto

En una lista de parámetros, no es necesario especificar el tipo de cada parámetro. Y, sin F# embargo, es un lenguaje de tipos estáticos y, por tanto, cada valor y expresión tiene un tipo definido en tiempo de compilación. En el caso de los tipos que no se especifican de forma explícita, el compilador deduce el tipo basándose en el contexto. Si el tipo no se especifica de otra forma, se deduce que es genérico. Si el código usa un valor de manera incoherente, de tal forma que no hay ningún tipo inferido único que cumpla todos los usos de un valor, el compilador informa de un error.

El tipo de valor devuelto de una función viene determinado por el tipo de la última expresión de la función.

Por ejemplo, en el código siguiente, se infiere `a` que `b` `int` los tipos de parámetro y y el tipo de valor devuelto son `100` porque el literal `int` es de tipo.

```
let f a b = a + b + 100
```

Puede influir en la inferencia de tipos cambiando los literales. `100` Si realiza la `uint32` adición del sufijo `u`, los tipos de `a` `b`, y el valor devuelto se deducen a `uint32`.

También puede influir en la inferencia de tipos mediante el uso de otras construcciones que impliquen restricciones en el tipo, como funciones y métodos que funcionan solo con un tipo determinado.

Además, puede aplicar anotaciones de tipo explícitos a parámetros de función o de método o a variables en expresiones, tal como se muestra en los ejemplos siguientes. Se producirán errores si se producen conflictos entre las distintas restricciones.

```
// Type annotations on a parameter.
let addu1 (x : uint32) y =
    x + y

// Type annotations on an expression.
let addu2 x y =
    (x : uint32) + y
```

También puede especificar explícitamente el valor devuelto de una función proporcionando una anotación de

tipo después de todos los parámetros.

```
let addu1 x y : uint32 =  
    x + y
```

Un caso común en el que una anotación de tipo es útil en un parámetro es cuando el parámetro es un tipo de objeto y se desea usar un miembro.

```
let replace(str: string) =  
    str.Replace("A", "a")
```

Generalización automática

Si el código de la función no depende del tipo de un parámetro, el compilador considera que el parámetro es genérico. Esto se denomina *generalización automática* y puede ser una ayuda eficaz para escribir código genérico sin aumentar la complejidad.

Por ejemplo, la función siguiente combina dos parámetros de cualquier tipo en una tupla.

```
let makeTuple a b = (a, b)
```

Se deduce que el tipo es

```
'a -> 'b -> 'a * 'b
```

Información adicional

La inferencia de tipos se describe con más detalle F# en la especificación del lenguaje.

Vea también

- [Generalización automática](#)

Tipos básicos

23/10/2019 • 2 minutes to read • [Edit Online](#)

Este tema enumeran los tipos básicos que se definen en el F# lenguaje. Estos tipos son las más importantes en F#, forman la base de casi todos F# programa. Son un superconjunto de los tipos primitivos. NET.

TIPO	TIPO DE .NET	DESCRIPCIÓN
<code>bool</code>	Boolean	Los valores posibles son <code>true</code> y <code>false</code> .
<code>byte</code>	Byte	Valores de 0 a 255.
<code>sbyte</code>	SByte	Valores de -128 a 127.
<code>int16</code>	Int16	Valores de -32768 a 32767.
<code>uint16</code>	UInt16	Valores de 0 a 65535.
<code>int</code>	Int32	Valores entre -2.147.483.648 a 2.147.483.647.
<code>uint32</code>	UInt32	Valores entre 0 y 4.294.967.295.
<code>int64</code>	Int64	Valores comprendidos entre -9.223.372.036.854.775.808 a +9.223.372.036.854.775.807.
<code>uint64</code>	UInt64	Valores de 0 a 18,446,744,073,709,551,615.
<code>nativeint</code>	IntPtr	Un puntero nativo como un entero con signo.
<code>unativeint</code>	UIntPtr	Un puntero nativo como un entero sin signo.
<code>char</code>	Char	Valores de caracteres Unicode.
<code>string</code>	String	Texto Unicode.
<code>decimal</code>	Decimal	Tipo de datos que tiene al menos 28 dígitos significativos de punto flotante.
<code>unit</code>	No aplicable	Indica la ausencia de un valor real. El tipo tiene un único valor formal, que se indica <code>()</code> . El valor de unidad, <code>()</code> , a menudo se usa como marcador de posición donde se necesita un valor pero ningún valor real está disponible o que tenga sentido.

TIPO	TIPO DE .NET	DESCRIPCIÓN
<code>void</code>	Void	No indica que ningún tipo de valor.
<code>float32</code> , <code>single</code>	Single	Tipo de punto flotante de 32 bits.
<code>float</code> , <code>double</code>	Double	Tipo de punto flotante de 64 bits.

NOTE

Puede realizar cálculos con números enteros demasiado grandes para el tipo de entero de 64 bits mediante el [bigint](#) tipo.

`bigint` no se considera un tipo básico; es la abreviatura de `System.Numerics.BigInteger`.

Vea también

- [Referencia del lenguaje F#](#)

Unit (Tipo)

04/11/2019 • 3 minutes to read • [Edit Online](#)

El tipo de `unit` es un tipo que indica la ausencia de un valor concreto; el tipo de `unit` solo tiene un valor único, que actúa como marcador de posición cuando no existe ningún otro valor o es necesario.

Sintaxis

```
// The value of the unit type.  
( )
```

Comentarios

Cada F# expresión debe evaluarse como un valor. En el caso de las expresiones que no generan un valor de interés, se utiliza el valor de tipo `unit`. El tipo de `unit` se parece al tipo de `void` en C# lenguajes C++ como y.

El tipo de `unit` tiene un valor único y ese valor se indica mediante el token `()`.

El valor del tipo de `unit` se usa a menudo F# en la programación para contener el lugar donde se requiere un valor en la sintaxis del lenguaje, pero cuando no se necesita ningún valor o no se desea. Un ejemplo podría ser el valor devuelto de una función `printf`. Dado que las acciones importantes de la operación de `printf` se producen en la función, la función no tiene que devolver un valor real. Por lo tanto, el valor devuelto es de tipo `unit`.

Algunas construcciones esperan un valor `unit`. Por ejemplo, se espera que un enlace de `do` o cualquier código en el nivel superior de un módulo se evalúe como un valor de `unit`. El compilador emite una advertencia cuando una `do` enlace o código en el nivel superior de un módulo produce un resultado distinto del valor de `unit` que no se utiliza, como se muestra en el ejemplo siguiente.

```
let function1 x y = x + y  
// The next line results in a compiler warning.  
function1 10 20  
// Changing the code to one of the following eliminates the warning.  
// Use this when you do want the return value.  
let result = function1 10 20  
// Use this if you are only calling the function for its side effects,  
// and do not want the return value.  
function1 10 20 |> ignore
```

Esta advertencia es una característica de la programación funcional; no aparece en otros lenguajes de programación de .NET. En un programa puramente funcional, en el que las funciones no tienen ningún efecto secundario, el valor devuelto final es el único resultado de una llamada de función. Por lo tanto, cuando se omite el resultado, es posible que se produzca un error de programación. Aunque F# no es un lenguaje de programación puramente funcional, se recomienda seguir el estilo de programación funcional siempre que sea posible.

Vea también

- [Rectangle](#)
- [Referencia del lenguaje F#](#)

Cadenas

21/04/2020 • 7 minutes to read • [Edit Online](#)

NOTE

Los vínculos de la referencia de API de este artículo le llevarán a MSDN. La referencia de API de docs.microsoft.com no está completa.

El `string` tipo representa texto inmutable como una secuencia de caracteres Unicode. `string` es un alias de `System.String` en .NET Framework.

Observaciones

Los literales de cadena están delimitados por el carácter de comillas ("). El carácter \ de barra diagonal inversa () se utiliza para codificar ciertos caracteres especiales. La barra diagonal inversa y el siguiente carácter juntos se conocen como una secuencia de *escape*. En la tabla siguiente se muestran las secuencias de escape admitidas en los literales de cadena de F.

CARÁCTER	SECUENCIA DE ESCAPE
Alerta	<code>\a</code>
Retroceso	<code>\b</code>
Avance de página	<code>\f</code>
Nueva línea	<code>\n</code>
Retorno de carro	<code>\r</code>
Pestaña	<code>\t</code>
Tabulación vertical	<code>\v</code>
Barra diagonal inversa	<code>\\</code>
Comillas	<code>\"</code>
Apóstrofo	<code>\'</code>
carácter Unicode	<code>\DDD</code> (donde <code>D</code> indica un dígito decimal; rango de 000 - 231 255; por ejemplo, á "o")
carácter Unicode	<code>\xHH</code> (donde <code>H</code> se indica un dígito hexadecimal; rango de 00 - FF; por ejemplo, á "o")

CARÁCTER	SECUENCIA DE ESCAPE
carácter Unicode	<code>\uHHHH</code> (UTF-16) (donde <code>H</code> indica un dígito hexadecimal; rango de 0000 - FFFF; por ejemplo, <code>\u00E7</code> "a")
carácter Unicode	<code>\U00HHHHHH</code> (UTF-32) (donde <code>H</code> indica un dígito hexadecimal; rango de 000000 - 10FFFF; por ejemplo, <code>\U0001F47D</code> 🐼 " ")

IMPORTANT

La `\DDD` secuencia de escape es notación decimal, no notación octal como en la mayoría de los otros idiomas. Por lo `89` tanto, los dígitos `\032` y son válidos, y una secuencia de representa un espacio (U+0020), mientras que ese mismo punto de código en notación octal sería `\040`.

NOTE

Al estar restringidos a un rango de 0 - `\DDD` `\x` 255 (0xFF), las secuencias y las secuencias de escape son efectivamente el juego de caracteres [ISO-8859-1](#), ya que coincide con los primeros 256 puntos de código Unicode.

Cuerdas literales

Si va precedido por el símbolo de la tecla ```, el literal es una cadena literal. Esto significa que se omiten las secuencias de escape, excepto que dos caracteres de comillas se interpretan como un carácter de comillas.

Cuerdas triples citadas

Además, una cadena puede estar entre comillas triples. En este caso, se omiten todas las secuencias de escape, incluidos los caracteres de comillas dobles. Para especificar una cadena que contenga una cadena entre comillas incrustada, puede usar una cadena literal o una cadena de comillas triples. Si utiliza una cadena literal, debe especificar dos caracteres de comillas para indicar un carácter de comillas simples. Si utiliza una cadena de comillas triples, puede usar los caracteres de comillas simples sin que se analicen como el final de la cadena. Esta técnica puede ser útil cuando se trabaja con XML u otras estructuras que incluyen comillas incrustadas.

```
// Using a verbatim string
let xmlFragment1 = @"<book author=""Milton, John"" title=""Paradise Lost"">"

// Using a triple-quoted string
let xmlFragment2 = """<book author="Milton, John" title="Paradise Lost">"""
```

En el código, se aceptan cadenas que tienen saltos de línea y los saltos de línea se interpretan literalmente como líneas nuevas, a menos que un carácter de barra diagonal inversa sea el último carácter antes del salto de línea. El espacio en blanco inicial en la siguiente línea se omite cuando se utiliza el carácter de barra diagonal inversa. El código siguiente genera `str1` una `"abc\ndef"` cadena que `str2` tiene `"abcdef"` valor y una cadena que tiene valor .

```
let str1 = "abc
def"
let str2 = "abc\
def"
```

Indización y segmentación de cadenas

Puede tener acceso a caracteres individuales en una cadena mediante la sintaxis de tipo matriz, como se indica a continuación.

```
printfn "%c" str1.[1]
```

El resultado es `b`

O bien, puede extraer subcadenas mediante la sintaxis de sector de matriz, como se muestra en el código siguiente.

```
printfn "%s" (str1.[0..2])
printfn "%s" (str2.[3..5])
```

La salida es la siguiente.

```
abc
def
```

Puede representar cadenas ASCII mediante matrices de `byte[]` bytes sin signo, escriba `.` Agregue el `b` sufijo a un literal de cadena para indicar que es una cadena ASCII. Los literales de cadena ASCII utilizados con matrices de bytes admiten las mismas secuencias de escape que las cadenas Unicode, excepto las secuencias de escape Unicode.

```
// "abc" interpreted as a Unicode string.
let str1 : string = "abc"
// "abc" interpreted as an ASCII byte array.
let bytearray : byte[] = "abc"B
```

Operadores de cuerdas

El `+` operador se puede usar para concatenar cadenas, manteniendo la compatibilidad con las características de control de cadenas de .NET Framework. En el ejemplo siguiente se muestra la concatenación de cadenas.

```
let string1 = "Hello, " + "world"
```

Clase String

Debido a que el tipo de cadena `System.String` en F# `System.String` es en realidad un tipo de .NET Framework, todos los miembros están disponibles. Esto incluye `+` el operador, que se utiliza `Length` para concatenar cadenas, la propiedad y la `Chars` propiedad, que devuelve la cadena como una matriz de caracteres Unicode. Para obtener más información `System.String` acerca de las cadenas, consulte [.](#)

Mediante el `Chars` uso `System.String` de la propiedad de `.`, puede tener acceso a los caracteres individuales de una cadena especificando un índice, como se muestra en el código siguiente.

```
let printChar (str : string) (index : int) =
    printfn "First character: %c" (str.Chars(index))
```

Módulo de cadena

La funcionalidad adicional para el `String` control de `FSharp.Core` cadenas se incluye en el módulo del espacio de nombres. Para obtener más información, vea [Core.String Module](#).

Consulte también

- [Referencia del lenguaje f](#)

Tuplas

23/07/2020 • 9 minutes to read • [Edit Online](#)

Una *tupla* es una agrupación de valores sin nombre pero ordenados, posiblemente de tipos diferentes. Las tuplas pueden ser tipos de referencia o Structs.

Sintaxis

```
(element, ... , element)
struct(element, ... ,element )
```

Observaciones

Cada *elemento* de la sintaxis anterior puede ser cualquier expresión de F # válida.

Ejemplos

Entre los ejemplos de tuplas se incluyen pares, tripas, etc., del mismo tipo o de tipos diferentes. En el código siguiente se muestran algunos ejemplos.

```
(1, 2)

// Triple of strings.
("one", "two", "three")

// Tuple of generic types.
(a, b)

// Tuple that has mixed types.
("one", 1, 2.0)

// Tuple of integer expressions.
(a + 1, b + 1)

// Struct Tuple of floats
struct (1.025f, 1.5f)
```

Obtener valores individuales

Puede usar la coincidencia de patrones para obtener acceso y asignar nombres para los elementos de tupla, como se muestra en el código siguiente.

```
let print tuple1 =
    match tuple1 with
    | (a, b) -> printfn "Pair %A %A" a b
```

También puede deconstruir una tupla a través de la coincidencia de patrones fuera de una `match` expresión a través del `let` enlace:

```
let (a, b) = (1, 2)

// Or as a struct
let struct (c, d) = struct (1, 2)
```

O bien, puede buscar coincidencias en tuplas como entradas en funciones:

```
let getDistance ((x1,y1): float*float) ((x2,y2): float*float) =
  // Note the ability to work on individual elements
  (x1*x2 - y1*y2)
  |> abs
  |> sqrt
```

Si solo necesita un elemento de la tupla, se puede usar el carácter comodín (el carácter de subrayado) para evitar la creación de un nuevo nombre para un valor que no necesite.

```
let (a, _) = (1, 2)
```

Copiar elementos de una tupla de referencia en una tupla de struct también es simple:

```
// Create a reference tuple
let (a, b) = (1, 2)

// Construct a struct tuple from it
let struct (c, d) = struct (a, b)
```

Las funciones `fst` y `snd` (solo tuplas de referencia) devuelven el primer y el segundo elemento de una tupla, respectivamente.

```
let c = fst (1, 2)
let d = snd (1, 2)
```

No hay ninguna función integrada que devuelva el tercer elemento de un triple, pero se puede escribir fácilmente como se indica a continuación.

```
let third (_, _, c) = c
```

Por lo general, es mejor usar la coincidencia de patrones para tener acceso a elementos de tupla individuales.

Usar tuplas

Las tuplas proporcionan una manera cómoda de devolver varios valores de una función, tal como se muestra en el ejemplo siguiente. Este ejemplo realiza una división de enteros y devuelve el resultado redondeado de la operación como un primer miembro de un par de tupla y el resto como un segundo miembro del par.

```
let divRem a b =
  let x = a / b
  let y = a % b
  (x, y)
```

Las tuplas también se pueden usar como argumentos de función cuando se desea evitar el currificación implícito de los argumentos de función que está implícito en la sintaxis de función habitual.

```
let sumNoCurry (a, b) = a + b
```

La sintaxis habitual para definir la función `let sum a b = a + b` permite definir una función que es la aplicación parcial del primer argumento de la función, tal y como se muestra en el código siguiente.

```
let sum a b = a + b

let addTen = sum 10
let result = addTen 95
// Result is 105.
```

El uso de una tupla como parámetro deshabilita currificación. Para obtener más información, vea "aplicación parcial de argumentos" en [funciones](#).

Nombres de tipos de tupla

Cuando se escribe el nombre de un tipo que es una tupla, se usa el `*` símbolo para separar los elementos. Para una tupla que consta de un, `int` un `float` y un `string`, como `(10, 10.0, "ten")`, el tipo se escribiría como se indica a continuación.

```
int * float * string
```

Interoperación con tuplas de C#

C# 7,0 presentó tuplas en el lenguaje. Las tuplas en C# son estructuras y son equivalentes a las tuplas de struct en F#. Si necesita interoperar con C#, debe utilizar tuplas de struct.

Esto es fácil de hacer. Por ejemplo, Imagine que tiene que pasar una tupla a una clase de C# y luego consumir su resultado, que también es una tupla:

```
namespace CSharpTupleInterop
{
    public static class Example
    {
        public static (int, int) AddOneToXAndY((int x, int y) a) =>
            (a.x + 1, a.y + 1);
    }
}
```

En el código de F#, puede pasar una tupla de struct como parámetro y consumir el resultado como una tupla de estructura.

```
open TupleInterop

let struct (newX, newY) = Example.AddOneToXAndY(struct (1, 2))
// newX is now 2, and newY is now 3
```

Convertir entre tuplas de referencia y tuplas de struct

Dado que las tuplas de referencia y las tuplas de struct tienen una representación subyacente completamente diferente, no se pueden convertir implícitamente. Es decir, el código como el siguiente no se compilará:

```
// Will not compile!  
let (a, b) = struct (1, 2)  
  
// Will not compile!  
let struct (c, d) = (1, 2)  
  
// Won't compile!  
let f(t: struct(int*int)): int*int = t
```

Debe crear patrones de coincidencia en una tupla y construir la otra con las partes constituyentes. Por ejemplo:

```
// Pattern match on the result.  
let (a, b) = (1, 2)  
  
// Construct a new tuple from the parts you pattern matched on.  
let struct (c, d) = struct (a, b)
```

Forma compilada de tuplas de referencia

En esta sección se explica la forma de las tuplas cuando se compilan. Esta información no es necesaria para leer a menos que tenga como destino .NET Framework 3,5 o inferior.

Las tuplas se compilan en objetos de uno de varios tipos genéricos, todos los nombres `System.Tuple`, que están sobrecargados en la aridad o el número de parámetros de tipo. Los tipos de tupla aparecen en este formulario cuando se ven desde otro lenguaje, como C# o Visual Basic, o cuando se usa una herramienta que no es consciente de las construcciones de F#. Los `Tuple` tipos se introdujeron en .NET Framework 4. Si el destino es una versión anterior del .NET Framework, el compilador usa versiones de `System.Tuple` de la versión 2,0 de la biblioteca básica de F#. Los tipos de esta biblioteca solo se usan para las aplicaciones destinadas a las versiones 2,0, 3,0 y 3,5 del .NET Framework. El reenvío de tipos se usa para garantizar la compatibilidad binaria entre .NET Framework 2,0 y .NET Framework 4 componentes de F#.

Forma compilada de tuplas de struct

Las tuplas de estructura (por ejemplo, `struct (x, y)`), son fundamentalmente distintas de las tuplas de referencia. Se compilan en el `ValueTuple` tipo, sobrecargado por aridad o el número de parámetros de tipo. Son equivalentes a las tuplas de C# 7,0 y Visual Basic las tuplas 2017, e interoperan bidireccionalmente.

Vea también

- [Referencia del lenguaje F#](#)
- [Tipos en F#](#)

Tipos de colección F#

27/02/2020 • 32 minutes to read • [Edit Online](#)

Al revisar este tema, puede determinar qué F# tipo de colección se adapta mejor a una necesidad concreta. Estos tipos de colección se diferencian de los tipos de colección en el .NET Framework, como los del espacio de nombres `System.Collections.Generic` F#, en que los tipos de colección están diseñados desde una perspectiva de programación funcional en lugar de una perspectiva orientada a objetos. Más concretamente, solo la colección de matrices tiene elementos mutables. Por lo tanto, cuando se modifica una colección, se crea una instancia de la colección modificada en lugar de modificar la colección original.

Los tipos de colección también difieren en el tipo de estructura de datos en el que se almacenan los objetos. Las estructuras de datos como las tablas hash, las listas vinculadas y las matrices tienen diferentes características de rendimiento y un conjunto diferente de operaciones disponibles.

Tipos de colección F#

En la tabla siguiente F# se muestran los tipos de colección.

TIPO	DESCRIPCIÓN	VÍNCULOS RELACIONADOS
Lista	Una serie ordenada e inmutable de elementos del mismo tipo. Se implementa como una lista vinculada.	Listas Módulo de lista
Array	Colección mutable de tamaño fijo y de base cero de elementos de datos consecutivos que son del mismo tipo.	Matrices Módulo de matriz Módulo Array2D Módulo Array3D
Próx	Serie lógica de elementos que son todos de un tipo. Las secuencias son especialmente útiles cuando se tiene una colección grande ordenada de datos, pero no es necesario utilizar todos los elementos. Los elementos de secuencia individuales solo se calculan según sea necesario, por lo que una secuencia puede funcionar mejor que una lista si no se utilizan todos los elementos. Las secuencias se representan mediante el tipo de <code>seq<'T></code> , que es un alias de <code>IEnumerable<'T></code> . Por lo tanto, cualquier tipo de .NET Framework que implementa <code>System.Collections.Generic.IEnumerable<'T></code> puede usarse como una secuencia.	Secuencias Seq (módulo)
Map	Diccionario inmutable de elementos. Se obtiene acceso a los elementos por clave.	Módulo de asignación
Establecimiento	Conjunto inmutable que se basa en árboles binarios, donde la comparación F# es la función de comparación estructural, que potencialmente utiliza implementaciones de la interfaz <code>System.IComparable</code> en valores de clave.	Establecer módulo

Tabla de funciones

En esta sección se comparan las funciones que F# están disponibles en los tipos de colección. Se proporciona la complejidad computacional de la función, donde N es el tamaño de la primera colección y M es el tamaño de la segunda colección, si existe. Un guión (-) indica que esta función no está disponible en la colección. Dado que las secuencias se evalúan de forma diferida, una función como `seq.DISTINCT` puede ser O(1) porque vuelve inmediatamente, aunque todavía afecta al rendimiento de la secuencia cuando se enumera.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
---------	-------	------	-----------	-----	-----	-------------

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
append	$O(N)$	$O(N)$	$O(N)$	-	-	Devuelve una nueva colección que contiene los elementos de la primera colección seguidos de los elementos de la segunda colección.
add	-	-	-	$O(\log N)$	$O(\log N)$	Devuelve una nueva colección con el elemento agregado.
average	$O(N)$	$O(N)$	$O(N)$	-	-	Devuelve el promedio de los elementos de la colección.
averageBy	$O(N)$	$O(N)$	$O(N)$	-	-	Devuelve el promedio de los resultados de la función proporcionada que se aplica a cada elemento.
fundir	$O(N)$	-	-	-	-	Copia una sección de una matriz.
caché	-	-	$O(N)$	-	-	Calcula y almacena elementos de una secuencia.
conversión	-	-	$O(N)$	-	-	Convierte los elementos en el tipo especificado.
choose	$O(N)$	$O(N)$	$O(N)$	-	-	Aplica la función especificada <code>f</code> a cada elemento <code>x</code> de la lista. Devuelve la lista que contiene los resultados de cada elemento en el que la función devuelve <code>Some(f(x))</code> .
collect	$O(N)$	$O(N)$	$O(N)$	-	-	Aplica la función especificada a cada elemento de la colección, concatena todos los resultados y devuelve la lista combinada.
compareWith (-	-	$O(N)$	-	-	Compara dos secuencias mediante la función de comparación especificada, elemento a elemento.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
concat	O (N)	O (N)	O (N)	-	-	Combina la enumeración de enumeraciones dada como una sola enumeración concatenada.
contains	-	-	-	-	O (log N)	Devuelve true si el conjunto contiene el elemento especificado.
containsKey	-	-	-	O (log N)	-	Comprueba si un elemento está en el dominio de un mapa.
count	-	-	-	-	O (N)	Devuelve el número de elementos del conjunto.
countBy	-	-	O (N)	-	-	Aplica una función de generación de claves a cada elemento de una secuencia y devuelve una secuencia que produce claves únicas y su número de apariciones en la secuencia original.
copy	O (N)	-	O (N)	-	-	Copia la colección.
create	O (N)	-	-	-	-	Crea una matriz de elementos completos que son todos inicialmente el valor especificado.
delay	-	-	O (1)	-	-	Devuelve una secuencia que se genera a partir de la especificación retrasada especificada de una secuencia.
diferencia	-	-	-	-	O (Registro * M N)	Devuelve un nuevo conjunto con los elementos del segundo conjunto que se han quitado del primer conjunto.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
distinct			$O(1)^*$			Devuelve una secuencia que no contiene ninguna entrada duplicada según las comparaciones de igualdad y hash genéricas en las entradas. Si un elemento aparece varias veces en la secuencia, se descartan las repeticiones posteriores.
Distinctby ($O(1)^*$			Devuelve una secuencia que no contiene ninguna entrada duplicada según las comparaciones de igualdad y hash genéricas de las claves que devuelve la función de generación de claves determinada. Si un elemento aparece varias veces en la secuencia, se descartan las repeticiones posteriores.
empty	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Crea una colección vacía.
exists	$O(N)$	$O(N)$	$O(N)$	$O(\log N)$	$O(\log N)$	Comprueba si algún elemento de la secuencia satisface el predicado especificado.
exists2 ($O(\min(N, M))$	-	$O(\min(N, M))$			Comprueba si algún par de elementos correspondientes de las secuencias de entrada satisface el predicado especificado.
fill	$O(N)$					Establece un intervalo de elementos de la matriz en el valor especificado.
filter	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	Devuelve una nueva colección que contiene solo los elementos de la colección para los que el predicado especificado devuelve <code>true</code> .

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
find	O (N)	O (N)	O (N)	O (log N)	-	Devuelve el primer elemento para el que la función especificada devuelve <code>true</code> . Devuelve <code>System.Collections.Generic</code> si no existe ningún elemento de este tipo.
findIndex	O (N)	O (N)	O (N)	-	-	Devuelve el índice del primer elemento de la matriz que cumple el predicado especificado. Genera <code>System.Collections.Generic</code> si ningún elemento satisface el predicado.
findKey	-	-	-	O (log N)	-	Evalúa la función en cada asignación de la colección y devuelve la clave de la primera asignación en la que la función devuelve <code>true</code> . Si no existe ningún elemento de este tipo, esta función genera <code>System.Collections.Generic</code> .
doblar	O (N)	O (N)	O (N)	O (N)	O (N)	Aplica una función a cada elemento de la colección y subprocesa un argumento acumulador a través del cálculo. Si la función de entrada es f y los elementos son I0... En, esta función calcula f (... (f s I0)...) de.
fold2 (O (N)	O (N)	-	-	-	Aplica una función a los elementos correspondientes de dos colecciones y subprocesa un argumento acumulador a través del cálculo. Las colecciones deben tener tamaños idénticos. Si la función de entrada es f y los elementos son I0... En y J0... jN, esta función calcula f (... (f s I0 J0)...) En jN.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
Foldback (O (N)	O (N)	-	O (N)	O (N)	Aplica una función a cada elemento de la colección y subprocesa un argumento acumulador a través del cálculo. Si la función de entrada es f y los elementos son l0... En, esta función calcula f l0 (... (f en s)).
Foldback2 (O (N)	O (N)	-	-	-	Aplica una función a los elementos correspondientes de dos colecciones y subprocesa un argumento acumulador a través del cálculo. Las colecciones deben tener tamaños idénticos. Si la función de entrada es f y los elementos son l0... En y j0... jN, esta función calcula j0 de f l0 (... (f en jN s)).
forall	O (N)	O (N)	O (N)	O (N)	O (N)	Comprueba si todos los elementos de la colección satisfacen el predicado especificado.
forall2 (O (N)	O (N)	O (N)	-	-	Comprueba si todos los elementos correspondientes de la colección satisfacen el predicado especificado en pares.
obtener/n	O (1)	O (N)	O (N)	-	-	Devuelve un elemento de la colección a partir de su índice.
head	-	O (1)	O (1)	-	-	Devuelve el primer elemento de la colección.
init	O (N)	O (N)	O (1)	-	-	Crea una colección a partir de la dimensión y una función de generador para calcular los elementos.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
initInfinite	-	-	O (1)	-	-	Genera una secuencia que, cuando se itera, devuelve elementos sucesivos mediante una llamada a la función especificada.
Intersect	-	-	-	-	O (log N * log M)	Calcula la intersección de dos conjuntos.
intersectMany	-	-	-	-	O (N1 * N2...)	Calcula la intersección de una secuencia de conjuntos. La secuencia no debe estar vacía.
Vacío	O (1)	O (1)	O (1)	O (1)	-	Devuelve <code>true</code> si la colección está vacía.
isProperSubset	-	-	-	-	O (Registro * M N)	Devuelve <code>true</code> si todos los elementos del primer conjunto se encuentran en el segundo conjunto y al menos un elemento del segundo conjunto no está en el primer conjunto.
isProperSuperset	-	-	-	-	O (Registro * M N)	Devuelve <code>true</code> si todos los elementos del segundo conjunto se encuentran en el primer conjunto y al menos un elemento del primer conjunto no está en el segundo conjunto.
isSubset	-	-	-	-	O (Registro * M N)	Devuelve <code>true</code> si todos los elementos del primer conjunto se encuentran en el segundo conjunto.
isSuperset	-	-	-	-	O (Registro * M N)	Devuelve <code>true</code> si todos los elementos del segundo conjunto se encuentran en el primer conjunto.
ITER	O (N)	O (N)	O (N)	O (N)	O (N)	Aplica la función especificada a cada elemento de la colección.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
ITERI (O (N)	O (N)	O (N)	-	-	Aplica la función especificada a cada elemento de la colección. El entero que se pasa a la función indica el índice del elemento.
iteri2	O (N)	O (N)	-	-	-	Aplica la función especificada a un par de elementos que se dibujan desde índices coincidentes en dos matrices. El entero que se pasa a la función indica el índice de los elementos. Las dos matrices deben tener la misma longitud.
iter2 (O (N)	O (N)	O (N)	-	-	Aplica la función especificada a un par de elementos que se dibujan desde índices coincidentes en dos matrices. Las dos matrices deben tener la misma longitud.
last	O (1)	O (N)	O (N)	-	-	Devuelve el último elemento de la colección correspondiente.
length	O (1)	O (N)	O (N)	-	-	Devuelve el número de elementos de la colección.
map	O (N)	O (N)	O (1)	-	-	Compila una colección cuyos elementos son los resultados de aplicar la función especificada a cada elemento de la matriz.
MAP2 (O (N)	O (N)	O (1)	-	-	Compila una colección cuyos elementos son los resultados de aplicar la función especificada a los elementos correspondientes de las dos colecciones en pares. Las dos matrices de entrada deben tener la misma longitud.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
map3 (-	O (N)	-	-	-	Compila una colección cuyos elementos son los resultados de aplicar la función especificada a los elementos correspondientes de las tres colecciones simultáneamente.
interfaz	O (N)	O (N)	O (N)	-	-	Crea una matriz cuyos elementos son los resultados de aplicar la función especificada a cada elemento de la matriz. El índice de entero que se pasa a la función indica el índice del elemento que se va a transformar.
mapi2	O (N)	O (N)	-	-	-	Compila una colección cuyos elementos son los resultados de aplicar la función especificada a los elementos correspondientes de las dos colecciones en pares, pasando también el índice de los elementos. Las dos matrices de entrada deben tener la misma longitud.
max	O (N)	O (N)	O (N)	-	-	Devuelve el mayor elemento de la colección, comparado mediante el operador Max .
Maxby (O (N)	O (N)	O (N)	-	-	Devuelve el mayor elemento de la colección, en comparación con el uso de Max en el resultado de la función.
maxElement (-	-	-	-	O (log N)	Devuelve el mayor elemento del conjunto según la ordenación usada para el conjunto.
Min	O (N)	O (N)	O (N)	-	-	Devuelve el elemento mínimo de la colección, comparado mediante el operador min .

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
minBy (O (N)	O (N)	O (N)	-	-	Devuelve el elemento mínimo de la colección, comparado mediante el operador <code>min</code> en el resultado de la función.
Minelement (-	-	-	-	O (log N)	Devuelve el elemento más bajo del conjunto según la ordenación que se usa para el conjunto.
Ofarray (-	O (N)	O (1)	O (N)	O (N)	Crea una colección que contiene los mismos elementos que la matriz especificada.
Oflist (O (N)	-	O (1)	O (N)	O (N)	Crea una colección que contiene los mismos elementos que la lista especificada.
ofSeq	O (N)	O (N)	-	O (N)	O (N)	Crea una colección que contiene los mismos elementos que la secuencia especificada.
quería	-	-	O (N)	-	-	Devuelve una secuencia de cada elemento en la secuencia de entrada y su antecesor, excepto para el primer elemento, que solo se devuelve como predecesor del segundo elemento.
partición	O (N)	O (N)	-	O (N)	O (N)	Divide la colección en dos colecciones. La primera colección contiene los elementos para los que el predicado especificado devuelve <code>true</code> , y la segunda colección contiene los elementos para los que el predicado especificado devuelve <code>false</code> .
permute (O (N)	O (N)	-	-	-	Devuelve una matriz con todos los elementos permutados según la permutación especificada.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
elegir	O (N)	O (N)	O (N)	O (log N)	-	<p>Aplica la función especificada a elementos sucesivos y devuelve el primer resultado en el que la función devuelve some. Si la función nunca devuelve some, se genera</p> <p><code>System.Collections.Gener</code></p> <p>.</p>
readonly	-	-	O (N)	-	-	<p>Crea un objeto de secuencia que delega en el objeto de secuencia especificado. Esta operación garantiza que una conversión de tipos no puede volver a detectar y mutar la secuencia original. Por ejemplo, si se proporciona una matriz, la secuencia devuelta devolverá los elementos de la matriz, pero no se puede convertir el objeto de secuencia devuelto en una matriz.</p>
reduce	O (N)	O (N)	O (N)	-	-	<p>Aplica una función a cada elemento de la colección y subprocesa un argumento acumulador a través del cálculo. Esta función comienza aplicando la función a los dos primeros elementos, pasa este resultado a la función junto con el tercer elemento, y así sucesivamente. La función devuelve el resultado final.</p>
Reduceback (O (N)	O (N)	-	-	-	<p>Aplica una función a cada elemento de la colección y subprocesa un argumento acumulador a través del cálculo. Si la función de entrada es f y los elementos son I0... En, esta función calcula f I0 (... (f en-1 iN)).</p>

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
remove	-	-	-	$O(\log N)$	$O(\log N)$	Quita un elemento del dominio de la asignación. No se produce ninguna excepción si el elemento no está presente.
réplica	-	$O(N)$	-	-	-	Crea una lista de una longitud especificada con cada elemento establecido en el valor especificado.
Rev	$O(N)$	$O(N)$	-	-	-	Devuelve una nueva lista con los elementos en orden inverso.
revisar	$O(N)$	$O(N)$	$O(N)$	-	-	Aplica una función a cada elemento de la colección y subprocesa un argumento acumulador a través del cálculo. Esta operación aplica la función al segundo argumento y al primer elemento de la lista. A continuación, la operación pasa este resultado a la función junto con el segundo elemento, etc. Por último, la operación devuelve la lista de resultados intermedios y el resultado final.
ScanBack ($O(N)$	$O(N)$	-	-	-	Se parece a la operación foldBack pero devuelve los resultados intermedios y finales.
singleton	-	-	$O(1)$	-	$O(1)$	Devuelve una secuencia que produce un solo elemento.
set	$O(1)$	-	-	-	-	Establece un elemento de una matriz en el valor especificado.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
skip	-	-	O (N)	-	-	Devuelve una secuencia que omite N elementos de la secuencia subyacente y, a continuación, produce los elementos restantes de la secuencia.
Skipwhile (-	-	O (N)	-	-	Devuelve una secuencia que, cuando se itera, omite los elementos de la secuencia subyacente mientras el predicado especificado devuelve <code>true</code> y, a continuación, produce los elementos restantes de la secuencia.
sort	Promedio de O (N log N) O (N ^ 2) peor de los casos	O (N log N)	O (N log N)	-	-	Ordena la colección por valor de elemento. Los elementos se comparan mediante Compare .
sortBy (Promedio de O (N log N) O (N ^ 2) peor de los casos	O (N log N)	O (N log N)	-	-	Ordena la lista especificada mediante el uso de claves que proporciona la proyección especificada. Las claves se comparan mediante Compare .
Sortinplace (Promedio de O (N log N) O (N ^ 2) peor de los casos	-	-	-	-	Ordena los elementos de una matriz mediante la mutación y el uso de la función de comparación especificada. Los elementos se comparan mediante Compare .
sortInPlaceBy	Promedio de O (N log N) O (N ^ 2) peor de los casos	-	-	-	-	Ordena los elementos de una matriz mediante la mutación y el uso de la proyección especificada para las claves. Los elementos se comparan mediante Compare .

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
sortInPlaceWith	Promedio de $O(N \log N)$ $O(N^2)$ peor de los casos	-	-	-	-	Ordena los elementos de una matriz mediante la mutación y el uso de la función de comparación especificada como el orden.
sortWith (Promedio de $O(N \log N)$ $O(N^2)$ peor de los casos	$O(N \log N)$	-	-	-	Ordena los elementos de una colección, usando la función de comparación especificada como el orden y devuelve una nueva colección.
sub	$O(N)$	-	-	-	-	Crea una matriz que contiene el subintervalo dado especificado por el índice de inicio y la longitud.
Sum	$O(N)$	$O(N)$	$O(N)$	-	-	Devuelve la suma de los elementos de la colección.
sumBy	$O(N)$	$O(N)$	$O(N)$	-	-	Devuelve la suma de los resultados que se generan aplicando la función a cada elemento de la colección.
cola	-	$O(1)$	-	-	-	Devuelve la lista sin su primer elemento.
take	-	-	$O(N)$	-	-	Devuelve los elementos de la secuencia hasta un recuento especificado.
takeWhile	-	-	$O(1)$	-	-	Devuelve una secuencia que, cuando se itera, proporciona elementos de la secuencia subyacente mientras el predicado especificado devuelve <code>true</code> y, a continuación, no devuelve ningún elemento más.
toArray (-	$O(N)$	$O(N)$	$O(N)$	$O(N)$	Crea una matriz a partir de la colección especificada.
toList	$O(N)$	-	$O(N)$	$O(N)$	$O(N)$	Crea una lista a partir de la colección especificada.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
toSeq	O (1)	O (1)	-	O (1)	O (1)	Crea una secuencia a partir de la colección especificada.
truncate	-	-	O (1)	-	-	Devuelve una secuencia que, cuando se enumera, no devuelve más de N elementos.
tryFind	O (N)	O (N)	O (N)	O (log N)	-	Busca un elemento que satisface un predicado determinado.
tryFindIndex	O (N)	O (N)	O (N)	-	-	Busca el primer elemento que cumple un predicado especificado y devuelve el índice del elemento coincidente, o bien <code>None</code> si no existe ese elemento.
tryFindKey	-	-	-	O (log N)	-	Devuelve la clave de la primera asignación de la colección que satisface el predicado especificado o devuelve <code>None</code> si no existe ese elemento.
tryPick (O (N)	O (N)	O (N)	O (log N)	-	Aplica la función especificada a elementos sucesivos y devuelve el primer resultado en el que la función devuelve <code>Some</code> para algún valor. Si no existe ningún elemento de este tipo, la operación devuelve <code>None</code> .
Fold	-	-	O (N)	-	-	Devuelve una secuencia que contiene los elementos generados por el cálculo especificado.
union	-	-	-	-	O (Registro * M N)	Calcula la Unión de los dos conjuntos.
unionMany (-	-	-	-	O (N1 * N2...)	Calcula la Unión de una secuencia de conjuntos.

FUNCIÓN	ARRAY	LIST	SECUENCIA	MAP	SET	DESCRIPCIÓN
unzip	O (N)	O (N)	O (N)	-	-	Divide una lista de pares en dos listas.
unzip3 (O (N)	O (N)	O (N)	-	-	Divide una lista de triples en tres listas.
ventanas	-	-	O (N)	-	-	Devuelve una secuencia que produce ventanas deslizantes de elementos contenedores que se extraen de la secuencia de entrada. Cada ventana se devuelve como una nueva matriz.
zip	O (N)	O (N)	O (N)	-	-	Combina las dos colecciones en una lista de pares. Las dos listas deben tener la misma longitud.
zip3 (O (N)	O (N)	O (N)	-	-	Combina las tres colecciones en una lista de triples. Las listas deben tener la misma longitud.

Consulte también

- [Tipos en F#](#)
- [Referencia del lenguaje F#](#)

Listas

23/10/2019 • 38 minutes to read • [Edit Online](#)

NOTE

Los vínculos de la referencia de API de este artículo le llevarán a MSDN. La referencia de API de docs.microsoft.com no está completa.

En F#, una lista es una serie ordenada e inmutable de elementos del mismo tipo. Para realizar operaciones básicas en listas, use las funciones del [módulo de lista](#).

Crear e inicializar listas

Para definir una lista, puede enumerar explícitamente los elementos, separados por punto y coma y escritos entre corchetes, tal y como se muestra en la línea de código siguiente.

```
let list123 = [ 1; 2; 3 ]
```

También puede insertar saltos de línea entre los elementos, en cuyo caso el signo de punto y coma es opcional. La última sintaxis produce un código más legible cuando las expresiones de inicialización de elementos son más largas o si quiere incluir un comentario para cada elemento.

```
let list123 = [  
    1  
    2  
    3 ]
```

Normalmente, todos los elementos de lista deben ser del mismo tipo. Una excepción es que una lista en la cual los elementos se han especificado para que sean de un tipo base puede tener elementos que sean de tipos derivados. Por lo tanto, lo siguiente es aceptable, porque tanto `Button` como `CheckBox` derivan de `Control`.

```
let myControlList : Control list = [ new Button(); new CheckBox() ]
```

También puede definir elementos de lista usando un intervalo indicado por enteros separados por el operador de intervalo (`..`), tal y como se muestra en el código siguiente.

```
let list1 = [ 1 .. 10 ]
```

Una lista vacía se especifica por un par de corchetes con nada entre ellos.

```
// An empty list.  
let listEmpty = []
```

También puede usar una expresión de secuencia para crear una lista. Vea [expresiones de secuencia](#) para obtener más información. Por ejemplo, el código siguiente crea una lista de cuadrados de enteros, de 1 a 10.

```
let listOfSquares = [ for i in 1 .. 10 -> i*i ]
```

Operadores para trabajar con listas

Puede asociar elementos a una lista usando el operador `::` (cons). Si `list1` es `[2; 3; 4]`, el código siguiente crea `list2` como `[100; 2; 3; 4]`.

```
let list2 = 100 :: list1
```

Para concatenar listas que tengan tipos compatibles, puede usar el operador `@`, como en el código siguiente. Si `list1` es `[2; 3; 4]` y `list2` es `[100; 2; 3; 4]`, este código crea `list3` como `[2; 3; 4; 100; 2; 3; 4]`.

```
let list3 = list1 @ list2
```

Las funciones para realizar operaciones en listas están disponibles en el [módulo de lista](#).

Como en F# las listas son inmutables, las operaciones de modificación generan nuevas listas en lugar de modificar las existentes.

Las listas F# se implementan como listas vinculadas individualmente, lo que significa que las operaciones que tienen acceso solo al encabezado de la lista son $O(1)$ y el acceso a los elementos es $O(n)$.

Propiedades

El tipo de lista admite las siguientes propiedades:

PROPIEDAD	ESCRIBA	DESCRIPCIÓN
Head	<code>'T</code>	El primer elemento.
Vacía	<code>'T list</code>	Propiedad estática que devuelve una lista vacía del tipo apropiado.
Vacío	<code>bool</code>	<code>true</code> si la lista no tiene ningún elemento.
Elemento	<code>'T</code>	Elemento en el índice especificado (base cero).
Longitud	<code>int</code>	Número de elementos.
Cola	<code>'T list</code>	La lista sin el primer elemento.

Los siguientes son algunos ejemplos de cómo usar estas propiedades.

```
let list1 = [ 1; 2; 3 ]

// Properties
printfn "list1.IsEmpty is %b" (list1.IsEmpty)
printfn "list1.Length is %d" (list1.Length)
printfn "list1.Head is %d" (list1.Head)
printfn "list1.Tail.Head is %d" (list1.Tail.Head)
printfn "list1.Tail.Tail.Head is %d" (list1.Tail.Tail.Head)
printfn "list1.Item(1) is %d" (list1.Item(1))
```

Usar listas

Programar con listas permite realizar operaciones complejas con una pequeña cantidad de código. En esta sección se describen las operaciones comunes en las listas que son importantes para la programación funcional.

Recursión con listas

Las listas están especialmente indicadas para las técnicas de programación recursiva. Tomemos una operación que deba realizarse en todos los elementos de una lista. Puede hacerlo recursivamente ejecutando una operación en el encabezado de la lista y, después, devolviendo la cola de la lista (que es una lista más pequeña formada por la lista original sin el primer elemento) de nuevo al siguiente nivel de recursión.

Para escribir una función recursiva de este estilo, se usa el operador `cons (::)` en coincidencia de patrones, lo que permite separar el encabezado de una lista de la cola.

En el ejemplo de código siguiente se muestra cómo usar la coincidencia de patrones para implementar una función recursiva que realice operaciones en una lista.

```
let rec sum list =
    match list with
    | head :: tail -> head + sum tail
    | [] -> 0
```

El código anterior funciona bien para listas pequeñas, pero en listas mayores podría desbordar la pila. El código siguiente mejora este código usando un argumento acumulador, una técnica estándar para trabajar con funciones recursivas. El uso del argumento acumulador hace que la función sea recursiva en la cola, lo que ahorra espacio en la pila.

```
let sum list =
    let rec loop list acc =
        match list with
        | head :: tail -> loop tail (acc + head)
        | [] -> acc
    loop list 0
```

La función `RemoveAllMultiples` es una función recursiva que toma dos listas. La primera lista contiene los números cuyos múltiplos se van a quitar, y la segunda es la lista de la que se van a quitar los números. El código del ejemplo siguiente usa esta función recursiva para eliminar de una lista todos los números que no sean primos, dejando como resultado una lista de números primos.

```

let IsPrimeMultipleTest n x =
    x = n || x % n <> 0

let rec RemoveAllMultiples listn listx =
    match listn with
    | head :: tail -> RemoveAllMultiples tail (List.filter (IsPrimeMultipleTest head) listx)
    | [] -> listx

let GetPrimesUpTo n =
    let max = int (sqrt (float n))
    RemoveAllMultiples [ 2 .. max ] [ 1 .. n ]

printfn "Primes Up To %d:\n %A" 100 (GetPrimesUpTo 100)

```

La salida es la siguiente:

```

Primes Up To 100:
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71; 73; 79; 83; 89; 97]

```

Funciones del módulo

El [módulo de lista](#) proporciona funciones que tienen acceso a los elementos de una lista. El elemento de encabezado es la manera más rápida y sencilla de acceder. Use el [encabezado](#) de propiedad o la función de módulo [List.Head](#). Puede tener acceso al final de una lista mediante la propiedad [tail](#) o la función [List.tail](#). Para buscar un elemento por índice, use la función [List.nth](#). `List.nth` atraviesa la lista. Por lo tanto, es $O(n)$. Si su código usa `List.nth` con frecuencia, quizás quiera considerar la posibilidad de usar una matriz en lugar de una lista. El acceso a elementos en las matrices es $O(1)$.

Operaciones booleanas en listas

La función [List.isEmpty](#) determina si una lista tiene elementos.

La función [List.exists](#) aplica una prueba booleana a los elementos de una lista `true` y devuelve si algún elemento cumple la prueba. [List.exists2](#) (es similar pero funciona en pares sucesivos de elementos de dos listas).

En el código siguiente se muestra cómo usar `List.exists`.

```

// Use List.exists to determine whether there is an element of a list satisfies a given Boolean
expression.
// containsNumber returns true if any of the elements of the supplied list match
// the supplied number.
let containsNumber number list = List.exists (fun elem -> elem = number) list
let list0to3 = [0 .. 3]
printfn "For list %A, contains zero is %b" list0to3 (containsNumber 0 list0to3)

```

La salida es la siguiente:

```

For list [0; 1; 2; 3], contains zero is true

```

El siguiente ejemplo muestra el uso de `List.exists2`.


```
// Use List.exists2 to compare elements in two lists.
// isEqualElement returns true if any elements at the same position in two supplied
// lists match.
let isEqualElement list1 list2 = List.exists2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
let list1to5 = [ 1 .. 5 ]
let list5to1 = [ 5 .. -1 .. 1 ]
if (isEqualElement list1to5 list5to1) then
    printfn "Lists %A and %A have at least one equal element at the same position." list1to5 list5to1
else
    printfn "Lists %A and %A do not have an equal element at the same position." list1to5 list5to1
```

La salida es la siguiente:

```
Lists [1; 2; 3; 4; 5] and [5; 4; 3; 2; 1] have at least one equal element at the same position.
```

Puede usar [List.ForAll](#) si desea comprobar si todos los elementos de una lista cumplen una condición.

```
let isAllZeroes list = List.forall (fun elem -> elem = 0.0) list
printfn "%b" (isAllZeroes [0.0; 0.0])
printfn "%b" (isAllZeroes [0.0; 1.0])
```

La salida es la siguiente:

```
true
false
```

De forma similar, [List.forall2](#) (determina si todos los elementos de las posiciones correspondientes de dos listas satisfacen una expresión booleana que implica cada par de elementos.

```
let listEqual list1 list2 = List.forall2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
printfn "%b" (listEqual [0; 1; 2] [0; 1; 2])
printfn "%b" (listEqual [0; 0; 0] [0; 1; 0])
```

La salida es la siguiente:

```
true
false
```

Operaciones de ordenación en listas

Las funciones [List.Sort](#), [List.sortBy](#) y [List.sortWith](#) (ordenan las listas. La función de ordenación determina cuál de estas tres funciones se usará. `List.sort` usa una comparación genérica predeterminada. La comparación genérica usa operadores globales basados en la función de comparación genérica para comparar valores. Funciona de forma eficaz con una amplia variedad de tipos de elemento, como tipos numéricos simples, tuplas, registros, uniones discriminadas, listas, matrices y cualquier tipo que implemente `System.IComparable`. Para los tipos que implementan `System.IComparable`, la comparación genérica usa la función `System.IComparable.CompareTo()`. La comparación genérica también trabaja con cadenas, pero usa una ordenación independiente de la referencia cultural. La comparación genérica no se debe usar en tipos no admitidos, como los tipos de función. Además, el rendimiento de la comparación genérica predeterminada es mejor para tipos estructurados pequeños; para los tipos estructurados mayores que deben compararse y ordenarse con frecuencia, considere la posibilidad de implementar `System.IComparable` y de proporcionar una implementación eficaz del método `System.IComparable.CompareTo()`.

`List.sortBy` toma una función que devuelve un valor que se usa como criterio de ordenación, y

`List.sortWith` toma una función de comparación como argumento. Estas dos últimas funciones son útiles cuando se trabaja con tipos que no permiten la comparación, o cuando la comparación requiere semánticas de comparación más complejas, como es el caso de las cadenas que tienen en cuenta la referencia cultural.

El siguiente ejemplo muestra el uso de `List.sort`.

```
let sortedList1 = List.sort [1; 4; 8; -2; 5]
printfn "%A" sortedList1
```

La salida es la siguiente:

```
[-2; 1; 4; 5; 8]
```

El siguiente ejemplo muestra el uso de `List.sortBy`.

```
let sortedList2 = List.sortBy (fun elem -> abs elem) [1; 4; 8; -2; 5]
printfn "%A" sortedList2
```

La salida es la siguiente:

```
[1; -2; 4; 5; 8]
```

El siguiente ejemplo muestra el uso de `List.sortWith`. En este ejemplo, la función de comparación personalizada `compareWidgets` se usa para comparar primero un campo de un tipo personalizado, y después otro cuando los valores del primer campo son iguales.

```
type Widget = { ID: int; Rev: int }

let compareWidgets widget1 widget2 =
    if widget1.ID < widget2.ID then -1 else
    if widget1.ID > widget2.ID then 1 else
    if widget1.Rev < widget2.Rev then -1 else
    if widget1.Rev > widget2.Rev then 1 else
    0

let listToCompare = [
    { ID = 92; Rev = 1 }
    { ID = 110; Rev = 1 }
    { ID = 100; Rev = 5 }
    { ID = 100; Rev = 2 }
    { ID = 92; Rev = 1 }
]

let sortedWidgetList = List.sortWith compareWidgets listToCompare
printfn "%A" sortedWidgetList
```

La salida es la siguiente:

```
[{ID = 92;
Rev = 1;}; {ID = 92;
Rev = 1;}; {ID = 100;
Rev = 2;}; {ID = 100;
Rev = 5;}; {ID = 110;
Rev = 1;}]
```

Operaciones de búsqueda en listas

Se admiten numerosas operaciones de búsqueda en las listas. La más sencilla, [List.find](#), permite buscar el primer elemento que coincide con una condición determinada.

El ejemplo de código siguiente muestra el uso de `List.find` para buscar el primer número que es divisible por 5 en una lista.

```
let isDivisibleBy number elem = elem % number = 0
let result = List.find (isDivisibleBy 5) [ 1 .. 100 ]
printfn "%d " result
```

El resultado es 5.

Si los elementos se deben transformar primero, llame a [List.Pick](#), que toma una función que devuelve una opción y busca el primer valor de opción que sea `Some(x)`. En lugar de devolver el elemento, `List.pick` devuelve el resultado `x`. Si no se encuentra ningún elemento coincidente, `List.pick` activa `System.Collections.Generic.KeyNotFoundException`. En el código siguiente se muestra el uso de `List.pick`.

```
let valuesList = [ ("a", 1); ("b", 2); ("c", 3) ]

let resultPick = List.pick (fun elem ->
    match elem with
    | (value, 2) -> Some value
    | _ -> None) valuesList
printfn "%A" resultPick
```

La salida es la siguiente:

```
"b"
```

Otro grupo de operaciones de búsqueda, [List.tryFind](#) y funciones relacionadas, devuelven un valor de opción. La función `List.tryFind` devuelve el primer elemento de una lista que cumple una condición si ese elemento existe, pero el valor de opción `None` si no. La variación [List.tryFindIndex](#) devuelve el índice del elemento, si se encuentra uno, en lugar del propio elemento. Estas funciones quedan reflejadas en el código siguiente.

```
let list1d = [1; 3; 7; 9; 11; 13; 15; 19; 22; 29; 36]
let isEven x = x % 2 = 0
match List.tryFind isEven list1d with
| Some value -> printfn "The first even value is %d." value
| None -> printfn "There is no even value in the list."

match List.tryFindIndex isEven list1d with
| Some value -> printfn "The first even value is at position %d." value
| None -> printfn "There is no even value in the list."
```

La salida es la siguiente:

```
The first even value is 22.
The first even value is at position 8.
```

Operaciones aritméticas en listas

Las operaciones aritméticas comunes, como SUM y Average, se integran en el [módulo List](#). Para trabajar con [List.SUM](#), el tipo de elemento de lista debe `+` admitir el operador y tener un valor de cero. Todos los tipos aritmético integrados cumplen estas condiciones. Para trabajar con [List.Average](#), el tipo de elemento debe admitir la división sin resto, lo que excluye los tipos enteros, pero permite tipos de punto flotante. Las

funciones `List.sumBy` (y `List.averageBy` toman una función como parámetro, y los resultados de esta función se usan para calcular los valores de la suma o el promedio.

En el código siguiente se muestra cómo usar `List.sum`, `List.sumBy` y `List.average`.

```
// Compute the sum of the first 10 integers by using List.sum.
let sum1 = List.sum [1 .. 10]

// Compute the sum of the squares of the elements of a list by using List.sumBy.
let sum2 = List.sumBy (fun elem -> elem*elem) [1 .. 10]

// Compute the average of the elements of a list by using List.average.
let avg1 = List.average [0.0; 1.0; 1.0; 2.0]

printfn "%f" avg1
```

El resultado es `1.000000`

En el código siguiente se muestra el uso de `List.averageBy`.

```
let avg2 = List.averageBy (fun elem -> float elem) [1 .. 10]
printfn "%f" avg2
```

El resultado es `5.5`

Listas y tuplas

Las listas que contienen tuplas se pueden manipular con las funciones `zip` y `unzip`. Estas funciones combinan dos listas de valores únicos en una lista de tuplas o separan una lista de tuplas en dos listas de valores únicos. La función `List.zip` más sencilla toma dos listas de elementos únicos y genera una sola lista de pares de tupla. Otra versión, `List.zip3` (, toma tres listas de elementos únicos y genera una única lista de tuplas que tienen tres elementos. En el siguiente ejemplo de código se muestra el uso de `List.zip`.

```
let list1 = [ 1; 2; 3 ]
let list2 = [ -1; -2; -3 ]
let listZip = List.zip list1 list2
printfn "%A" listZip
```

La salida es la siguiente:

```
[(1, -1); (2, -2); (3, -3)]
```

En el siguiente ejemplo de código se muestra el uso de `List.zip3`.

```
let list3 = [ 0; 0; 0 ]
let listZip3 = List.zip3 list1 list2 list3
printfn "%A" listZip3
```

La salida es la siguiente:

```
[(1, -1, 0); (2, -2, 0); (3, -3, 0)]
```

Las versiones de `unzip` correspondientes, `List.unzip` y `List.unzip3` (, toman listas de tuplas y devuelven listas en una tupla, donde la primera lista contiene todos los elementos que estaban en primer lugar en cada tupla y la segunda lista contiene el segundo elemento de cada uno. tupla, etc.

En el ejemplo de código siguiente se muestra el uso de [List.unzip](#).

```
let lists = List.unzip [(1,2); (3,4)]
printfn "%A" lists
printfn "%A %A" (fst lists) (snd lists)
```

La salida es la siguiente:

```
([1; 3], [2; 4])
[1; 3] [2; 4]
```

En el ejemplo de código siguiente se muestra el uso de [List.unzip3](#) (.

```
let listsUnzip3 = List.unzip3 [(1,2,3); (4,5,6)]
printfn "%A" listsUnzip3
```

La salida es la siguiente:

```
([1; 4], [2; 5], [3; 6])
```

Operar en elementos de lista

F# admite diferentes operaciones en los elementos de lista. La más sencilla es [List.ITER](#), que le permite llamar a una función en cada elemento de una lista. Las variaciones incluyen [List.iter2](#) (, que permite realizar una operación en elementos de dos listas, [List.ITERI](#), que es como `List.iter`, a excepción de que el índice de cada elemento se pasa como argumento a la función a la que se llama para cada uno. Element y [List.iteri2](#) (, que es una combinación de la funcionalidad de `List.iter2` y `List.iteri`. En el siguiente ejemplo de código se muestran estas funciones.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
List.iter (fun x -> printfn "List.iter: element is %d" x) list1
List.iteri (fun i x -> printfn "List.iteri: element %d is %d" i x) list1
List.iter2 (fun x y -> printfn "List.iter2: elements are %d %d" x y) list1 list2
List.iteri2 (fun i x y ->
    printfn "List.iteri2: element %d of list1 is %d element %d of list2 is %d"
        i x i y)
    list1 list2
```

La salida es la siguiente:

```
List.iter: element is 1
List.iter: element is 2
List.iter: element is 3
List.iteri: element 0 is 1
List.iteri: element 1 is 2
List.iteri: element 2 is 3
List.iter2: elements are 1 4
List.iter2: elements are 2 5
List.iter2: elements are 3 6
List.iteri2: element 0 of list1 is 1; element 0 of list2 is 4
List.iteri2: element 1 of list1 is 2; element 1 of list2 is 5
List.iteri2: element 2 of list1 is 3; element 2 of list2 is 6
```

Otra función que se usa con frecuencia que transforma elementos de lista es [List.map](#), que permite aplicar una función a cada elemento de una lista y colocar todos los resultados en una nueva lista. [List.MAP2](#) (y [List](#).

`map3` (son variaciones que toman varias listas. También puede usar `List.MAPI` y `List.mapi2` (si, además del elemento, la función debe pasar el índice de cada elemento. La única diferencia entre `List.mapi2` y `List.mapi` es que `List.mapi2` trabaja con dos listas. En el ejemplo siguiente se muestra `List.map`.

```
let list1 = [1; 2; 3]
let newList = List.map (fun x -> x + 1) list1
printfn "%A" newList
```

La salida es la siguiente:

```
[2; 3; 4]
```

En el ejemplo siguiente se muestra el uso de `List.map2`.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
let sumList = List.map2 (fun x y -> x + y) list1 list2
printfn "%A" sumList
```

La salida es la siguiente:

```
[5; 7; 9]
```

En el ejemplo siguiente se muestra el uso de `List.map3`.

```
let newList2 = List.map3 (fun x y z -> x + y + z) list1 list2 [2; 3; 4]
printfn "%A" newList2
```

La salida es la siguiente:

```
[7; 10; 13]
```

En el ejemplo siguiente se muestra el uso de `List.mapi`.

```
let newListAddIndex = List.mapi (fun i x -> x + i) list1
printfn "%A" newListAddIndex
```

La salida es la siguiente:

```
[1; 3; 5]
```

En el ejemplo siguiente se muestra el uso de `List.mapi2`.

```
let listAddTimesIndex = List.mapi2 (fun i x y -> (x + y) * i) list1 list2
printfn "%A" listAddTimesIndex
```

La salida es la siguiente:

```
[0; 7; 18]
```

[List. Collect](#) es como `List.map`, salvo que cada elemento genera una lista y todas estas listas se concatenan en una lista final. En el código siguiente, cada elemento de la lista genera tres números. Todos ellos se recopilan en una lista.

```
let collectList = List.collect (fun x -> [for i in 1..3 -> x * i]) list1
printfn "%A" collectList
```

La salida es la siguiente:

```
[1; 2; 3; 2; 4; 6; 3; 6; 9]
```

También puede usar [List. Filter](#), que toma una condición booleana y genera una nueva lista que solo consta de los elementos que satisfacen la condición especificada.

```
let evenOnlyList = List.filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6]
```

La lista resultante es `[2; 4; 6]`.

Una combinación de asignación y filtro, [List. Choose](#) permite transformar y seleccionar elementos al mismo tiempo. `List.choose` aplica una función que devuelve una opción a cada elemento de una lista, y devuelve una nueva lista de resultados de elementos cuando la función devuelve el valor de opción `Some`.

El código siguiente muestra cómo usar `List.choose` para seleccionar las palabras en mayúsculas de una lista de palabras.

```
let listWords = [ "and"; "Rome"; "Bob"; "apple"; "zebra" ]
let isCapitalized (string1:string) = System.Char.IsUpper string1.[0]
let results = List.choose (fun elem ->
    match elem with
    | elem when isCapitalized elem -> Some(elem + "'s")
    | _ -> None) listWords
printfn "%A" results
```

La salida es la siguiente:

```
["Rome's"; "Bob's"]
```

Operar en varias listas

Las listas se pueden unir entre sí. Para combinar dos listas en una, use [List. Append](#). Para combinar más de dos listas, use [List. concat](#).

```
let list1to10 = List.append [1; 2; 3] [4; 5; 6; 7; 8; 9; 10]
let listResult = List.concat [ [1; 2; 3]; [4; 5; 6]; [7; 8; 9] ]
List.iter (fun elem -> printf "%d " elem) list1to10
printfn ""
List.iter (fun elem -> printf "%d " elem) listResult
```

Operaciones de plegamiento y exploración

Algunas operaciones de lista implican interdependencias entre todos los elementos de lista. Las operaciones de plegamiento y de recorrido `List.iter` son `List.map` como y, en el caso de que se invoque una función en cada elemento, pero estas operaciones proporcionan un parámetro adicional denominado el *acumulador* que lleva información a través del cálculo.

Use `List.fold` para realizar un cálculo en una lista.

En el ejemplo de código siguiente se muestra el uso de `List.Fold` para realizar varias operaciones.

La lista se atraviesa; el acumulador `acc` es un valor que va pasando mientras se realiza el cálculo. El primer argumento toma el acumulador y el elemento de lista y devuelve el resultado provisional del cálculo para ese elemento de lista. El segundo argumento es el valor inicial del acumulador.

```
let sumList list = List.fold (fun acc elem -> acc + elem) 0 list
printfn "Sum of the elements of list %A is %d." [ 1 .. 3 ] (sumList [ 1 .. 3 ])

// The following example computes the average of a list.
let averageList list = (List.fold (fun acc elem -> acc + float elem) 0.0 list / float list.Length)

// The following example computes the standard deviation of a list.
// The standard deviation is computed by taking the square root of the
// sum of the variances, which are the differences between each value
// and the average.
let stdDevList list =
    let avg = averageList list
    sqrt (List.fold (fun acc elem -> acc + (float elem - avg) ** 2.0 ) 0.0 list / float list.Length)

let testList listTest =
    printfn "List %A average: %f stddev: %f" listTest (averageList listTest) (stdDevList listTest)

testList [1; 1; 1]
testList [1; 2; 1]
testList [1; 2; 3]

// List.fold is the same as to List.iter when the accumulator is not used.
let printList list = List.fold (fun acc elem -> printfn "%A" elem) () list
printList [0.0; 1.0; 2.5; 5.1 ]

// The following example uses List.fold to reverse a list.
// The accumulator starts out as the empty list, and the function uses the cons operator
// to add each successive element to the head of the accumulator list, resulting in a
// reversed form of the list.
let reverseList list = List.fold (fun acc elem -> elem::acc) [] list
printfn "%A" (reverseList [1 .. 10])
```

Las versiones de estas funciones que tienen un dígito en el nombre de la función operan en más de una lista. Por ejemplo, `List.fold2` realiza cálculos en dos listas.

El siguiente ejemplo muestra el uso de `List.fold2`.

```
// Use List.fold2 to perform computations over two lists (of equal size) at the same time.
// Example: Sum the greater element at each list position.
let sumGreatest list1 list2 = List.fold2 (fun acc elem1 elem2 ->
    acc + max elem1 elem2) 0 list1 list2

let sum = sumGreatest [1; 2; 3] [3; 2; 1]
printfn "The sum of the greater of each pair of elements in the two lists is %d." sum
```

`List.fold` y `List.Scan` difiere en `List.fold` que devuelve el valor final del parámetro adicional, pero `List.scan` devuelve la lista de los valores intermedios (junto con el valor final) del parámetro adicional.

Cada una de estas funciones incluye una variación inversa, por ejemplo, `List.foldBack`, que difiere en el orden en el que se recorre la lista y el orden de los argumentos. Además, `List.fold` y `List.foldBack` tienen variaciones, `List.fold2` (y `List.foldBack2` , que toman dos listas de la misma longitud. La función que se ejecuta en cada elemento puede usar los elementos correspondientes de ambas listas para realizar alguna acción. Los tipos de elemento de las dos listas pueden ser diferentes, como en el ejemplo siguiente, en el que

una lista contiene los importes de las transacciones de una cuenta bancaria y la otra lista contiene el tipo de transacción: ingreso o retirada.

```
// Discriminated union type that encodes the transaction type.
type Transaction =
  | Deposit
  | Withdrawal

let transactionTypes = [Deposit; Deposit; Withdrawal]
let transactionAmounts = [100.00; 1000.00; 95.00 ]
let initialBalance = 200.00

// Use fold2 to perform a calculation on the list to update the account balance.
let endingBalance = List.fold2 (fun acc elem1 elem2 ->
    match elem1 with
    | Deposit -> acc + elem2
    | Withdrawal -> acc - elem2)
    initialBalance
    transactionTypes
    transactionAmounts

printfn "%f" endingBalance
```

Para un cálculo como la suma, `List.fold` y `List.foldBack` tiene el mismo efecto porque el resultado no depende del orden del recorrido. En el ejemplo siguientes, se usa `List.foldBack` para sumar los elementos de una lista.

```
let sumListBack list = List.foldBack (fun acc elem -> acc + elem) list 0
printfn "%d" (sumListBack [1; 2; 3])

// For a calculation in which the order of traversal is important, fold and foldBack have different
// results. For example, replacing fold with foldBack in the listReverse function
// produces a function that copies the list, rather than reversing it.
let copyList list = List.foldBack (fun elem acc -> elem::acc) list []
printfn "%A" (copyList [1 .. 10])
```

El ejemplo siguiente vuelve al ejemplo de la cuenta bancaria. Esta vez se agrega un nuevo tipo de transacción: un cálculo de interés. El saldo final depende ahora del orden de las transacciones.

```

type Transaction2 =
    | Deposit
    | Withdrawal
    | Interest

let transactionTypes2 = [Deposit; Deposit; Withdrawal; Interest]
let transactionAmounts2 = [100.00; 1000.00; 95.00; 0.05 / 12.0 ]
let initialBalance2 = 200.00

// Because fold2 processes the lists by starting at the head element,
// the interest is calculated last, on the balance of 1205.00.
let endingBalance2 = List.fold2 (fun acc elem1 elem2 ->
    match elem1 with
    | Deposit -> acc + elem2
    | Withdrawal -> acc - elem2
    | Interest -> acc * (1.0 + elem2))
    initialBalance2
    transactionTypes2
    transactionAmounts2

printfn "%f" endingBalance2

// Because foldBack2 processes the lists by starting at end of the list,
// the interest is calculated first, on the balance of only 200.00.
let endingBalance3 = List.foldBack2 (fun elem1 elem2 acc ->
    match elem1 with
    | Deposit -> acc + elem2
    | Withdrawal -> acc - elem2
    | Interest -> acc * (1.0 + elem2))
    transactionTypes2
    transactionAmounts2
    initialBalance2

printfn "%f" endingBalance3

```

La lista de funciones `.reduce` es algo `List.fold` parecido `List.scan` a y, salvo que, en lugar de pasar un acumulador independiente, `List.reduce` toma una función que toma dos argumentos del tipo de elemento en lugar de uno solo y uno de ellos. los argumentos actúan como el acumulador, lo que significa que almacena el resultado intermedio del cálculo. `List.reduce` comienza por operar en los dos primeros elementos y, después, usa el resultado de la operación con el siguiente elemento. Como no hay un acumulador diferente que tenga su propio tipo, se puede usar `List.reduce` en lugar de `List.fold` solo cuando el acumulador y el elemento sean del mismo tipo. En el código siguiente se muestra cómo usar `List.reduce`. `List.reduce` activa una excepción si la lista proporciona no tiene elementos.

En el código siguiente, en la primera llamada a la expresión lambda se proporcionan los argumentos 2 y 4, y devuelve 6; en la siguiente llamada se proporcionan los argumentos 6 y 10, por lo que el resultado es 16.

```

let sumAList list =
    try
        List.reduce (fun acc elem -> acc + elem) list
    with
        | :? System.ArgumentException as exc -> 0

let resultSum = sumAList [2; 4; 10]
printfn "%d " resultSum

```

Convertir entre listas y otros tipos de colecciones

El módulo `List` proporciona funciones para convertir desde y hacia secuencias y matrices. Para convertir a o desde una secuencia, use `List.toSeq` (o `List.ofSeq`). Para realizar la conversión a o desde una matriz, use `List.toArray` o `List.myArray`.

Otras operaciones

Para obtener información sobre las operaciones adicionales de las listas, vea el módulo de referencia de la biblioteca [Collections. List](#).

Vea también

- [Referencia del lenguaje F#](#)
- [Tipos en F#](#)
- [Secuencias](#)
- [Matrices](#)
- [Opciones](#)

Matrices

23/10/2019 • 29 minutes to read • [Edit Online](#)

NOTE

El vínculo de la referencia de API le llevará a MSDN. La referencia de API de docs.microsoft.com no está completa.

Las matrices son colecciones mutables de tamaño fijo y de base cero de elementos de datos consecutivos que son del mismo tipo.

Crear matrices

Puede crear matrices de varias maneras. Puede crear una matriz pequeña enumerando valores consecutivos entre `[]` y `[]` y separados por punto y coma, tal como se muestra en los ejemplos siguientes.

```
let array1 = [ | 1; 2; 3 | ]
```

También puede colocar cada elemento en una línea independiente, en cuyo caso el separador de punto y coma es opcional.

```
let array1 =  
  [  
    1  
    2  
    3  
  ]
```

El tipo de los elementos de la matriz se deduce de los literales usados y debe ser coherente. El código siguiente produce un error porque 1,0 es float y 2 y 3 son enteros.

```
// Causes an error.  
// let array2 = [ | 1.0; 2; 3 | ]
```

También puede utilizar expresiones de secuencia para crear matrices. A continuación se encuentra un ejemplo que crea una matriz de cuadrados de enteros de 1 a 10.

```
let array3 = [ | for i in 1 .. 10 -> i * i | ]
```

Para crear una matriz en la que todos los elementos se inicializan en cero, use `Array.zeroCreate`.

```
let arrayOfTenZeroes : int array = Array.zeroCreate 10
```

Obtener acceso a elementos

Puede tener acceso a los elementos de la matriz mediante un operador punto (`.`) y corchetes (`[]` y `[]`).

```
array1.[0]
```

Los índices de matriz comienzan en 0.

También puede tener acceso a los elementos de la matriz mediante la notación de segmentación, que permite especificar un subintervalo de la matriz. A continuación se muestran ejemplos de notación de segmentación.

```
// Accesses elements from 0 to 2.

array1.[0..2]

// Accesses elements from the beginning of the array to 2.

array1[..2]

// Accesses elements from 2 to the end of the array.

array1.[2..]
```

Cuando se usa la notación de segmentos, se crea una nueva copia de la matriz.

Tipos de matriz y módulos

El tipo de todas las matrices en F# es el tipo de .NET Framework [System.Array](#). Por lo tanto, las matrices admiten toda la funcionalidad disponible en [System.Array](#).

El módulo de biblioteca [Microsoft.FSharp.Collections.Array](#) admite operaciones en matrices unidimensionales. Los módulos [Array2D](#), [Array3D](#) y [Array4D](#) contienen funciones que admiten operaciones en matrices de dos, tres y cuatro dimensiones, respectivamente. Puede crear matrices de rango mayores que cuatro mediante [System.Array](#).

Funciones simples

[Array.get](#) obtiene un elemento. [Array.length](#) proporciona la longitud de una matriz. [Array.set](#) establece un elemento en un valor especificado. En el ejemplo de código siguiente se muestra el uso de estas funciones.

```
let array1 = Array.create 10 ""
for i in 0 .. array1.Length - 1 do
    Array.set array1 i (i.ToString())
for i in 0 .. array1.Length - 1 do
    printf "%s " (Array.get array1 i)
```

La salida es la siguiente.

```
0 1 2 3 4 5 6 7 8 9
```

Funciones que crean matrices

Varias funciones crean matrices sin necesidad de una matriz existente. [Array.empty](#) crea una nueva matriz que no contiene ningún elemento. [Array.create](#) crea una matriz de un tamaño especificado y establece todos los elementos en los valores proporcionados. [Array.init](#) crea una matriz, dada una dimensión y una función para generar los elementos. [Array.zeroCreate](#) crea una matriz en la que todos los elementos se inicializan en el valor cero para el tipo de la matriz. En el código siguiente se muestran estas funciones.

```

let myEmptyArray = Array.empty
println "Length of empty array: %d" myEmptyArray.Length

println "Array of floats set to 5.0: %A" (Array.create 10 5.0)

println "Array of squares: %A" (Array.init 10 (fun index -> index * index))

let (myZeroArray : float array) = Array.zeroCreate 10

```

La salida es la siguiente.

```

Length of empty array: 0
Area of floats set to 5.0: [|5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0|]
Array of squares: [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]

```

`Array.copy` crea una nueva matriz que contiene los elementos que se copian de una matriz existente. Tenga en cuenta que la copia es una copia superficial, lo que significa que si el tipo de elemento es un tipo de referencia, solo se copia la referencia, no el objeto subyacente. En el siguiente ejemplo código se muestra cómo hacerlo.

```

open System.Text

let firstArray : StringBuilder array = Array.init 3 (fun index -> new StringBuilder(""))
let secondArray = Array.copy firstArray
// Reset an element of the first array to a new value.
firstArray.[0] <- new StringBuilder("Test1")
// Change an element of the first array.
firstArray.[1].Insert(0, "Test2") |> ignore
println "%A" firstArray
println "%A" secondArray

```

La salida del código anterior es la siguiente:

```

[|Test1; Test2; |]
[|; Test2; |]

```

La cadena `Test1` solo aparece en la primera matriz porque la operación de creación de un nuevo elemento sobrescribe la referencia en `firstArray` pero no afecta a la referencia original a una cadena vacía que todavía está presente en `secondArray`. La cadena `Test2` aparece en ambas matrices porque la operación `Insert` en el tipo `System.Text.StringBuilder` afecta al objeto `System.Text.StringBuilder` subyacente, al que se hace referencia en ambas matrices.

`Array.sub` genera una nueva matriz a partir de un subintervalo de una matriz. Para especificar el subintervalo, proporcione el índice de inicio y la longitud. En el código siguiente se muestra cómo usar `Array.sub`.

```

let a1 = [| 0 .. 99 |]
let a2 = Array.sub a1 5 10
println "%A" a2

```

La salida muestra que la submatriz comienza en el elemento 5 y contiene 10 elementos.

```

[|5; 6; 7; 8; 9; 10; 11; 12; 13; 14|]

```

`Array.append` crea una nueva matriz combinando dos matrices existentes.

En el código siguiente se muestra **array.append**.

```
println "%A" (Array.append [| 1; 2; 3|] [| 4; 5; 6|])
```

La salida del código anterior es la siguiente.

```
[|1; 2; 3; 4; 5; 6|]
```

`Array.choose` selecciona los elementos de una matriz para incluirlos en una nueva matriz. En el código siguiente se muestra `Array.choose`. Tenga en cuenta que el tipo de elemento de la matriz no tiene que coincidir con el tipo del valor devuelto en el tipo de opción. En este ejemplo, el tipo de elemento es `int` y la opción es el resultado de una función polinómica, `elem*elem - 1`, como un número de punto flotante.

```
println "%A" (Array.choose (fun elem -> if elem % 2 = 0 then
                                     Some(float (elem*elem - 1))
                                   else
                                   None) [| 1 .. 10 |])
```

La salida del código anterior es la siguiente.

```
[|3.0; 15.0; 35.0; 63.0; 99.0|]
```

`Array.collect` ejecuta una función especificada en cada elemento de matriz de una matriz existente y, a continuación, recopila los elementos generados por la función y los combina en una nueva matriz. En el código siguiente se muestra `Array.collect`.

```
println "%A" (Array.collect (fun elem -> [| 0 .. elem |]) [| 1; 5; 10|])
```

La salida del código anterior es la siguiente.

```
[|0; 1; 0; 1; 2; 3; 4; 5; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

`Array.concat` toma una secuencia de matrices y las combina en una sola matriz. En el código siguiente se muestra `Array.concat`.

```
Array.concat [| [|0..3|] ; [|4|] |]
//output [|0; 1; 2; 3; 4|]

Array.concat [| [|0..3|] ; [|4|] |]
//output [|0; 1; 2; 3; 4|]
```

La salida del código anterior es la siguiente.

```
[|(1, 1, 1); (1, 2, 2); (1, 3, 3); (2, 1, 2); (2, 2, 4); (2, 3, 6); (3, 1, 3);
(3, 2, 6); (3, 3, 9)|]
```

`Array.filter` toma una función de condición booleana y genera una nueva matriz que contiene solo los elementos de la matriz de entrada para los que la condición es verdadera. En el código siguiente se muestra `Array.filter`.

```
printfn "%A" (Array.filter (fun elem -> elem % 2 = 0) [| 1 .. 10|])
```

La salida del código anterior es la siguiente.

```
[|2; 4; 6; 8; 10|]
```

`Array.rev` genera una nueva matriz invirtiendo el orden de una matriz existente. En el código siguiente se muestra `Array.rev`.

```
let stringReverse (s: string) =  
    System.String(Array.rev (s.ToCharArray()))  
  
printfn "%A" (stringReverse("!dlrow olleH"))
```

La salida del código anterior es la siguiente.

```
"Hello world!"
```

Puede combinar fácilmente funciones en el módulo de matriz que transforman matrices mediante el operador de canalización (`|>`), como se muestra en el ejemplo siguiente.

```
[| 1 .. 10 |]  
|> Array.filter (fun elem -> elem % 2 = 0)  
|> Array.choose (fun elem -> if (elem <> 8) then Some(elem*elem) else None)  
|> Array.rev  
|> printfn "%A"
```

El resultado es

```
[|100; 36; 16; 4|]
```

Matrices multidimensionales

Se puede crear una matriz multidimensional, pero no hay ninguna sintaxis para escribir un literal de matriz multidimensional. Use el operador `array2D` para crear una matriz a partir de una secuencia de secuencias de elementos de matriz. Las secuencias pueden ser literales de matriz o lista. Por ejemplo, el código siguiente crea una matriz bidimensional.

```
let my2DArray = array2D [ [ 1; 0 ]; [0; 1] ]
```

También puede usar la función `Array2D.init` para inicializar matrices de dos dimensiones, y las funciones similares están disponibles para matrices de tres y cuatro dimensiones. Estas funciones toman una función que se utiliza para crear los elementos. Para crear una matriz bidimensional que contenga elementos establecidos en un valor inicial en lugar de especificar una función, use la función `Array2D.create`, que también está disponible para matrices de hasta cuatro dimensiones. En el ejemplo de código siguiente se muestra primero cómo crear una matriz de matrices que contienen los elementos deseados y, a continuación, se usa `Array2D.init` para generar la matriz bidimensional deseada.

```
let arrayOfArrays = [ [| 1.0; 0.0 |]; [|0.0; 1.0 |] ]  
let twoDimensionalArray = Array2D.init 2 2 (fun i j -> arrayOfArrays.[i].[j])
```


La sintaxis de indización y segmentación de matrices es compatible con matrices hasta el rango 4. Cuando se especifica un índice en varias dimensiones, se usan comas para separar los índices, tal y como se muestra en el ejemplo de código siguiente.

```
twoDimensionalArray.[0, 1] <- 1.0
```

El tipo de una matriz bidimensional se escribe como `<type>[,]` (por ejemplo, `int[,]`, `double[,]`) y el tipo de una matriz tridimensional se escribe como `<type>[, ,]`, y así sucesivamente para las matrices de dimensiones superiores.

Solo hay disponible un subconjunto de las funciones disponibles para las matrices unidimensionales para las matrices multidimensionales. Para obtener más información, vea [Collections.Array Module](#), [Collections.Array2D Module](#), [Collections.Array3D Module](#) y [Collections.Array4D Module](#).

Segmentación de matrices y matrices multidimensionales

En una matriz bidimensional (matriz), puede extraer una submatriz especificando los intervalos y usando un carácter comodín (`*`) para especificar las filas o columnas completas.

```
// Get rows 1 to N from an NxM matrix (returns a matrix):
matrix.[1.., *]

// Get rows 1 to 3 from a matrix (returns a matrix):
matrix.[1..3, *]

// Get columns 1 to 3 from a matrix (returns a matrix):
matrix.[*, 1..3]

// Get a 3x3 submatrix:
matrix.[1..3, 1..3]
```

A partir F# de 3,1, puede descomponer una matriz multidimensional en submatrices de la misma dimensión o inferior. Por ejemplo, puede obtener un vector de una matriz especificando una sola fila o columna.

```
// Get row 3 from a matrix as a vector:
matrix.[3, *]

// Get column 3 from a matrix as a vector:
matrix.[*, 3]
```

Puede usar esta sintaxis de segmentación para los tipos que implementan los operadores de acceso a elementos y los métodos `GetSlice` sobrecargados. Por ejemplo, el código siguiente crea un tipo de matriz que contiene la F# matriz 2D, implementa una propiedad `Item` para proporcionar compatibilidad con la indización de matrices e implementa tres versiones de `GetSlice`. Si puede usar este código como plantilla para los tipos de matriz, puede usar todas las operaciones de segmentación que se describen en esta sección.

```

type Matrix<'T>(N: int, M: int) =
    let internalArray = Array2D.zeroCreate<'T> N M

    member this.Item
        with get(a: int, b: int) = internalArray.[a, b]
        and set(a: int, b: int) (value:'T) = internalArray.[a, b] <- value

    member this.GetSlice(rowStart: int option, rowFinish : int option, colStart: int option, colFinish : int option) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray.[rowStart..rowFinish, colStart..colFinish]

    member this.GetSlice(row: int, colStart: int option, colFinish: int option) =
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray.[row, colStart..colFinish]

    member this.GetSlice(rowStart: int option, rowFinish: int option, col: int) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        internalArray.[rowStart..rowFinish, col]

module test =
    let generateTestMatrix x y =
        let matrix = new Matrix<float>(3, 3)
        for i in 0..2 do
            for j in 0..2 do
                matrix.[i, j] <- float(i) * x - float(j) * y
        matrix

    let test1 = generateTestMatrix 2.3 1.1
    let submatrix = test1.[0..1, 0..1]
    printfn "%A" submatrix

    let firstRow = test1.[0,*]
    let secondRow = test1.[1,*]
    let firstCol = test1.[*,0]
    printfn "%A" firstCol

```

Funciones booleanas en matrices

Las funciones `Array.exists` y `Array.exists2` elementos de prueba en una o dos matrices, respectivamente. Estas funciones toman una función de prueba y devuelven `true` si hay un elemento (o par de elementos para `Array.exists2`) que cumple la condición.

En el código siguiente se muestra el uso de `Array.exists` y `Array.exists2`. En estos ejemplos, se crean nuevas funciones aplicando solo uno de los argumentos, en estos casos, el argumento de la función.

```
let allNegative = Array.exists (fun elem -> abs (elem) = elem) >> not
printfn "%A" (allNegative [| -1; -2; -3 |])
printfn "%A" (allNegative [| -10; -1; 5 |])
printfn "%A" (allNegative [| 0 |])

let haveEqualElement = Array.exists2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (haveEqualElement [| 1; 2; 3 |] [| 3; 2; 1 |])
```

La salida del código anterior es la siguiente.

```
true
false
false
true
```

Del mismo modo, la función `Array.forall` prueba una matriz para determinar si todos los elementos satisfacen una condición booleana. La variación `Array.forall2` hace lo mismo mediante el uso de una función booleana que implica elementos de dos matrices de igual longitud. En el código siguiente se muestra el uso de estas funciones.

```
let allPositive = Array.forall (fun elem -> elem > 0)
printfn "%A" (allPositive [| 0; 1; 2; 3 |])
printfn "%A" (allPositive [| 1; 2; 3 |])

let allEqual = Array.forall2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (allEqual [| 1; 2 |] [| 1; 2 |])
printfn "%A" (allEqual [| 1; 2 |] [| 2; 1 |])
```

La salida de estos ejemplos es la siguiente.

```
false
true
true
false
```

Buscar matrices

`Array.find` toma una función booleana y devuelve el primer elemento para el que la función devuelve `true`, o bien genera un `System.Collections.Generic.KeyNotFoundException` si no se encuentra ningún elemento que cumpla la condición. `Array.findIndex` es como `Array.find`, con la diferencia de que devuelve el índice del elemento en lugar del propio elemento.

En el código siguiente se usa `Array.find` y `Array.findIndex` para buscar un número que sea tanto un cuadrado perfecto como un cubo perfecto.

```

let arrayA = [| 2 .. 100 |]
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let element = Array.find (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
let index = Array.findIndex (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
printfn "The first element that is both a square and a cube is %d and its index is %d." element index

```

La salida es la siguiente.

```
The first element that is both a square and a cube is 64 and its index is 62.
```

`Array.tryFind` es como `Array.find`, excepto que su resultado es un tipo de opción y devuelve `None` si no se encuentra ningún elemento. se debe utilizar `Array.tryFind` en lugar de `Array.find` cuando no se sabe si un elemento coincidente está en la matriz. Del mismo modo, `Array.tryFindIndex` es como `Array.findIndex`, salvo que el tipo de opción es el valor devuelto. Si no se encuentra ningún elemento, la opción es `None`.

En el código siguiente se muestra cómo usar `Array.tryFind`. Este código depende del código anterior.

```

let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let lookForCubeAndSquare array1 =
    let result = Array.tryFind (fun elem -> isPerfectSquare elem && isPerfectCube elem) array1
    match result with
    | Some x -> printfn "Found an element: %d" x
    | None -> printfn "Failed to find a matching element."

lookForCubeAndSquare [| 1 .. 10 |]
lookForCubeAndSquare [| 100 .. 1000 |]
lookForCubeAndSquare [| 2 .. 50 |]

```

La salida es la siguiente.

```

Found an element: 1
Found an element: 729
Failed to find a matching element.

```

Use `Array.tryPick` cuando necesite transformar un elemento además de buscarlo. El resultado es el primer elemento para el que la función devuelve el elemento transformado como un valor de opción o `None` si no se encuentra dicho elemento.

En el código siguiente se muestra el uso de `Array.tryPick`. En este caso, en lugar de una expresión lambda, se definen varias funciones auxiliares locales para simplificar el código.

```

let findPerfectSquareAndCube array1 =
    let delta = 1.0e-10
    let isPerfectSquare (x:int) =
        let y = sqrt (float x)
        abs(y - round y) < delta
    let isPerfectCube (x:int) =
        let y = System.Math.Pow(float x, 1.0/3.0)
        abs(y - round y) < delta
    // intFunction : (float -> float) -> int -> int
    // Allows the use of a floating point function with integers.
    let intFunction function1 number = int (round (function1 (float number)))
    let cubeRoot x = System.Math.Pow(x, 1.0/3.0)
    // testElement: int -> (int * int * int) option
    // Test an element to see whether it is a perfect square and a perfect
    // cube, and, if so, return the element, square root, and cube root
    // as an option value. Otherwise, return None.
    let testElement elem =
        if isPerfectSquare elem && isPerfectCube elem then
            Some(elem, intFunction sqrt elem, intFunction cubeRoot elem)
        else None
    match Array.tryPick testElement array1 with
    | Some (n, sqrt, cuberoot) -> printfn "Found an element %d with square root %d and cube root %d." n sqrt
    cuberoot
    | None -> printfn "Did not find an element that is both a perfect square and a perfect cube."

findPerfectSquareAndCube [| 1 .. 10 |]
findPerfectSquareAndCube [| 2 .. 100 |]
findPerfectSquareAndCube [| 100 .. 1000 |]
findPerfectSquareAndCube [| 1000 .. 10000 |]
findPerfectSquareAndCube [| 2 .. 50 |]

```

La salida es la siguiente.

```

Found an element 1 with square root 1 and cube root 1.
Found an element 64 with square root 8 and cube root 4.
Found an element 729 with square root 27 and cube root 9.
Found an element 4096 with square root 64 and cube root 16.
Did not find an element that is both a perfect square and a perfect cube.

```

Realizar cálculos en matrices

La función `Array.average` devuelve el promedio de cada elemento de una matriz. Se limita a los tipos de elemento que admiten la división exacta mediante un entero, que incluye los tipos de punto flotante, pero no los tipos enteros. La función `Array.averageBy` devuelve el promedio de los resultados de llamar a una función en cada elemento. Para una matriz de tipo entero, puede usar `Array.averageBy` y hacer que la función convierta cada elemento en un tipo de punto flotante para el cálculo.

Use `Array.max` o `Array.min` para obtener el elemento máximo o mínimo, si el tipo de elemento lo admite. Del mismo modo, `Array.maxBy` y `Array.minBy` permiten que una función se ejecute en primer lugar, quizás para realizar la transformación en un tipo que admita la comparación.

`Array.sum` agrega los elementos de una matriz y `Array.sumBy` llama a una función en cada elemento y agrega los resultados juntos.

Para ejecutar una función en cada elemento de una matriz sin almacenar los valores devueltos, utilice `Array.iter`. Para una función que implique dos matrices de igual longitud, use `Array.iter2`. Si también necesita mantener una matriz de los resultados de la función, use `Array.map` o `Array.map2`, que opera en dos matrices a la vez.

Las variaciones `Array.iteri` y `Array.iteri2` permiten que el índice del elemento esté implicado en el cálculo; lo mismo ocurre para `Array.mapi` y `Array.mapi2`.

Las funciones `Array.fold`, `Array.foldBack`, `Array.reduce`, `Array.reduceBack`, `Array.scan` y `1` ejecutan algoritmos que implican todos los elementos de una matriz. Del mismo modo, las variaciones `Array.fold2` y `Array.foldBack2` realizan cálculos en dos matrices.

Estas funciones para realizar cálculos se corresponden con las funciones del mismo nombre en el [módulo de lista](#). Para obtener ejemplos de uso, vea [listas](#).

Modificar matrices

`Array.set` establece un elemento en un valor especificado. `Array.fill` establece un intervalo de elementos de una matriz en un valor especificado. El código siguiente proporciona un ejemplo de `Array.fill`.

```
let arrayFill1 = [| 1 .. 25 |]  
Array.fill arrayFill1 2 20 0  
printfn "%A" arrayFill1
```

La salida es la siguiente.

```
[|1; 2; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 23; 24; 25|]
```

Puede usar `Array.blit` para copiar una subsección de una matriz en otra matriz.

Conversión a y desde otros tipos

`Array.ofList` crea una matriz a partir de una lista. `Array.ofSeq` crea una matriz a partir de una secuencia. `Array.toList` y `Array.toSeq` se convierten en estos otros tipos de colección del tipo de matriz.

Ordenar matrices

Use `Array.sort` para ordenar una matriz utilizando la función de comparación genérica. Use `Array.sortBy` para especificar una función que genera un valor, al que se hace referencia como *clave*, para ordenar mediante la función de comparación genérica de la clave. Use `Array.sortWith` si desea proporcionar una función de comparación personalizada. `Array.sort`, `Array.sortBy` y `Array.sortWith` devuelven la matriz ordenada como una nueva matriz. Las variaciones `Array.sortInPlace`, `Array.sortInPlaceBy` y `Array.sortInPlaceWith` modifican la matriz existente en lugar de devolver una nueva.

Matrices y tuplas

Las funciones `Array.zip` y `Array.unzip` convierten matrices de pares de tupla en tuplas de matrices y viceversa. `Array.zip3` y `Array.unzip3` son similares, salvo que funcionan con tuplas de tres elementos o tuplas de tres matrices.

Cálculos paralelos en matrices

El módulo `Array.Parallel` contiene funciones para realizar cálculos paralelos en matrices. Este módulo no está disponible en las aplicaciones que tienen como destino versiones de .NET Framework anteriores a la versión 4.

Vea también

- [Referencia del lenguaje F#](#)
- [Tipos en F#](#)

Secuencias

25/11/2019 • 32 minutes to read • [Edit Online](#)

NOTE

Los vínculos de la referencia de API de este artículo le llevarán a MSDN. La referencia de API de docs.microsoft.com no está completa.

Una *secuencia* es una serie lógica de elementos de un tipo. Las secuencias son especialmente útiles cuando se tiene una colección grande ordenada de datos, pero no se espera necesariamente usar todos los elementos. Los elementos de secuencia individuales solo se calculan según sea necesario, por lo que una secuencia puede proporcionar un mejor rendimiento que una lista en situaciones en las que no se usan todos los elementos. Las secuencias se representan mediante el tipo de `seq<'T>`, que es un alias de `IEnumerable<T>`. Por lo tanto, cualquier tipo .NET que implemente `IEnumerable<T>` interfaz se puede usar como una secuencia. El [módulo SEQ](#) proporciona compatibilidad con las manipulaciones que implican secuencias.

Expresiones de secuencia

Una *expresión de secuencia* es una expresión que se evalúa como una secuencia. Las expresiones de secuencia pueden tomar varias formas. La forma más sencilla especifica un intervalo. Por ejemplo, `seq { 1 .. 5 }` crea una secuencia que contiene cinco elementos, incluidos los puntos de conexión 1 y 5. También puede especificar un incremento (o decremento) entre dos períodos dobles. Por ejemplo, el código siguiente crea la secuencia de múltiplos de 10.

```
// Sequence that has an increment.  
seq { 0 .. 10 .. 100 }
```

Las expresiones de secuencia se componen de F# expresiones que generan valores de la secuencia. También puede generar valores mediante programación:

```
seq { for i in 1 .. 10 -> i * i }
```

En el ejemplo anterior se usa el operador `->`, que permite especificar una expresión cuyo valor se convertirá en parte de la secuencia. Solo puede utilizar `->` si cada parte del código que le sigue devuelve un valor.

Como alternativa, puede especificar la palabra clave `do`, con un `yield` opcional que sigue:

```
seq { for i in 1 .. 10 do yield i * i }  
  
// The 'yield' is implicit and doesn't need to be specified in most cases.  
seq { for i in 1 .. 10 do i * i }
```

El código siguiente genera una lista de pares de coordenadas junto con un índice en una matriz que representa la cuadrícula. Tenga en cuenta que la primera expresión `for` requiere que se especifique un `do`.

```
let (height, width) = (10, 10)

seq {
  for row in 0 .. width - 1 do
    for col in 0 .. height - 1 ->
      (row, col, row*width + col)
}
```

Una expresión `if` utilizada en una secuencia es un filtro. Por ejemplo, para generar una secuencia de números primos, suponiendo que tiene una función `isprime` de tipo `int -> bool`, construya la secuencia como se indica a continuación.

```
seq { for n in 1 .. 100 do if isprime n then n }
```

Como se mencionó anteriormente, aquí se requiere `do` porque no hay ninguna rama `else` que se refiere a la `if`. Si intenta usar `->`, obtendrá un error que indica que no todas las ramas devuelven un valor.

Palabra clave `yield!`

A veces, puede que desee incluir una secuencia de elementos en otra secuencia. Para incluir una secuencia dentro de otra secuencia, debe usar la palabra clave `yield!`:

```
// Repeats '1 2 3 4 5' ten times
seq {
  for _ in 1..10 do
    yield! seq { 1; 2; 3; 4; 5 }
}
```

Otra manera de pensar en `yield!` es que reduce una secuencia interna y, a continuación, la incluye en la secuencia contenedora.

Cuando se utiliza `yield!` en una expresión, todos los demás valores únicos deben usar la palabra clave `yield`:

```
// Combine repeated values with their values
seq {
  for x in 1..10 do
    yield x
    yield! seq { for i in 1..x -> i }
}
```

Si solo se especifica `x` en el ejemplo anterior, la secuencia no genera valores.

Ejemplos

En el primer ejemplo se usa una expresión de secuencia que contiene una iteración, un filtro y un rendimiento para generar una matriz. Este código imprime una secuencia de números primos entre 1 y 100 en la consola.


```
// Recursive isprime function.
let isprime n =
    let rec check i =
        i > n/2 || (n % i <> 0 && check (i + 1))
    check 2

let aSequence =
    seq {
        for n in 1..100 do
            if isprime n then
                n
    }

for x in aSequence do
    printfn "%d" x
```

En el ejemplo siguiente se crea una tabla de multiplicación que consta de tuplas de tres elementos, cada una de las cuales consta de dos factores y el producto:

```
let multiplicationTable =
    seq {
        for i in 1..9 do
            for j in 1..9 ->
                (i, j, i*j)
    }
```

En el ejemplo siguiente se muestra el uso de `yield!` para combinar secuencias individuales en una sola secuencia final. En este caso, las secuencias de cada subárbol de un árbol binario se concatenan en una función recursiva para generar la secuencia final.

```
// Yield the values of a binary tree in a sequence.
type Tree<'a> =
    | Tree of 'a * Tree<'a> * Tree<'a>
    | Leaf of 'a

// inorder : Tree<'a> -> seq<'a>
let rec inorder tree =
    seq {
        match tree with
        | Tree(x, left, right) ->
            yield! inorder left
            yield x
            yield! inorder right
        | Leaf x -> yield x
    }

let mytree = Tree(6, Tree(2, Leaf(1), Leaf(3)), Leaf(9))
let seq1 = inorder mytree
printfn "%A" seq1
```

Usar secuencias

Las secuencias admiten muchas de las mismas funciones que las [listas](#). Las secuencias también admiten operaciones como la agrupación y el recuento mediante el uso de funciones de generación de claves. Las secuencias también admiten funciones más diversas para extraer subsecuencias.

Muchos tipos de datos, como listas, matrices, conjuntos y asignaciones, son secuencias implícitamente porque son colecciones enumerables. Una función que toma una secuencia como argumento funciona con cualquiera de los tipos de F# datos comunes, además de cualquier tipo de datos de .net que implemente

`System.Collections.Generic.IEnumerable<'T>`. Compare esto con una función que toma una lista como argumento, que solo puede tomar listas. El tipo `seq<'T>` es una abreviatura de tipo para `IEnumerable<'T>`. Esto significa que cualquier tipo que implemente el `System.Collections.Generic.IEnumerable<'T>` genérico, que incluye matrices, listas, conjuntos y asignaciones en y, F#además, la mayoría de los tipos de colección de .net, es compatible con el tipo de `seq` y se puede usar siempre que se espera una secuencia.

Funciones del módulo

El [módulo SEQ](#) del [espacio de nombres Microsoft.FSharp.Collections](#) contiene funciones para trabajar con secuencias. Estas funciones también funcionan con listas, matrices, asignaciones y conjuntos, ya que todos estos tipos son enumerables y, por tanto, se pueden tratar como secuencias.

Crear secuencias

Puede crear secuencias mediante el uso de expresiones de secuencia, como se ha descrito anteriormente, o mediante el uso de determinadas funciones.

Puede crear una secuencia vacía mediante [Seq.empty](#), o bien, puede crear una secuencia de solo un elemento especificado mediante [Seq.singleton](#).

```
let seqEmpty = Seq.empty
let seqOne = Seq.singleton 10
```

Puede usar [Seq.init](#) para crear una secuencia para la que se crean los elementos mediante el uso de una función proporcionada por el usuario. También proporciona un tamaño para la secuencia. Esta función es como [List.init](#), salvo que los elementos no se crean hasta que se recorre en iteración la secuencia. En el código siguiente, se muestra el uso de `Seq.init`.

```
let seqFirst5MultiplesOf10 = Seq.init 5 (fun n -> n * 10)
Seq.iter (fun elem -> printf "%d " elem) seqFirst5MultiplesOf10
```

El resultado es

```
0 10 20 30 40
```

Mediante el uso de la [función Seq.myArray](#) y `seq`, no `List`, puede crear secuencias a partir de matrices y listas. Sin embargo, también puede convertir matrices y listas en secuencias mediante el uso de un operador de conversión. Ambas técnicas se muestran en el código siguiente.

```
// Convert an array to a sequence by using a cast.
let seqFromArray1 = [| 1 .. 10 |] :> seq<int>

// Convert an array to a sequence by using Seq.ofArray.
let seqFromArray2 = [| 1 .. 10 |] |> Seq.ofArray
```

Con [Seq.Cast](#), puede crear una secuencia a partir de una colección débilmente tipada, como las definidas en `System.Collections`. Estas colecciones débilmente tipadas tienen el tipo de elemento `System.Object` y se enumeran mediante el tipo de `System.Collections.Generic.IEnumerable<'T>` no genérico. En el código siguiente se muestra el uso de `Seq.cast` para convertir un `System.Collections.ArrayList` en una secuencia.

```
open System

let arr = ResizeArray<int>(10)

for i in 1 .. 10 do
    arr.Add(10)

let seqCast = Seq.cast arr
```

Puede definir secuencias infinitas mediante la función [Seq.initInfinite](#) (). Para este tipo de secuencia, se proporciona una función que genera cada elemento a partir del índice del elemento. Las secuencias infinitas son posibles debido a una evaluación diferida; los elementos se crean según sea necesario mediante una llamada a la función que especifique. En el ejemplo de código siguiente se genera una secuencia infinita de números de punto flotante, en este caso, la serie alterna de recíprocos de cuadrados de enteros sucesivos.

```
let seqInfinite =
    Seq.initInfinite (fun index ->
        let n = float (index + 1)
        1.0 / (n * n * (if ((index + 1) % 2 = 0) then 1.0 else -1.0)))

printfn "%A" seqInfinite
```

[Seq.unfold](#) genera una secuencia a partir de una función de cálculo que toma un estado y la transforma para generar cada elemento subsiguiente de la secuencia. El estado es simplemente un valor que se usa para calcular cada elemento y puede cambiar a medida que se calcula cada elemento. El segundo argumento para `Seq.unfold` es el valor inicial que se utiliza para iniciar la secuencia. `Seq.unfold` usa un tipo de opción para el estado, lo que permite finalizar la secuencia devolviendo el valor de `None`. En el código siguiente se muestran dos ejemplos de secuencias, `seq1` y `fib`, que se generan mediante una operación de `unfold`. La primera, `seq1`, es simplemente una secuencia simple con números de hasta 20. El segundo, `fib`, utiliza `unfold` para calcular la secuencia de Fibonacci. Dado que cada elemento de la secuencia de Fibonacci es la suma de los dos números de Fibonacci anteriores, el valor de estado es una tupla que consta de los dos números anteriores de la secuencia. El valor inicial es `(1,1)`, los dos primeros números de la secuencia.

```
let seq1 =
    0 // Initial state
    |> Seq.unfold (fun state ->
        if (state > 20) then
            None
        else
            Some(state, state + 1))

printfn "The sequence seq1 contains numbers from 0 to 20."

for x in seq1 do
    printf "%d " x

let fib =
    (1, 1) // Initial state
    |> Seq.unfold (fun state ->
        if (snd state > 1000) then
            None
        else
            Some(fst state + snd state, (snd state, fst state + snd state)))

printfn "\nThe sequence fib contains Fibonacci numbers."
for x in fib do printf "%d " x
```

La salida es la siguiente:

The sequence seq1 contains numbers from 0 to 20.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The sequence fib contains Fibonacci numbers.

2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

El código siguiente es un ejemplo que usa muchas de las funciones de módulo de secuencia que se describen aquí para generar y calcular los valores de las secuencias infinitas. El código puede tardar unos minutos en ejecutarse.

```

// generateInfiniteSequence generates sequences of floating point
// numbers. The sequences generated are computed from the fDenominator
// function, which has the type (int -> float) and computes the
// denominator of each term in the sequence from the index of that
// term. The isAlternating parameter is true if the sequence has
// alternating signs.
let generateInfiniteSequence fDenominator isAlternating =
  if (isAlternating) then
    Seq.initInfinite (fun index ->
      1.0 /(fDenominator index) * (if (index % 2 = 0) then -1.0 else 1.0))
  else
    Seq.initInfinite (fun index -> 1.0 /(fDenominator index))

// The harmonic alternating series is like the harmonic series
// except that it has alternating signs.
let harmonicAlternatingSeries = generateInfiniteSequence (fun index -> float index) true

// This is the series of reciprocals of the odd numbers.
let oddNumberSeries = generateInfiniteSequence (fun index -> float (2 * index - 1)) true

// This is the series of reciprocals of the squares.
let squaresSeries = generateInfiniteSequence (fun index -> float (index * index)) false

// This function sums a sequence, up to the specified number of terms.
let sumSeq length sequence =
  (0, 0.0)
  |>
  Seq.unfold (fun state ->
    let subtotal = snd state + Seq.item (fst state + 1) sequence
    if (fst state >= length) then
      None
    else
      Some(subtotal, (fst state + 1, subtotal)))

// This function sums an infinite sequence up to a given value
// for the difference (epsilon) between subsequent terms,
// up to a maximum number of terms, whichever is reached first.
let infiniteSum infiniteSeq epsilon maxIteration =
  infiniteSeq
  |> sumSeq maxIteration
  |> Seq.pairwise
  |> Seq.takeWhile (fun elem -> abs (snd elem - fst elem) > epsilon)
  |> List.ofSeq
  |> List.rev
  |> List.head
  |> snd

// Compute the sums for three sequences that converge, and compare
// the sums to the expected theoretical values.
let result1 = infiniteSum harmonicAlternatingSeries 0.00001 100000
printfn "Result: %f ln2: %f" result1 (log 2.0)

let pi = Math.PI
let result2 = infiniteSum oddNumberSeries 0.00001 10000
printfn "Result: %f pi/4: %f" result2 (pi/4.0)

// Because this is not an alternating series, a much smaller epsilon
// value and more terms are needed to obtain an accurate result.
let result3 = infiniteSum squaresSeries 0.0000001 1000000
printfn "Result: %f pi*pi/6: %f" result3 (pi*pi/6.0)

```

Buscar y buscar elementos

Las secuencias admiten la funcionalidad disponible con listas: [Seq.exists](#), [Seq.exists2](#) (, [Seq.find](#), [Seq.findIndex](#), [Seq.pick](#), [Seq.tryFind](#), [Seq.tryFindIndex](#) (. Las versiones de estas funciones que están disponibles

para las secuencias evalúan la secuencia solo hasta el elemento que se está buscando. Para obtener ejemplos, vea [listas](#).

Obtener subsecuencias

[Seq. Filter](#) y [Seq. Choose](#) son como las funciones correspondientes que están disponibles para las listas, con la excepción de que el filtrado y la elección no se producen hasta que se evalúan los elementos de la secuencia.

[Seq. TRUNCATE](#) crea una secuencia a partir de otra secuencia, pero limita la secuencia a un número especificado de elementos. [Seq. Take](#) crea una nueva secuencia que solo contiene un número especificado de elementos desde el inicio de una secuencia. Si hay menos elementos en la secuencia de los que se van a tomar, `Seq.take` produce una `System.InvalidOperationException`. La diferencia entre `Seq.take` y `Seq.truncate` es que `Seq.truncate` no genera un error si el número de elementos es menor que el número especificado.

En el código siguiente se muestra el comportamiento de y las diferencias entre `Seq.truncate` y `Seq.take`.

```
let mySeq = seq { for i in 1 .. 10 -> i*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takenSeq = Seq.take 5 mySeq

let truncatedSeq2 = Seq.truncate 20 mySeq
let takenSeq2 = Seq.take 20 mySeq

let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""

// Up to this point, the sequences are not evaluated.
// The following code causes the sequences to be evaluated.
truncatedSeq |> printSeq
truncatedSeq2 |> printSeq
takenSeq |> printSeq
// The following line produces a run-time error (in printSeq):
takenSeq2 |> printSeq
```

La salida, antes de que se produzca el error, es como se indica a continuación.

```
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
```

Mediante [Seq. takeWhile](#), puede especificar una función de predicado (una función booleana) y crear una secuencia a partir de otra secuencia que se compone de los elementos de la secuencia original para los que se `true` el predicado, pero que se detienen antes del primer elemento para que el predicado devuelve `false`.

[Seq. SKIP](#) devuelve una secuencia que omite un número especificado de los primeros elementos de otra secuencia y devuelve los elementos restantes. [Seq. skipwhile](#) (devuelve una secuencia que omite los primeros elementos de otra secuencia, siempre que el predicado devuelva `true` y, a continuación, devuelva los elementos restantes, empezando por el primer elemento para el que el predicado devuelve `false`.

En el ejemplo de código siguiente se muestra el comportamiento de y las diferencias entre `Seq.takeWhile`, `Seq.skip` y `Seq.skipWhile`.

```
// takeWhile
let mySeqLessThan10 = Seq.takeWhile (fun elem -> elem < 10) mySeq
mySeqLessThan10 |> printSeq

// skip
let mySeqSkipFirst5 = Seq.skip 5 mySeq
mySeqSkipFirst5 |> printSeq

// skipWhile
let mySeqSkipWhileLessThan10 = Seq.skipWhile (fun elem -> elem < 10) mySeq
mySeqSkipWhileLessThan10 |> printSeq
```

La salida es la siguiente.

```
1 4 9
36 49 64 81 100
16 25 36 49 64 81 100
```

Transformar secuencias

[Seq. en pares](#) crea una nueva secuencia en la que los elementos sucesivos de la secuencia de entrada se agrupan en tuplas.

```
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let seqPairwise = Seq.pairwise (seq { for i in 1 .. 10 -> i*i })
printSeq seqPairwise

printfn ""
let seqDelta = Seq.map (fun elem -> snd elem - fst elem) seqPairwise
printSeq seqDelta
```

[Seq. windowed](#) es como `Seq.pairwise`, salvo que en lugar de generar una secuencia de tuplas, genera una secuencia de matrices que contienen copias de elementos adyacentes (una *ventana*) de la secuencia. Especifique el número de elementos adyacentes que desee en cada matriz.

En el siguiente ejemplo de código se muestra el uso de `Seq.windowed`. En este caso, el número de elementos de la ventana es 3. En el ejemplo se usa `printSeq`, que se define en el ejemplo de código anterior.

```
let seqNumbers = [ 1.0; 1.5; 2.0; 1.5; 1.0; 1.5 ] :> seq<float>
let seqWindows = Seq.windowed 3 seqNumbers
let seqMovingAverage = Seq.map Array.average seqWindows
printfn "Initial sequence: "
printSeq seqNumbers
printfn "\nWindows of length 3: "
printSeq seqWindows
printfn "\nMoving average: "
printSeq seqMovingAverage
```

La salida es la siguiente.

Secuencia inicial:

```
1.0 1.5 2.0 1.5 1.0 1.5
```

```
Windows of length 3:
```

```
[|1.0; 1.5; 2.0|] [|1.5; 2.0; 1.5|] [|2.0; 1.5; 1.0|] [|1.5; 1.0; 1.5|]
```

```
Moving average:
```

```
1.5 1.666666667 1.5 1.333333333
```

Operaciones con varias secuencias

[Seq.zip](#) y [Seq.zip3](#) (tienen dos o tres secuencias y generan una secuencia de tuplas. Estas funciones son similares a las funciones correspondientes disponibles para [las listas](#). No hay ninguna funcionalidad correspondiente para separar una secuencia en dos o más secuencias. Si necesita esta funcionalidad para una secuencia, convierta la secuencia en una lista y use [List.unzip](#).

Ordenar, comparar y agrupar

Las funciones de ordenación compatibles con las listas también funcionan con secuencias. Esto incluye [Seq.Sort](#) y [Seq.sortBy](#). Estas funciones recorren en iteración toda la secuencia.

Se comparan dos secuencias mediante la función [Seq.compareWith](#) (. La función compara los elementos sucesivos a su vez y se detiene cuando encuentra el primer par diferente. Cualquier elemento adicional no contribuye a la comparación.

En el código siguiente se muestra el uso de `Seq.compareWith` .

```
let sequence1 = seq { 1 .. 10 }
let sequence2 = seq { 10 .. -1 .. 1 }

// Compare two sequences element by element.
let compareSequences =
    Seq.compareWith (fun elem1 elem2 ->
        if elem1 > elem2 then 1
        elif elem1 < elem2 then -1
        else 0)

let compareResult1 = compareSequences sequence1 sequence2
match compareResult1 with
| 1 -> printfn "Sequence1 is greater than sequence2."
| -1 -> printfn "Sequence1 is less than sequence2."
| 0 -> printfn "Sequence1 is equal to sequence2."
| _ -> failwith("Invalid comparison result.")
```

En el código anterior, solo se calcula y examina el primer elemento y el resultado es -1.

[Seq.countBy](#) toma una función que genera un valor denominado *clave* para cada elemento. Se genera una clave para cada elemento mediante una llamada a esta función en cada elemento. `Seq.countBy` devuelve una secuencia que contiene los valores de clave y un recuento del número de elementos que generaron cada valor de la clave.


```
let mySeq1 = seq { 1.. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let seqResult =
    mySeq1
    |> Seq.countBy (fun elem ->
        if elem % 3 = 0 then 0
        elif elem % 3 = 1 then 1
        else 2)

printSeq seqResult
```

La salida es la siguiente.

```
(1, 34) (2, 33) (0, 33)
```

La salida anterior muestra que había 34 elementos de la secuencia original que generaban los valores de la clave 1, 33 que generaban los valores de la clave 2 y 33 que generaban la clave 0.

Puede agrupar los elementos de una secuencia mediante una llamada a [Seq.groupBy](#). `Seq.groupBy` toma una secuencia y una función que genera una clave a partir de un elemento. La función se ejecuta en cada elemento de la secuencia. `Seq.groupBy` devuelve una secuencia de tuplas, donde el primer elemento de cada tupla es la clave y el segundo es una secuencia de elementos que generan esa clave.

En el ejemplo de código siguiente se muestra el uso de `Seq.groupBy` para particionar la secuencia de números de 1 a 100 en tres grupos que tienen los valores de clave DISTINCT 0, 1 y 2.

```
let sequence = seq { 1 .. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let sequences3 =
    sequence
    |> Seq.groupBy (fun index ->
        if (index % 3 = 0) then 0
        elif (index % 3 = 1) then 1
        else 2)

sequences3 |> printSeq
```

La salida es la siguiente.

```
(1, seq [1; 4; 7; 10; ...]) (2, seq [2; 5; 8; 11; ...]) (0, seq [3; 6; 9; 12; ...])
```

Puede crear una secuencia que elimine elementos duplicados mediante una llamada a [Seq.DISTINCT](#). O bien, puede usar [Seq.distinctby](#) (, que toma una función de generación de claves a la que se va a llamar en cada elemento. La secuencia resultante contiene elementos de la secuencia original que tienen claves únicas. los elementos posteriores que producen una clave duplicada en un elemento anterior se descartan.

En el siguiente ejemplo de código, se muestra el uso de `Seq.distinct`. `Seq.distinct` se muestra generando secuencias que representan números binarios y, a continuación, mostrando que los únicos elementos distintos son 0 y 1.

```

let binary n =
    let rec generateBinary n =
        if (n / 2 = 0) then [n]
        else (n % 2) :: generateBinary (n / 2)

    generateBinary n
    |> List.rev
    |> Seq.ofList

printfn "%A" (binary 1024)

let resultSequence = Seq.distinct (binary 1024)
printfn "%A" resultSequence

```

En el código siguiente se muestra `Seq.distinctBy` iniciando con una secuencia que contiene números positivos y negativos y utilizando la función de valor absoluto como la función de generación de claves. En la secuencia resultante faltan todos los números positivos que corresponden a los números negativos de la secuencia, ya que los números negativos aparecen antes en la secuencia y, por lo tanto, se seleccionan en lugar de los números positivos que tienen el mismo valor absoluto. valor o clave.

```

let inputSequence = { -5 .. 10 }
let printSeq seq1 = Seq.iter (printf "%A ") seq1

printfn "Original sequence: "
printSeq inputSequence

printfn "\nSequence with distinct absolute values: "
let seqDistinctAbsoluteValue = Seq.distinctBy (fun elem -> abs elem) inputSequence
printSeq seqDistinctAbsoluteValue

```

Secuencias de solo lectura y en caché

[Seq.ReadOnly](#) crea una copia de solo lectura de una secuencia. `Seq.readonly` resulta útil cuando se tiene una colección de lectura y escritura, como una matriz, y no se desea modificar la colección original. Esta función se puede utilizar para conservar la encapsulación de datos. En el ejemplo de código siguiente, se crea un tipo que contiene una matriz. Una propiedad expone la matriz, pero en lugar de devolver una matriz, devuelve una secuencia que se crea a partir de la matriz mediante `Seq.readonly`.

```

type ArrayContainer(start, finish) =
    let internalArray = [| start .. finish |]
    member this.RangeSeq = Seq.readonly internalArray
    member this.RangeArray = internalArray

let newArray = new ArrayContainer(1, 10)
let rangeSeq = newArray.RangeSeq
let rangeArray = newArray.RangeArray
// These lines produce an error:
//let myArray = rangeSeq :> int array
//myArray.[0] <- 0
// The following line does not produce an error.
// It does not preserve encapsulation.
rangeArray.[0] <- 0

```

[Seq.cache](#) crea una versión almacenada de una secuencia. Use `Seq.cache` para evitar la reevaluación de una secuencia o si tiene varios subprocesos que usan una secuencia, pero debe asegurarse de que cada elemento se actúa en una sola vez. Si tiene una secuencia que usan varios subprocesos, puede tener un subproceso que Enumere y calcule los valores de la secuencia original, y los subprocesos restantes puedan utilizar la secuencia almacenada en caché.

Realizar cálculos en secuencias

Las operaciones aritméticas simples son similares a las de las listas, como [Seq. Average](#), [Seq. SUM](#), [Seq. averageBy](#), [Seq. sumBy](#) (`,` etc.

[Seq. Fold](#), [Seq. reducey](#) [Seq. Scan](#) son como las funciones correspondientes que están disponibles para las listas. Las secuencias admiten un subconjunto de las variaciones completas de estas funciones que enumeran la compatibilidad. Para obtener más información y ejemplos, vea [listas](#).

Vea también

- [Referencia del lenguaje F#](#)
- [Tipos en F#](#)

Segmentos

05/02/2020 • 6 minutes to read • [Edit Online](#)

En F#, un segmento es un subconjunto de cualquier tipo de datos que tenga un método `GetSlice` en su definición o en una [extensión de tipo](#) en el ámbito. Normalmente se usa con F# matrices y listas. En este artículo se explica cómo tomar los segmentos F# de los tipos existentes y cómo definir sus propios segmentos.

Los segmentos son similares a los [indizadores](#), pero en lugar de producir un valor único de la estructura de datos subyacente, producen varios.

F#Actualmente tiene compatibilidad intrínseca con la segmentación de cadenas, listas, matrices y matrices 2D.

Segmentación básica con F# listas y matrices

Los tipos de datos más comunes que se segmentan F# son listas y matrices. En el ejemplo siguiente se muestra cómo hacerlo con listas:

```
// Generate a list of 100 integers
let fullList = [ 1 .. 100 ]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullList.[1..5]
printfn "Small slice: %A" smallSlice

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullList[..5]
printfn "Unbounded beginning slice: %A" unboundedBeginning

// Create a slice from an index to the end of the list
let unboundedEnd = fullList.[94..]
printfn "Unbounded end slice: %A" unboundedEnd
```

La segmentación de matrices es igual que las listas de segmentación:

```
// Generate an array of 100 integers
let fullArray = [| 1 .. 100 |]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullArray.[1..5]
printfn "Small slice: %A" smallSlice

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullArray[..5]
printfn "Unbounded beginning slice: %A" unboundedBeginning

// Create a slice from an index to the end of the list
let unboundedEnd = fullArray.[94..]
printfn "Unbounded end slice: %A" unboundedEnd
```

Segmentar matrices multidimensionales

F#admite matrices multidimensionales en la F# biblioteca principal. Al igual que con las matrices unidimensionales, los segmentos de matrices multidimensionales también pueden ser útiles. Sin embargo, la introducción de dimensiones adicionales asigna una sintaxis ligeramente diferente para que pueda tomar segmentos de filas y columnas específicas.

En los siguientes ejemplos se muestra cómo segmentar una matriz 2D:

```
// Generate a 3x3 2D matrix
let A = array2D [[1;2;3];[4;5;6];[7;8;9]]
printfn "Full matrix:\n %A" A

// Take the first row
let row0 = A.[0,*]
printfn "Row 0: %A" row0

// Take the first column
let col0 = A.[*,0]
printfn "Column 0: %A" col0

// Take all rows but only two columns
let subA = A.[*,0..1]
printfn "%A" subA

// Take two rows and all columns
let subA' = A.[0..1,*]
printfn "%A" subA'

// Slice a 2x2 matrix out of the full 3x3 matrix
let twoByTwo = A.[0..1,0..1]
printfn "%A" twoByTwo
```

La F# biblioteca principal no define actualmente `GetSlice` para las matrices 3D. Si desea segmentar matrices 3D u otras matrices de más dimensiones, defina el miembro de `GetSlice`.

Definir segmentos para otras estructuras de datos

La F# biblioteca principal define los segmentos para un conjunto limitado de tipos. Si desea definir segmentos para más tipos de datos, puede hacerlo en la propia definición de tipo o en una extensión de tipo.

Por ejemplo, aquí se muestra cómo puede definir segmentos para la clase `ArraySegment<T>` para permitir una manipulación de datos adecuada:

```
open System

type ArraySegment<'TItem> with
    member segment.GetSlice(start, finish) =
        let start = defaultArg start 0
        let finish = defaultArg finish segment.Count
        ArraySegment(segment.Array, segment.Offset + start, finish - start)

let arr = ArraySegment [| 1 .. 10 |]
let slice = arr.[2..5] //[ 3; 4; 5]
```

Usar la inclusión para evitar la conversión boxing si es necesario

Si va a definir segmentos para un tipo que es realmente un struct, recomendamos que `inline` el miembro `GetSlice`. El F# compilador optimiza los argumentos opcionales, evitando las asignaciones de montón como resultado de la segmentación. Esto es muy importante para las construcciones de segmentación como `Span<T>` que no se pueden asignar en el montón.

open System

```
type ReadOnlySpan<'T> with
    // Note the 'inline' in the member definition
    member inline sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)
```

```
type Span<'T> with
    // Note the 'inline' in the member definition
    member inline sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)
```

```
let printSpan (sp: Span<int>) =
    let arr = sp.ToArray()
    printfn "%A" arr
```

```
let sp = [| 1; 2; 3; 4; 5 |].AsSpan()
printSpan sp.[0..] // [|1; 2; 3; 4; 5|]
printSpan sp[..5] // [|1; 2; 3; 4; 5|]
printSpan sp.[0..3] // [|1; 2; 3|]
printSpan sp.[1..3] // |2; 3|]
```

Los F# segmentos integrados son de fin-inclusivo

Todos los sectores intrínsecos de F# son de fin-inclusivo; es decir, el límite superior se incluye en el segmento. Para un segmento determinado con el índice de inicio `x` y el final `y` de índice, el segmento resultante incluirá el valor de *YTH*.

```
// Define a new list
let xs = [1 .. 10]

printfn "%A" xs.[2..5] // Includes the 5th index
```

Vea también

- [Propiedades indizadas](#)

Opciones

23/10/2019 • 7 minutes to read • [Edit Online](#)

El tipo de opción F# en se utiliza cuando es posible que no exista un valor real para una variable o valor con nombre. Una opción tiene un tipo subyacente y puede contener un valor de ese tipo, o puede que no tenga un valor.

Comentarios

En el código siguiente se muestra una función que genera un tipo de opción.

```
let keepIfPositive (a : int) = if a > 0 then Some(a) else None
```

Como puede ver, si la entrada `a` es mayor que 0, `Some(a)` se genera. De lo contrario, se genera.

El valor `None` se utiliza cuando una opción no tiene un valor real. De lo contrario, `Some(...)` la expresión proporciona a la opción un valor. Los valores `Some` y `None` son útiles en la coincidencia de patrones, como en la `exists` función siguiente, `true` que devuelve si la opción tiene un `false` valor y si no lo hace.

```
let exists (x : int option) =  
    match x with  
    | Some(x) -> true  
    | None -> false
```

Usar opciones

Las opciones se usan normalmente cuando una búsqueda no devuelve un resultado de coincidencia, como se muestra en el código siguiente.

```
let rec tryFindMatch pred list =  
    match list with  
    | head :: tail -> if pred(head)  
                       then Some(head)  
                       else tryFindMatch pred tail  
    | [] -> None  
  
// result1 is Some 100 and its type is int option.  
let result1 = tryFindMatch (fun elem -> elem = 100) [ 200; 100; 50; 25 ]  
  
// result2 is None and its type is int option.  
let result2 = tryFindMatch (fun elem -> elem = 26) [ 200; 100; 50; 25 ]
```

En el código anterior, se busca una lista de forma recursiva. La función `tryFindMatch` toma una función `pred` de predicado que devuelve un valor booleano y una lista que se va a buscar. Si se encuentra un elemento que satisface el predicado, la recursividad finaliza y la función devuelve el valor como una opción en la `Some(head)` expresión. La recursividad finaliza cuando se encuentra una coincidencia con la lista vacía. En ese momento, no `head` se ha encontrado el valor y `None` se devuelve.

Muchas F# funciones de la biblioteca que buscan en una colección un valor que puede o no existir devuelven el `option` tipo. Por Convención, estas funciones comienzan con el `try` prefijo, por ejemplo `Seq.tryFindIndex` ,.

Las opciones también pueden ser útiles cuando es posible que un valor no exista, por ejemplo, si es posible que se produzca una excepción al intentar construir un valor. En el siguiente ejemplo código se muestra cómo hacerlo.

```
open System.IO
let openFile filename =
    try
        let file = File.Open (filename, FileMode.Create)
        Some(file)
    with
        | ex -> eprintf "An exception occurred with message %s" ex.Message
        None
```

La `openFile` función del ejemplo anterior tiene el tipo `string -> File option` porque devuelve un `File` objeto si el archivo se abre correctamente y `None` si se produce una excepción. Dependiendo de la situación, puede que no sea una opción de diseño adecuada para detectar una excepción en lugar de permitir que se propague.

Además, sigue siendo posible pasar `null` o un valor que sea NULL a las `Some` mayúsculas y minúsculas de una opción. Por lo general, esto se evita y, normalmente, está F# en programación rutinaria, pero es posible debido a la naturaleza de los tipos de referencia en .net.

Propiedades y métodos de la opción

El tipo de opción admite las siguientes propiedades y métodos.

PROPIEDAD O MÉTODO	TYPE	DESCRIPCIÓN
<code>None</code>	<code>'T option</code>	Propiedad estática que permite crear un valor de opción que tiene el <code>None</code> valor.
<code>Isnone (</code>	<code>bool</code>	Devuelve <code>true</code> si la opción tiene el <code>None</code> valor.
<code>IsSome</code>	<code>bool</code>	Devuelve <code>true</code> si la opción tiene un valor que no <code>None</code> es.
<code>Existen</code>	<code>'T option</code>	Miembro estático que crea una opción que tiene un valor que no <code>None</code> es.
<code>Valor</code>	<code>'T</code>	Devuelve el valor subyacente o produce una <code>System.NullReferenceException</code> excepción si el valor de es. <code>None</code>

Módulo de opción

Hay un módulo, `opción`, que contiene funciones útiles que realizan operaciones en las opciones. Algunas funciones repiten la funcionalidad de las propiedades pero son útiles en contextos en los que se necesita una función. `Option.IsSome` (y `Option.isnone (` son dos funciones de módulo que prueban si una opción contiene un valor. `Option.Get` obtiene el valor, si lo hay. Si no hay ningún valor, se produce una `System.ArgumentException` excepción.

La función `Option.bind` ejecuta una función en el valor, si hay un valor. La función debe tomar exactamente un argumento y su tipo de parámetro debe ser el tipo de opción. El valor devuelto de la función es otro tipo de

opción.

El módulo de opciones también incluye funciones que corresponden a las funciones que están disponibles para las listas, matrices, secuencias y otros tipos de colección. Estas funciones incluyen `Option.map`, `Option.iter`, `Option.forall`, `Option.exists`, `Option.foldBack` y `Option.count`. `Option.fold` Estas funciones permiten usar opciones como una colección de cero o un elemento. Para obtener más información y ejemplos, vea la explicación de las funciones de colección en [las listas](#).

Convertir a otros tipos

Las opciones se pueden convertir en listas o matrices. Cuando una opción se convierte en cualquiera de estas estructuras de datos, la estructura de datos resultante tiene cero o un elemento. Para convertir una opción en una matriz, use `Option.toArray`. Para convertir una opción en una lista, use `Option.toList`.

Vea también

- [Referencia del lenguaje F#](#)
- [Tipos en F#](#)

Opciones de valores

05/12/2019 • 3 minutes to read • [Edit Online](#)

El tipo de opción Value F# se usa cuando se mantienen las dos circunstancias siguientes:

1. Un escenario es adecuado para una [F# opción](#).
2. El uso de una estructura proporciona una ventaja de rendimiento en su escenario.

No todos los escenarios sensibles al rendimiento se "resuelven" mediante el uso de Structs. Debe tener en cuenta el costo adicional de copia cuando se usan en lugar de los tipos de referencia. Sin embargo, F# los programas de gran tamaño suelen crear instancias de muchos tipos opcionales que fluyen a través de rutas de acceso activas y, en tales casos, los Structs pueden obtener un mejor rendimiento general a lo largo de la duración de un programa.

de esquema JSON

La opción Value se define como una [Unión discriminada de struct](#) que es similar al tipo de opción de referencia. Su definición se puede considerar de esta manera:

```
[<StructuralEquality; StructuralComparison>]
[<Struct>]
type ValueOption<'T> =
    | ValueNone
    | ValueSome of 'T
```

La opción Value se ajusta a la igualdad estructural y a la comparación. La principal diferencia es que el nombre compilado, el nombre de tipo y los nombres de caso indican que se trata de un tipo de valor.

Usar opciones de valor

Las opciones de valor se usan de la misma manera que [las opciones](#). `ValueSome` se utiliza para indicar que un valor está presente y `ValueNone` se utiliza cuando un valor no está presente:

```
let tryParseDateTime (s: string) =
    match System.DateTime.TryParse(s) with
    | (true, dt) -> ValueSome dt
    | (false, _) -> ValueNone

let possibleDateString1 = "1990-12-25"
let possibleDateString2 = "This is not a date"

let result1 = tryParseDateTime possibleDateString1
let result2 = tryParseDateTime possibleDateString2

match (result1, result2) with
| ValueSome d1, ValueSome d2 -> printfn "Both are dates!"
| ValueSome d1, ValueNone -> printfn "Only the first is a date!"
| ValueNone, ValueSome d2 -> printfn "Only the second is a date!"
| ValueNone, ValueNone -> printfn "None of them are dates!"
```

Como con [las opciones](#), la Convención de nomenclatura para una función que devuelve `ValueOption` es prefijarla con `try`.

Propiedades y métodos de la opción Value

En este momento hay una propiedad para las opciones de valor: `Value`. Se produce una [InvalidOperationException](#) si no hay ningún valor presente cuando se invoca esta propiedad.

Funciones de opción de valor

El módulo `ValueOption` de FSharp.Core contiene una funcionalidad equivalente al módulo `Option`. Hay algunas diferencias en el nombre, como `defaultValueArg`:

```
val defaultValueArg : arg:'T voption -> defaultValue:'T -> 'T
```

Esto actúa como `defaultArg` en el módulo de `Option`, pero funciona en una opción de valor en su lugar.

Vea también

- [Opciones](#)

Resultados

04/11/2019 • 3 minutes to read • [Edit Online](#)

A partir F# de 4.1, hay un tipo de `Result<'T, 'TFailure>` que puede usar para escribir código tolerante a errores que se puede componer.

Sintaxis

```
// The definition of Result in FSharp.Core
[<StructuralEquality; StructuralComparison>]
[<CompiledName("FSharpResult`2")>]
[<Struct>]
type Result<'T, 'TError> =
    | Ok of ResultValue:'T
    | Error of ErrorValue:'TError
```

Comentarios

Tenga en cuenta que el tipo de resultado es una [Unión discriminada de struct](#), que es F# otra característica presentada en 4.1. La semántica de igualdad estructural se aplica aquí.

El tipo de `Result` se usa normalmente en el control de errores de Monad, que a menudo se conoce como [programación orientada a ferrocarriles](#) dentro de la F# comunidad. En el siguiente ejemplo trivial se muestra este enfoque.

```
// Define a simple type which has fields that can be validated
type Request =
    { Name: string
      Email: string }

// Define some logic for what defines a valid name.
//
// Generates a Result which is an Ok if the name validates;
// otherwise, it generates a Result which is an Error.
let validateName req =
    match req.Name with
    | null -> Error "No name found."
    | "" -> Error "Name is empty."
    | "bananas" -> Error "Bananas is not a name."
    | _ -> Ok req

// Similarly, define some email validation logic.
let validateEmail req =
    match req.Email with
    | null -> Error "No email found."
    | "" -> Error "Email is empty."
    | s when s.EndsWith("bananas.com") -> Error "No email from bananas.com is allowed."
    | _ -> Ok req

let validateRequest reqResult =
    reqResult
    |> Result.bind validateName
    |> Result.bind validateEmail

let test() =
    // Now, create a Request and pattern match on the result.
    let req1 = { Name = "Phillip"; Email = "phillip@contoso.biz" }
    let res1 = validateRequest (Ok req1)
    match res1 with
    | Ok req -> printfn "My request was valid! Name: %s Email %s" req.Name req.Email
    | Error e -> printfn "Error: %s" e
    // Prints: "My request was valid! Name: Phillip Email: phillip@consoto.biz"

    let req2 = { Name = "Phillip"; Email = "phillip@bananas.com" }
    let res2 = validateRequest (Ok req2)
    match res2 with
    | Ok req -> printfn "My request was valid! Name: %s Email %s" req.Name req.Email
    | Error e -> printfn "Error: %s" e
    // Prints: "Error: No email from bananas.com is allowed."

test()
```

Como puede ver, es muy fácil encadenar varias funciones de validación si las obliga a que devuelvan un `Result`. Esto le permite dividir la funcionalidad como esta en pequeñas piezas que son comparables según sea necesario. También tiene el valor agregado de *exigir* el uso de la [coincidencia de patrones](#) al final de una ronda de validación, que a su vez exige un mayor grado de corrección del programa.

Vea también

- [Uniones discriminadas](#)
- [Coincidencia de patrones](#)

Genéricos

23/10/2019 • 11 minutes to read • [Edit Online](#)

Los valores de función, los métodos, las propiedades y los tipos agregados de F#, como las clases, los registros y las uniones discriminadas, pueden ser *genéricos*. Las construcciones genéricas contienen al menos un parámetro de tipo que, por lo general, proporciona el usuario de la construcción genérica. Las funciones y los tipos genéricos permiten escribir código que funciona con una variedad de tipos sin repetir el código para cada tipo. Convertir el código en genérico puede ser sencillo en F#, porque a menudo se infiere implícitamente el código para que sea genérico por la inferencia de tipos del compilador y los mecanismos de generalización automáticos.

Sintaxis

```
// Explicitly generic function.
let function-name<type-parameters> parameter-list =
    function-body

// Explicitly generic method.
[ static ] member object-identifer.method-name<type-parameters> parameter-list [ return-type ] =
    method-body

// Explicitly generic class, record, interface, structure,
// or discriminated union.
type type-name<type-parameters> type-definition
```

Comentarios

La declaración explícita de una función o tipo genérico es muy parecida a la de una función o tipo no genérico, excepto en la especificación (y uso) de los parámetros de tipos, en corchetes angulares después del nombre de la función o el tipo.

Las declaraciones a menudo son implícitamente genéricas. Si no se especifica completamente el tipo de cada parámetro que se usa para componer una función o un tipo, el compilador intenta inferir el tipo de cada parámetro, valor y variable del código que se escribe. Para más información, vea [Inferencia de tipos](#). Si el código del tipo o función no limita de otro modo los tipos de los parámetros, la función o el tipo son implícitamente genéricos. Este proceso se denomina *generalización automática*. Existen algunas limitaciones en la generalización automática. Por ejemplo, si el compilador de F# no puede inferir los tipos de una construcción genérica, informa sobre un error que hace referencia a una restricción denominada *restricción de valor*. En ese caso, puede que sea necesario agregar algunas anotaciones de tipo. Para más información sobre la generalización automática y la restricción de valor, y cómo cambiar el código para resolver el problema, vea [Generalización automática](#).

En la sintaxis anterior, *parámetros de tipo* es una lista separada por comas de parámetros que representan tipos desconocidos, cada uno de los cuales comienza por una comilla simple, opcionalmente con una cláusula de restricción que limita aún más los tipos que se pueden usar para ese tipo de parámetro. Para obtener información sobre la sintaxis de las cláusulas de restricción de varios tipos y otra información sobre restricciones, vea [Restricciones](#).

La *definición de tipos* de la sintaxis es la misma que la definición de tipos de un tipo no genérico. Incluye los parámetros del constructor para un tipo de clase, una cláusula `as` opcional, el símbolo igual, los campos de registro, la cláusula `inherit`, las opciones para una unión discriminada, enlaces `let` y `do`, definiciones de miembros y todo lo que se permite en una definición de tipos no genéricos.

Los demás elementos de la sintaxis son los mismos que los de los tipos y funciones no genéricos. Por ejemplo,

identificador de objeto es un identificador que representa el objeto contenedor.

Las propiedades, campos y constructores no pueden ser más genéricos que el tipo envolvente. Además, los valores de un módulo no pueden ser genéricos.

Construcciones genéricas implícitas

Cuando el compilador de F# infiere los tipos del código, trata automáticamente todas las funciones que pueden ser genéricas como tales. Si se especifica un tipo explícitamente, como un tipo de parámetro, se impide la generalización automática.

En el ejemplo de código siguiente, `makeList` es genérico aunque no se ha declarado como tal y sus parámetros tampoco.

```
let makeList a b =  
    [a; b]
```

La firma de la función se infiere como `'a -> 'a -> 'a list`. Observe que en este ejemplo `a` y `b` se infieren para tener el mismo tipo. Esto se debe a que se incluyen en una lista y todos los elementos de una lista deben ser del mismo tipo.

También se puede hacer genérica una función usando la sintaxis de comillas simples en una anotación de tipo para indicar que un tipo de parámetro es un parámetro de tipo genérico. En el código siguiente, `function1` es genérico porque sus parámetros se han declarado de esta manera, como parámetros de tipo.

```
let function1 (x: 'a) (y: 'a) =  
    printfn "%A %A" x y
```

Construcciones explícitamente genéricas

Para convertir una función en genérica, se declaran explícitamente sus parámetros de tipo entre corchetes angulares (`<type-parameter>`). Esto se ilustra en el código siguiente:

```
let function2<'T> x y =  
    printfn "%A, %A" x y
```

Uso de construcciones genéricas

Cuando se usan funciones o métodos genéricos, no siempre es necesario especificar los argumentos de tipo. El compilador usa la inferencia de tipos para inferir los argumentos de tipo adecuados. Si la ambigüedad se mantiene, se pueden proporcionar argumentos de tipo en corchetes angulares y separarlos con comas.

En el código siguiente se muestra el uso de las funciones que se definen en las secciones anteriores.

```
// In this case, the type argument is inferred to be int.
function1 10 20
// In this case, the type argument is float.
function1 10.0 20.0
// Type arguments can be specified, but should only be specified
// if the type parameters are declared explicitly. If specified,
// they have an effect on type inference, so in this example,
// a and b are inferred to have type int.
let function3 a b =
    // The compiler reports a warning:
    function1<int> a b
    // No warning.
    function2<int> a b
```

NOTE

Hay dos maneras de hacer referencia a un tipo genérico por nombre. Por ejemplo, `list<int>` e `int list` son dos formas de hacer referencia a un tipo genérico `list` que tiene un único argumento de tipo `int`. Por convención, la segunda forma solo se usa con tipos de F# integrados como `list` y `option`. Si hay varios argumentos de tipo, normalmente se usa la sintaxis `Dictionary<int, string>`, pero también se puede usar la sintaxis `(int, string) Dictionary`.

Caracteres comodín como argumentos de tipo

Para especificar que el compilador debe inferir un argumento de tipo, se puede usar un subrayado o símbolo comodín (`_`), en lugar de un argumento de tipo con nombre. Esto se muestra en el código siguiente.

```
let printSequence (sequence1: Collections.seq<_>) =
    Seq.iter (fun elem -> printf "%s " (elem.ToString())) sequence1
```

Restricciones en los tipos y funciones genéricos

En una definición de función o tipo genérico, solo se pueden usar las construcciones que se sabe que están disponibles en el parámetro de tipo genérico. Esto es necesario para habilitar la comprobación de las llamadas a métodos y funciones en tiempo de compilación. Si se declaran explícitamente los parámetros de tipo, se podrá aplicar una restricción explícita a un parámetro de tipo genérico para indicar al compilador que están disponibles determinados métodos y funciones. Pero si se permite que el compilador de F# infiera los tipos de parámetro genéricos, determinará las restricciones apropiadas. Para más información, vea [Restricciones](#).

Parámetros de tipo resueltos estáticamente

Hay dos clases de parámetros de tipo que se pueden usar en programas de F#. La primera son los parámetros de tipo genérico de la naturaleza descrita en las secciones anteriores. Esta primera clase equivale a los parámetros de tipo genérico que se usan en lenguajes como Visual Basic y C#. Otra clase de parámetro de tipo es específica de F# y se denomina *parámetro de tipo resuelto estáticamente*. Para obtener información sobre estas construcciones, vea [Parámetros de tipo resueltos estáticamente](#).

Ejemplos


```

// A generic function.
// In this example, the generic type parameter 'a' makes function3 generic.
let function3 (x : 'a) (y : 'a) =
    printf "%A %A" x y

// A generic record, with the type parameter in angle brackets.
type GR<'a> =
    {
        Field1: 'a;
        Field2: 'a;
    }

// A generic class.
type C<'a>(a : 'a, b : 'a) =
    let z = a
    let y = b
    member this.GenericMethod(x : 'a) =
        printfn "%A %A %A" x y z

// A generic discriminated union.
type U<'a> =
    | Choice1 of 'a
    | Choice2 of 'a * 'a

type Test() =
    // A generic member
    member this.Function1<'a>(x, y) =
        printfn "%A, %A" x y

    // A generic abstract method.
    abstract abstractMethod<'a, 'b> : 'a * 'b -> unit
    override this.abstractMethod<'a, 'b>(x:'a, y:'b) =
        printfn "%A, %A" x y

```

Vea también

- [Referencia del lenguaje](#)
- [Tipos](#)
- [Parámetros de tipo resueltos estáticamente](#)
- [Genéricos](#)
- [Generalización automática](#)
- [Restricciones](#)

Generalización automática

23/10/2019 • 7 minutes to read • [Edit Online](#)

F# utiliza la inferencia de tipos para evaluar los tipos de funciones y expresiones. En este tema se describe cómo generaliza automáticamente los argumentos y los tipos de funciones para que funcionen con varios tipos cuando sea posible.

Generalización automática

El F# compilador, cuando realiza la inferencia de tipos en una función, determina si un parámetro determinado puede ser genérico. El compilador examina cada parámetro y determina si la función tiene una dependencia en el tipo específico de ese parámetro. Si no es así, se deduce que el tipo es genérico.

En el ejemplo de código siguiente se muestra una función que el compilador deduce que es genérica.

```
let max a b = if a > b then a else b
```

El tipo se deduce como `'a -> 'a -> 'a`.

El tipo indica que se trata de una función que toma dos argumentos del mismo tipo desconocido y devuelve un valor del mismo tipo. Uno de los motivos por los que la función anterior puede ser genérico es que el operador "mayor que" (`>`) es genérico. El operador mayor que tiene la firma `'a -> 'a -> bool`. No todos los operadores son genéricos y, si el código de una función usa un tipo de parámetro junto con una función o un operador no genéricos, ese tipo de parámetro no se puede generalizar.

Dado `max` que es genérico, se puede usar con tipos `int` como, `float`, etc., como se muestra en los ejemplos siguientes.

```
let biggestFloat = max 2.0 3.0
let biggestInt = max 2 3
```

Sin embargo, los dos argumentos deben ser del mismo tipo. La firma es `'a -> 'a -> 'a`, no `'a -> 'b -> 'a`. Por lo tanto, el código siguiente genera un error porque los tipos no coinciden.

```
// Error: type mismatch.
let biggestIntFloat = max 2.0 3
```

La `max` función también funciona con cualquier tipo que admita el operador mayor que. Por lo tanto, también puede utilizarlo en una cadena, como se muestra en el código siguiente.

```
let testString = max "cab" "cat"
```

Restricción de valor

El compilador realiza la generalización automática solo en definiciones de función completas que tienen argumentos explícitos y en valores inmutables simples.

Esto significa que el compilador emite un error si se intenta compilar código que no está suficientemente restringido para ser un tipo específico, sino que tampoco se puede generalizar. El mensaje de error para este

problema hace referencia a esta restricción en la generalización automática de los valores como *restricción de valor*.

Normalmente, el error de restricción de valor se produce cuando se desea que una construcción sea genérica pero el compilador no tiene información suficiente para generalizarla, o cuando se omite involuntariamente suficiente información de tipo en una construcción no genérica. La solución al error de restricción de valor es proporcionar información más explícita para restringir más el problema de inferencia de tipos de una de las siguientes maneras:

- Restrinja un tipo para que no sea genérico agregando una anotación de tipo explícito a un valor o parámetro.
- Si el problema utiliza una construcción no generalizable para definir una función genérica, como una composición de función o argumentos de función curificados aplicados incompletamente, intente volver a escribir la función como una definición de función normal.
- Si el problema es una expresión que es demasiado compleja para generalizarla, puede convertirla en una función agregando un parámetro adicional sin usar.
- Agregue parámetros de tipo genérico explícitos. Esta opción rara vez se usa.
- En los siguientes ejemplos de código se muestra cada uno de estos escenarios.

Caso 1: Una expresión demasiado compleja. En este ejemplo, la lista `counter` está pensada para `int option ref` ser, pero no se define como un valor inmutable simple.

```
let counter = ref None
// Adding a type annotation fixes the problem:
let counter : int option ref = ref None
```

Caso 2: Usar una construcción no generalizable para definir una función genérica. En este ejemplo, la construcción no es generalizable porque implica la aplicación parcial de argumentos de función.

```
let maxhash = max << hash
// The following is acceptable because the argument for maxhash is explicit:
let maxhash obj = (max << hash) obj
```

Caso 3: Agregar un parámetro adicional sin usar. Dado que esta expresión no es lo suficientemente sencilla para la generalización, el compilador emite el error de restricción de valor.

```
let emptyList10 = Array.create 10 []
// Adding an extra (unused) parameter makes it a function, which is generalizable.
let emptyList10 () = Array.create 10 []
```

Caso 4: Agregar parámetros de tipo.

```
let arrayOf10Lists = Array.create 10 []
// Adding a type parameter and type annotation lets you write a generic value.
let arrayOf10Lists<'T> = Array.create 10 ([]:'T list)
```

En el último caso, el valor se convierte en una función de tipo, que se puede usar para crear valores de muchos tipos diferentes, como se indica a continuación:

```
let intLists = arrayOf10Lists<int>  
let floatLists = arrayOf10Lists<float>
```

Vea también

- [Inferencia de tipos](#)
- [Genéricos](#)
- [Parámetros de tipo resueltos estáticamente](#)
- [Restricciones](#)

Restricciones

05/11/2019 • 7 minutes to read • [Edit Online](#)

En este tema se describen las restricciones que se pueden aplicar a los parámetros de tipo genérico para especificar los requisitos de un argumento de tipo en una función o un tipo genérico.

Sintaxis

```
type-parameter-list when constraint1 [ and constraint2]
```

Comentarios

Hay varias restricciones diferentes que puede aplicar para limitar los tipos que se pueden usar en un tipo genérico. En la tabla siguiente se enumeran y se describen estas restricciones.

RESTRICCIÓN	SINTAXIS	DESCRIPCIÓN
Restricción de tipo	<i>type-parameter</i> :> <i>Type</i>	El tipo proporcionado debe ser igual o derivarse del tipo especificado, o bien, si el tipo es una interfaz, el tipo proporcionado debe implementar la interfaz.
Restricción null	<i>type-parameter</i> : null	El tipo proporcionado debe admitir el literal null. Esto incluye todos los tipos de objetos de F# .net, pero no los tipos de lista, tupla, función, clase, registro o Unión.
Restricción de miembro explícita	<i>[() tipo-parámetro [o... o tipo-parámetro]]</i> : (<i>firma de miembro</i>)	Al menos uno de los argumentos de tipo proporcionados debe tener un miembro que tenga la firma especificada; no está pensado para su uso común. Los miembros deben estar definidos explícitamente en el tipo o parte de una extensión de tipo implícita para ser destinos válidos para una restricción de miembro explícita.
Restricción de constructor	<i>type-parameter</i> : (New: unit-> 'a)	El tipo proporcionado debe tener un constructor sin parámetros.
Restricción de tipo de valor	: struct	El tipo proporcionado debe ser un tipo de valor de .NET.
Restricción de tipo de referencia	: no struct	El tipo proporcionado debe ser un tipo de referencia de .NET.
Restricción de tipo de enumeración	: enumeración<> <i>de tipo subyacente</i>	El tipo proporcionado debe ser un tipo enumerado que tenga el tipo subyacente especificado. no está pensado para su uso común.

RESTRICCIÓN	SINTAXIS	DESCRIPCIÓN
Restricción de delegado	: delegado < <i>tupla-Parameter-Type</i> , <i>return-type</i> >	El tipo proporcionado debe ser un tipo de delegado que tenga los argumentos y el valor devuelto especificados. no está pensado para su uso común.
Restricción de comparación	: comparación	El tipo proporcionado debe admitir la comparación.
Restricción de igualdad	: igualdad	El tipo proporcionado debe admitir la igualdad.
Restricción no administrada	: no administrado	El tipo proporcionado debe ser un tipo no administrado. Los tipos no administrados son algunos tipos primitivos (<code>sbyte</code> , <code>byte</code> , <code>char</code> , <code>nativeint</code> , <code>unativeint</code> , <code>float32</code> , <code>float</code> , <code>int16</code> , <code>uint16</code> , <code>int32</code> , <code>uint32</code> , <code>int64</code> , <code>uint64</code> , o <code>decimal</code>), tipos de enumeración, <code>nativeptr<_></code> o una estructura no genérica cuyos campos son tipos no administrados.

Tiene que agregar una restricción cuando el código tiene que usar una característica que está disponible en el tipo de restricción, pero no en los tipos en general. Por ejemplo, si usa la restricción de tipo para especificar un tipo de clase, puede usar cualquiera de los métodos de esa clase en la función o el tipo genérico.

A veces se requiere la especificación de restricciones cuando se escriben parámetros de tipo de forma explícita, ya que sin una restricción, el compilador no tiene ninguna manera de comprobar que las características que está usando estarán disponibles en cualquier tipo que se pueda proporcionar en tiempo de ejecución para el tipo parámetro.

Las restricciones más comunes que se usan en el F# código son restricciones de tipo que especifican clases base o interfaces. La F# biblioteca usa las demás restricciones para implementar ciertas funciones, como la restricción explícita de miembros, que se usa para implementar la sobrecarga de operadores para los operadores aritméticos, o se proporcionan principalmente porque F# admite el conjunto completo de restricciones admitidas por el Common Language Runtime.

Durante el proceso de inferencia de tipos, el compilador deduce algunas restricciones automáticamente. Por ejemplo, si usa el operador `+` en una función, el compilador deduce una restricción de miembro explícita en los tipos de variables que se usan en la expresión.

En el código siguiente se muestran algunas declaraciones de restricción:

```

// Base Type Constraint
type Class1<'T when 'T :> System.Exception> =
class end

// Interface Type Constraint
type Class2<'T when 'T :> System.IComparable> =
class end

// Null constraint
type Class3<'T when 'T : null> =
class end

// Member constraint with instance member
type Class5<'T when 'T : (member Method1 : 'T -> int)> =
class end

// Member constraint with property
type Class6<'T when 'T : (member Property1 : int)> =
class end

// Constructor constraint
type Class7<'T when 'T : (new : unit -> 'T)>() =
member val Field = new 'T()

// Reference type constraint
type Class8<'T when 'T : not struct> =
class end

// Enumeration constraint with underlying value specified
type Class9<'T when 'T : enum<uint32>> =
class end

// 'T must implement IComparable, or be an array type with comparable
// elements, or be System.IntPtr or System.UIntPtr. Also, 'T must not have
// the NoComparison attribute.
type Class10<'T when 'T : comparison> =
class end

// 'T must support equality. This is true for any type that does not
// have the NoEquality attribute.
type Class11<'T when 'T : equality> =
class end

type Class12<'T when 'T : delegate<obj * System.EventArgs, unit>> =
class end

type Class13<'T when 'T : unmanaged> =
class end

// Member constraints with two type parameters
// Most often used with static type parameters in inline functions
let inline add(value1 : ^T when ^T : (static member (+) : ^T * ^T -> ^T), value2: ^T) =
value1 + value2

// ^T and ^U must support operator +
let inline heterogenousAdd(value1 : ^T when (^T or ^U) : (static member (+) : ^T * ^U -> ^T), value2 : ^U)
=
value1 + value2

// If there are multiple constraints, use the and keyword to separate them.
type Class14<'T,'U when 'T : equality and 'U : equality> =
class end

```

Vea también

- [Genéricos](#)
- [Restricciones](#)

Parámetros de tipo resueltos estáticamente

04/11/2019 • 6 minutes to read • [Edit Online](#)

Un *parámetro de tipo resuelto estáticamente* es un parámetro de tipo que se reemplaza con un tipo real en tiempo de compilación en lugar de en tiempo de ejecución. Va precedido por el símbolo de intercalación (^).

Sintaxis

```
^type-parameter
```

Comentarios

En el lenguaje F#, hay dos clases de parámetros de tipo. En primer lugar está el parámetro de tipo genérico estándar. Estos parámetros se indican mediante un apóstrofo ('), como en 'T y 'U. Equivalen a los parámetros de tipo genérico de otros lenguajes .NET Framework. El otro tipo de parámetro se resuelve estáticamente y se indica mediante el símbolo de intercalación, como en ^T y ^U.

Los parámetros de tipo resueltos estáticamente son sobre todo útiles cuando se usan con restricciones de miembro, que son restricciones que permiten especificar que un argumento de tipo debe tener uno o varios miembros determinados. Este tipo de restricción no se puede crear mediante un parámetro de tipo genérico normal.

En la siguiente tabla, se resumen las similitudes y las diferencias entre las dos clases de parámetros de tipo.

CARACTERÍSTICA	GENÉRICO	SE RESUELVE ESTÁTICAMENTE
Sintaxis	'T, 'U	^T, ^U
Tiempo de resolución	Tiempo de ejecución	Tiempo de compilación
Restricciones de miembro	No se puede utilizar con restricciones de miembro.	Se puede utilizar con restricciones de miembro.
Generación de código	Un tipo o método con parámetros de tipo genérico estándar da lugar a la generación de un solo tipo o método genérico.	Se generan varias instancias de los tipos y métodos, una por cada tipo que se necesita.
Uso con tipos	Se puede utilizar con tipos.	No se puede utilizar con tipos.
Uso con funciones inline	No. Una función inline no puede tener un parámetro de tipo genérico estándar.	Sí. Los parámetros de tipo resueltos estáticamente no se pueden utilizar con funciones o métodos que no sean inline.

Muchas funciones de la biblioteca básica de F#, sobre todo los operadores, tienen parámetros de tipo que se resuelven estáticamente. Se trata de funciones y operadores inline, que dan lugar a una generación de código eficaz para los cálculos numéricos.

Los métodos y funciones inline que usan operadores u otras funciones que utilicen parámetros de tipo resueltos

estáticamente, también pueden emplear ellos mismos este tipo de parámetro. En muchas ocasiones, durante la inferencia de tipos, se deduce que esas funciones inline tienen parámetros de tipo que se resuelven estáticamente. En el siguiente ejemplo, se muestra una definición de operador para la cual se deduce que tiene un parámetro de tipo que se resuelve estáticamente.

```
let inline (+@) x y = x + x * y
// Call that uses int.
printfn "%d" (1 +@ 1)
// Call that uses float.
printfn "%f" (1.0 +@ 0.5)
```

El tipo resuelto de `(+@)` se basa en el uso de `(+)` y `(*)`, que hacen que mediante la inferencia de tipos se deduzcan las restricciones de miembro en los parámetros de tipo estáticamente resueltos. Tal y como se muestra en el intérprete de F#, el tipo resuelto es el siguiente.

```
^a -> ^c -> ^d
when (^a or ^b) : (static member ( + ) : ^a * ^b -> ^d) and
(^a or ^c) : (static member ( * ) : ^a * ^c -> ^b)
```

La salida es la siguiente.

```
2
1.500000
```

A partir F# de 4.1, también puede especificar nombres de tipo concretos en firmas de parámetros de tipo resueltos estáticamente. En versiones anteriores del lenguaje, el compilador podía inferir el nombre del tipo, pero en realidad no se podía especificar en la signatura. A partir F# de 4.1, también puede especificar nombres de tipo concretos en firmas de parámetros de tipo resueltos estáticamente. Por ejemplo:

```
let inline konst x _ = x

type CFuncutor() =
    static member inline fmap (f: ^a -> ^b, a: ^a list) = List.map f a
    static member inline fmap (f: ^a -> ^b, a: ^a option) =
        match a with
        | None -> None
        | Some x -> Some (f x)

    // default implementation of replace
    static member inline replace< ^a, ^b, ^c, ^d, ^e when ^a :> CFuncutor and (^a or ^d): (static member fmap:
(^b -> ^c) * ^d -> ^e) > (a, f) =
        ((^a or ^d) : (static member fmap : (^b -> ^c) * ^d -> ^e) (konst a, f))

    // call overridden replace if present
    static member inline replace< ^a, ^b, ^c when ^b: (static member replace: ^a * ^b -> ^c)>(a: ^a, f: ^b) =
        (^b : (static member replace: ^a * ^b -> ^c) (a, f))

let inline replace_instance< ^a, ^b, ^c, ^d when (^a or ^c): (static member replace: ^b * ^c -> ^d)> (a: ^b,
f: ^c) =
    ((^a or ^c): (static member replace: ^b * ^c -> ^d) (a, f))

// Note the concrete type 'CFuncutor' specified in the signature
let inline replace (a: ^a) (f: ^b): ^a0 when (CFuncutor or ^b): (static member replace: ^a * ^b -> ^a0) =
    replace_instance<CFuncutor, _, _, _> (a, f)
```

Vea también

- [Genéricos](#)
- [Inferencia de tipos](#)
- [Generalización automática](#)
- [Restricciones](#)
- [Funciones insertadas](#)

Registros

23/10/2019 • 11 minutes to read • [Edit Online](#)

Los registros representan agregados simples de valores con nombre, opcionalmente con miembros. Pueden ser Structs o tipos de referencia. Son tipos de referencia de forma predeterminada.

Sintaxis

```
[ attributes ]
type [accessibility-modifier] typename =
    { [ mutable ] label1 : type1;
      [ mutable ] label2 : type2;
      ... }
[ member-list ]
```

Comentarios

En la sintaxis anterior, *TypeName* es el nombre del tipo de registro, *Label1* y *Label2* son nombres de valores, denominados *etiquetas*, y *Type1* y *tipo2* son los tipos de estos valores. *member-List* es la lista opcional de miembros del tipo. Puede usar el `[<Struct>]` atributo para crear un registro struct en lugar de un registro que sea un tipo de referencia.

A continuación se muestran algunos ejemplos.

```
// Labels are separated by semicolons when defined on the same line.
type Point = { X: float; Y: float; Z: float; }

// You can define labels on their own line with or without a semicolon.
type Customer =
    { First: string
      Last: string;
      SSN: uint32
      AccountNumber: uint32; }

// A struct record.
[<Struct>]
type StructPoint =
    { X: float
      Y: float
      Z: float }
```

Cuando cada etiqueta está en una línea independiente, el punto y coma es opcional.

Puede establecer valores en expresiones conocidas como *expresiones de registro*. El compilador deduce el tipo a partir de las etiquetas utilizadas (si las etiquetas son suficientemente distintas de las de otros tipos de registro). Las llaves ({}) incluyen la expresión de registro. En el código siguiente se muestra una expresión de registro que inicializa un registro con tres elementos Float `x` con `y` etiquetas `z`, `y`.

```
let mypoint = { X = 1.0; Y = 1.0; Z = -1.0; }
```

No utilice la forma abreviada si puede haber otro tipo que también tenga las mismas etiquetas.

```
type Point = { X: float; Y: float; Z: float; }  
type Point3D = { X: float; Y: float; Z: float }  
// Ambiguity: Point or Point3D?  
let mypoint3D = { X = 1.0; Y = 1.0; Z = 0.0; }
```

Las etiquetas del tipo declarado más recientemente tienen prioridad sobre las del tipo declarado previamente, por lo que en el ejemplo anterior, `mypoint3D` se deduce `Point3D` que es. Puede especificar explícitamente el tipo de registro, como en el código siguiente.

```
let myPoint1 = { Point.X = 1.0; Y = 1.0; Z = 0.0; }
```

Los métodos se pueden definir para tipos de registro del mismo modo que para los tipos de clase.

Crear registros mediante expresiones de registro

Puede inicializar los registros mediante las etiquetas definidas en el registro. Una expresión que hace esto se conoce como una *expresión de registro*. Use llaves para encerrar la expresión de registro y use el punto y coma como delimitador.

En el ejemplo siguiente se muestra cómo crear un registro.

```
type MyRecord =  
  { X: int  
    Y: int  
    Z: int }  
  
let myRecord1 = { X = 1; Y = 2; Z = 3; }
```

Los signos de punto y coma después del último campo de la expresión de registro y de la definición de tipo son opcionales, independientemente de si todos los campos están en una sola línea.

Al crear un registro, debe proporcionar valores para cada campo. No se puede hacer referencia a los valores de otros campos de la expresión de inicialización de ningún campo.

En el código siguiente, el tipo de `myRecord2` se deduce de los nombres de los campos. Opcionalmente, puede especificar el nombre del tipo explícitamente.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

Otra forma de construcción de registros puede ser útil si tiene que copiar un registro existente y, posiblemente, cambiar algunos de los valores de los campos. La siguiente línea de código ilustra esto.

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

Esta forma de la expresión de registro se denomina *expresión de registro de copia y actualización*.

Los registros son inmutables de forma predeterminada; sin embargo, puede crear fácilmente registros modificados mediante una expresión de copia y actualización. También puede especificar explícitamente un campo mutable.

```

type Car =
  { Make : string
    Model : string
    mutable Odometer : int }

let myCar = { Make = "Fabrikam"; Model = "Coupe"; Odometer = 108112 }
myCar.Odometer <- myCar.Odometer + 21

```

No use el atributo `DefaultValue` con campos de registro. Un mejor enfoque es definir instancias predeterminadas de registros con campos que se inicializan con valores predeterminados y, a continuación, usar una expresión de registro de copia y actualización para establecer los campos que difieren de los valores predeterminados.

```

// Rather than use [<DefaultValue>], define a default record.
type MyRecord =
  { Field1 : int
    Field2 : int }

let defaultRecord1 = { Field1 = 0; Field2 = 0 }
let defaultRecord2 = { Field1 = 1; Field2 = 25 }

// Use the with keyword to populate only a few chosen fields
// and leave the rest with default values.
let rr3 = { defaultRecord1 with Field2 = 42 }

```

Crear registros recursivos mutuamente

En algún momento al crear un registro, puede que desee que dependa de otro tipo que le gustaría definir después. Se trata de un error de compilación a menos que defina los tipos de registro para que sean recursivos mutuamente.

La definición de registros recursivos mutuamente se realiza `and` con la palabra clave. Esto le permite vincular 2 o más tipos de registro juntos.

Por ejemplo, el código siguiente define un `Person` tipo `Address` y como recursivo mutuamente:

```

// Create a Person type and use the Address type that is not defined
type Person =
  { Name: string
    Age: int
    Address: Address }
// Define the Address type which is used in the Person record
and Address =
  { Line1: string
    Line2: string
    PostCode: string
    Occupant: Person }

```

Si fuera a definir el ejemplo anterior sin la `and` palabra clave, no se compilaría. La `and` palabra clave es necesaria para las definiciones mutuamente recursivas.

Coincidencia de patrones con registros

Los registros se pueden usar con la coincidencia de patrones. Puede especificar algunos campos explícitamente y proporcionar variables para otros campos que se asignarán cuando se produzca una coincidencia. En el siguiente ejemplo código se muestra cómo hacerlo.

```

type Point3D = { X: float; Y: float; Z: float }
let evaluatePoint (point: Point3D) =
    match point with
    | { X = 0.0; Y = 0.0; Z = 0.0 } -> printfn "Point is at the origin."
    | { X = xVal; Y = 0.0; Z = 0.0 } -> printfn "Point is on the x-axis. Value is %f." xVal
    | { X = 0.0; Y = yVal; Z = 0.0 } -> printfn "Point is on the y-axis. Value is %f." yVal
    | { X = 0.0; Y = 0.0; Z = zVal } -> printfn "Point is on the z-axis. Value is %f." zVal
    | { X = xVal; Y = yVal; Z = zVal } -> printfn "Point is at (%f, %f, %f)." xVal yVal zVal

evaluatePoint { X = 0.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 100.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 10.0; Y = 0.0; Z = -1.0 }

```

El resultado de este código es el siguiente.

```

Point is at the origin.
Point is on the x-axis. Value is 100.000000.
Point is at (10.000000, 0.000000, -1.000000).

```

Diferencias entre los registros y las clases

Los campos de registro difieren de las clases en que se exponen automáticamente como propiedades y se usan en la creación y copia de registros. La construcción de registros también difiere de la construcción de clases. En un tipo de registro, no se puede definir un constructor. En su lugar, se aplica la sintaxis de construcción que se describe en este tema. Las clases no tienen ninguna relación directa entre los parámetros de constructor, los campos y las propiedades.

Al igual que los tipos de estructura y Unión, los registros tienen una semántica estructural de igualdad. Las clases tienen semántica de igualdad de referencia. El siguiente ejemplo de código muestra esto.

```

type RecordTest = { X: int; Y: int }

let record1 = { X = 1; Y = 2 }
let record2 = { X = 1; Y = 2 }

if (record1 = record2) then
    printfn "The records are equal."
else
    printfn "The records are unequal."

```

El resultado de este código es el siguiente:

```

The records are equal.

```

Si escribe el mismo código con clases, los dos objetos de clase no serían iguales porque los dos valores representarían dos objetos en el montón y solo se compararían las direcciones (a menos que el tipo de `System.Object.Equals` clase invalide el método).

Si necesita igualdad de referencia para los registros, agregue el `[<ReferenceEquality>]` atributo encima del registro.

Vea también

- [Tipos en F#](#)
- [Clases](#)

- Referencia del lenguaje F#
- Igualdad de referencia
- Coincidencia de patrones

Registros anónimos

21/04/2020 • 12 minutes to read • [Edit Online](#)

Los registros anónimos son agregados simples de valores con nombre que no es necesario declarar antes de su uso. Puede declararlos como structs o tipos de referencia. Son tipos de referencia de forma predeterminada.

Sintaxis

En los ejemplos siguientes se muestra la sintaxis de registro anónimo. Elementos delimitados como `[item]` opcionales.

```
// Construct an anonymous record
let value-name = [struct] {| Label1: Type1; Label2: Type2; ...|}

// Use an anonymous record as a type parameter
let value-name = Type-Name<[struct] {| Label1: Type1; Label2: Type2; ...|}>

// Define a parameter with an anonymous record as input
let function-name (arg-name: [struct] {| Label1: Type1; Label2: Type2; ...|}) ...
```

Uso básico

Los registros anónimos se consideran mejor como tipos de registros de F- que no es necesario declarar antes de la creación de instancias.

Por ejemplo, aquí se puede interactuar con una función que genera un registro anónimo:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

En el ejemplo siguiente se expande el anterior con una `printCircleStats` función que toma un registro anónimo como entrada:

```

open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let printCircleStats r (stats: {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats

```

Llamar `printCircleStats` con cualquier tipo de registro anónimo que no tenga la misma "forma" que el tipo de entrada no se compilará:

```

printCircleStats r {| Diameter = 2.0; Area = 4.0; MyCircumference = 12.566371 |}
// Two anonymous record types have mismatched sets of field names
// '["Area"; "Circumference"; "Diameter"]' and '["Area"; "Diameter"; "MyCircumference"]'

```

Estructurar registros anónimos

Los registros anónimos también se `struct` pueden definir como struct con la palabra clave opcional. En el ejemplo siguiente se aumenta el anterior mediante la producción y el consumo de un registro anónimo struct:

```

open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    // Note that the keyword comes before the '{| |}' brace pair
    struct {| Area = a; Circumference = c; Diameter = d |}

// the 'struct' keyword also comes before the '{| |}' brace pair when declaring the parameter type
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats

```

Inferencia de estructuración

Los registros anónimos de estructura también permiten la "inferencia de estructuración" donde no es necesario especificar la `struct` palabra clave en el sitio de llamada. En este ejemplo, se `struct` elimina la `printCircleStats` palabra clave al llamar a:

```
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

printCircleStats r {| Area = 4.0; Circumference = 12.6; Diameter = 12.6 |}
```

El patrón inverso, que especifica `struct` cuándo el tipo de entrada no es un registro anónimo `struct`, no se compilará.

Incrustar registros anónimos dentro de otros tipos

Es útil declarar [uniones discriminadas](#) cuyos casos son registros. Pero si los datos de los registros son del mismo tipo que la unión discriminada, debe definir todos los tipos como mutuamente recursivos. El uso de registros anónimos evita esta restricción. Lo que sigue es un tipo de ejemplo y una función que el patrón coincide sobre él:

```
type FullName = { FirstName: string; LastName: string }

// Note that using a named record for Manager and Executive would require mutually recursive definitions.
type Employee =
    | Engineer of FullName
    | Manager of {| Name: FullName; Reports: Employee list |}
    | Executive of {| Name: FullName; Reports: Employee list; Assistant: Employee |}

let getFirstName e =
    match e with
    | Engineer fullName -> fullName.FirstName
    | Manager m -> m.Name.FirstName
    | Executive ex -> ex.Name.FirstName
```

Copiar y actualizar expresiones

Los registros anónimos admiten la construcción con expresiones de [copia y actualización](#). Por ejemplo, a continuación se muestra cómo puede construir una nueva instancia de un registro anónimo que copia los datos de uno existente:

```
let data = {| X = 1; Y = 2 |}
let data' = {| data with Y = 3 |}
```

Sin embargo, a diferencia de los registros con nombre, los registros anónimos le permiten construir formularios completamente diferentes con expresiones de copia y actualización. En el ejemplo siguiente se toma el mismo registro anónimo del ejemplo anterior y se expande en un nuevo registro anónimo:

```
let data = {| X = 1; Y = 2 |}
let expandedData = {| data with Z = 3 |} // Gives {| X=1; Y=2; Z=3 |}
```

También es posible construir registros anónimos a partir de instancias de registros con nombre:

```
type R = { X: int }
let data = { X = 1 }
let data' = {| data with Y = 2 |} // Gives {| X=1; Y=2 |}
```

También puede copiar datos a y desde registros anónimos de referencia y estructuración:

```
// Copy data from a reference record into a struct anonymous record
type R1 = { X: int }
let r1 = { X = 1 }

let data1 = struct { | r1 with Y = 1 | }

// Copy data from a struct record into a reference anonymous record
[<Struct>]
type R2 = { X: int }
let r2 = { X = 1 }

let data2 = { | r1 with Y = 1 | }

// Copy the reference anonymous record data into a struct anonymous record
let data3 = struct { | data2 with Z = r2.X | }
```

Propiedades de registros anónimos

Los registros anónimos tienen una serie de características que son esenciales para comprender completamente cómo se pueden usar.

Los registros anónimos son nominales

Los registros anónimos son [tipos nominales](#). Se piensan mejor como tipos de [registro](#) con nombre (que también son nominales) que no requieren una declaración inicial.

Considere el siguiente ejemplo con dos declaraciones de registros anónimos:

```
let x = { | X = 1 | }
let y = { | Y = 1 | }
```

Los `x` y `y` valores y tienen diferentes tipos y no son compatibles entre sí. No son ecuatables y no son comparables. Para ilustrar esto, considere un equivalente de registro con nombre:

```
type X = { X: int }
type Y = { Y: int }

let x = { X = 1 }
let y = { Y = 1 }
```

No hay nada inherentemente diferente acerca de los registros anónimos en comparación con sus equivalentes de registro con su nombre cuando se refiere a equivalencia de tipo o comparación.

Los registros anónimos utilizan la igualdad estructural y la comparación

Al igual que los tipos de registros, los registros anónimos son estructuralmente igualables y comparables. Esto solo es cierto si todos los tipos constituyentes admiten la igualdad y la comparación, como con los tipos de registro. Para admitir la igualdad o la comparación, dos registros anónimos deben tener la misma "forma".

```
{ | a = 1+1 | } = { | a = 2 | } // true
{ | a = 1+1 | } > { | a = 1 | } // true

// error FS0001: Two anonymous record types have mismatched sets of field names '["a"]' and '["a"; "b"]'
{ | a = 1 + 1 | } = { | a = 2; b = 1 | }
```

Los registros anónimos son serializables

Puede serializar registros anónimos del igual que con los registros con nombre. Aquí está un ejemplo usando [Newtonsoft.Json](#):

```
open Newtonsoft.Json

let philip' = {| name="Phillip"; age=28 |}
let philStr = JsonConvert.SerializeObject(philip')

let philip = JsonConvert.DeserializeObject<{|name: string; age: int|}>(philStr)
printfn "Name: %s Age: %d" philip.name philip.age
```

Los registros anónimos son útiles para enviar datos ligeros a través de una red sin necesidad de definir un dominio para los tipos serializados/deserializados por adelantado.

Los registros anónimos interoperan con los tipos anónimos de CTM

Es posible usar una API de .NET que requiera el uso de [tipos anónimos](#) de C. Los tipos anónimos de CTM son triviales de interoperar mediante el uso de registros anónimos. En el ejemplo siguiente se muestra cómo usar registros anónimos para llamar a una sobrecarga [LINQ](#) que requiere un tipo anónimo:

```
open System.Linq

let names = [ "Ana"; "Felipe"; "Emilia" ]
let nameGrouping = names.Select(fun n -> {| Name = n; FirstLetter = n.[0] |})
for ng in nameGrouping do
    printfn "%s has first letter %c" ng.Name ng.FirstLetter
```

Hay una multitud de otras API utilizadas en .NET que requieren el uso de pasar un tipo anónimo. Los registros anónimos son su herramienta para trabajar con ellos.

Limitaciones

Los registros anónimos tienen algunas restricciones en su uso. Algunos son inherentes a su diseño, pero otros son susceptibles de cambiar.

Limitaciones con coincidencia de patrones

Los registros anónimos no admiten la coincidencia de patrones, a diferencia de los registros con nombre. Hay tres razones:

1. Un patrón tendría que tener en cuenta cada campo de un registro anónimo, a diferencia de los tipos de registro con nombre. Esto se debe a que los registros anónimos no admiten la subtipación estructural: son tipos nominales.
2. Debido a (1), no hay capacidad para tener patrones adicionales en una expresión de coincidencia de patrón, ya que cada patrón distinto implicaría un tipo de registro anónimo diferente.
3. Debido a (3), cualquier patrón de registro anónimo sería más detallado que el uso de la notación "punto".

Hay una sugerencia de lenguaje abierto para permitir la coincidencia de [patrones en contextos limitados](#).

Limitaciones con mutabilidad

Actualmente no es posible definir `mutable` un registro anónimo con datos. Hay una sugerencia de [lenguaje abierto](#) para permitir datos mutables.

Limitaciones con registros anónimos de estructura

No es posible declarar registros `IsByRefLike` `IsReadOnly` anónimos struct como o . Hay una sugerencia de `IsByRefLike` lenguaje `IsReadOnly` [abierto](#) para y registros anónimos.

Expresiones de registro de copia y actualización

21/04/2020 • 2 minutes to read • [Edit Online](#)

Una expresión de registro de *copia y actualización* es una expresión que copia un registro existente, actualiza los campos especificados y devuelve el registro actualizado.

Sintaxis

```
{ record-name with  
  updated-labels }  
  
{| anonymous-record-name with  
  updated-labels |}
```

Observaciones

Los registros y registros anónimos son inmutables de forma predeterminada, por lo que no es posible actualizar un registro existente. Para crear un registro actualizado, todos los campos de un registro tendrían que especificarse de nuevo. Para simplificar esta tarea se puede utilizar una expresión de *copia y actualización*. Esta expresión toma un registro existente, crea uno nuevo del mismo tipo mediante el uso de campos especificados de la expresión y el campo que falta especificado por la expresión.

Esto puede ser útil cuando tiene que copiar un registro existente y, posiblemente, cambiar algunos de los valores de campo.

Tomemos por ejemplo un registro recién creado.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

Para actualizar solo dos campos de ese registro, puede utilizar la expresión de *registro de copia y actualización*:

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

Consulte también

- [Registros](#)
- [Registros anónimos](#)
- [Referencia del lenguaje f](#)

Uniones discriminadas

19/03/2020 • 15 minutes to read • [Edit Online](#)

Las uniones discriminadas proporcionan compatibilidad con valores que pueden ser uno de una serie de casos con nombre, posiblemente cada uno con valores y tipos diferentes. Las uniones discriminadas son útiles para datos heterogéneos; datos que pueden tener casos especiales, incluidos los casos válidos y de error; datos que varían en tipo de una instancia a otra; y como alternativa para jerarquías de objetos pequeños. Además, las uniones discriminadas recursivas se utilizan para representar estructuras de datos de árbol.

Sintaxis

```
[ attributes ]
type [accessibility-modifier] type-name =
  | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ] type2 ...]
  | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4 ...]

[ member-list ]
```

Observaciones

Las uniones discriminadas son similares a los tipos de unión en otros idiomas, pero hay diferencias. Al igual que con un tipo de unión en C++ o un tipo de variante en Visual Basic, los datos almacenados en el valor no son fijos; puede ser una de varias opciones distintas. A diferencia de las uniones en estos otros idiomas, sin embargo, cada una de las opciones posibles recibe un identificador de *caso*. Los identificadores de caso son nombres para los distintos tipos posibles de valores que podrían ser los objetos de este tipo; los valores son opcionales. Si los valores no están presentes, el caso es equivalente a un caso de enumeración. Si hay valores presentes, cada valor puede ser un único valor de un tipo especificado o una tupla que agrega varios campos de los mismos tipos o diferentes. Puede asignar un nombre a un campo individual, pero el nombre es opcional, incluso si se nombran otros campos en el mismo caso.

La accesibilidad de las uniones `public` discriminadas tiene como valor predeterminado .

Por ejemplo, considere la siguiente declaración de un Shape tipo.

```
type Shape =
  | Rectangle of width : float * length : float
  | Circle of radius : float
  | Prism of width : float * float * height : float
```

El código anterior declara una unión discriminada Shape, que puede tener valores de cualquiera de los tres casos: Rectangle, Circle y Prism. Cada caso tiene un conjunto diferente de campos. El caso Rectangle tiene dos campos `float` con nombre, ambos de tipo , que tienen los nombres width y length. El caso Círculo tiene un solo campo con nombre, radio. El caso Prism tiene tres campos, dos de los cuales (ancho y alto) se denominan campos. Los campos sin nombre se conocen como campos anónimos.

Los objetos se construyen proporcionando valores para los campos con nombre y anónimos según los ejemplos siguientes.

```
let rect = Rectangle(length = 1.3, width = 10.0)
let circ = Circle (1.0)
let prism = Prism(5., 2.0, height = 3.0)
```

Este código muestra que puede usar los campos con nombre en la inicialización o puede confiar en el orden de los campos de la declaración y simplemente proporcionar los valores para cada campo a su vez. La llamada `rect` del constructor en el código anterior usa `circ` los campos con nombre, pero la llamada del constructor para usa el orden. Puede mezclar los campos ordenados y los `prism` campos con nombre, como en la construcción de .

El `option` tipo es una unión discriminada simple en la biblioteca principal de F . El `option` tipo se declara como sigue.

```
// The option type is a discriminated union.
type Option<'a> =
    | Some of 'a
    | None
```

El código anterior especifica `option` que el tipo es una `Some` unión discriminada que tiene dos casos y `None` . El `Some` caso tiene un valor asociado que consta de un campo `'a` anónimo cuyo tipo está representado por el parámetro `type` . El `None` caso no tiene ningún valor asociado. Por `option` lo tanto, el tipo especifica un tipo genérico que tiene un valor de algún tipo o ningún valor. El `option` tipo también tiene un `option` alias de tipo en minúsculas, `option`, que se utiliza más comúnmente.

Los identificadores de caso se pueden usar como constructores para el tipo de unión discriminada. Por ejemplo, el código siguiente se `option` utiliza para crear valores del tipo.

```
let myOption1 = Some(10.0)
let myOption2 = Some("string")
let myOption3 = None
```

Los identificadores de mayúsculas y minúsculas también se utilizan en expresiones de coincidencia de patrones. En una expresión de coincidencia de patrones, se proporcionan identificadores para los valores asociados con los casos individuales. Por ejemplo, en el `x` código siguiente, es el identificador `Some` dado `option` el valor que está asociado con el caso del tipo.

```
let printValue opt =
    match opt with
    | Some x -> printfn "%A" x
    | None -> printfn "No value."
```

En las expresiones de coincidencia de patrones, puede usar campos con nombre para especificar coincidencias de unión discriminadas. Para el tipo `Shape` que se declaró anteriormente, puede usar los campos con nombre como se muestra en el código siguiente para extraer los valores de los campos.

```
let getShapeWidth shape =
    match shape with
    | Rectangle(width = w) -> w
    | Circle(radius = r) -> 2. * r
    | Prism(width = w) -> w
```

Normalmente, los identificadores de caso se pueden utilizar sin calificarlos con el nombre de la unión. Si desea que el nombre siempre se califique con el nombre de la unión, puede aplicar el atributo

[RequireQualifiedAccess](#) a la definición de tipo de unión.

Desenvolviendo uniones discriminadas

En las Uniones Discriminadas de F- se usan a menudo en el modelado de dominios para ajustar un solo tipo. También es fácil extraer el valor subyacente a través de la coincidencia de patrones. No es necesario utilizar una expresión de coincidencia para un solo caso:

```
let ([UnionCaseIdentifier] [values]) = [UnionValue]
```

En el siguiente ejemplo se muestra esto:

```
type ShaderProgram = | ShaderProgram of id:int

let someFunctionUsingShaderProgram shaderProgram =
    let (ShaderProgram id) = shaderProgram
    // Use the unwrapped value
    ...
```

La coincidencia de patrones también se permite directamente en los parámetros de función, por lo que puede desencapsular un solo caso allí:

```
let someFunctionUsingShaderProgram (ShaderProgram id) =
    // Use the unwrapped value
    ...
```

Estructurar uniones discriminadas

También puede representar Uniones Discriminadas como estructuras. Esto se hace `[<Struct>]` con el atributo.

```
[<Struct>]
type SingleCase = Case of string

[<Struct>]
type Multicase =
    | Case1 of Case1 : string
    | Case2 of Case2 : int
    | Case3 of Case3 : double
```

Dado que se trata de tipos de valor y no de tipos de referencia, hay consideraciones adicionales en comparación con las uniones discriminadas de referencia:

1. Se copian como tipos de valor y tienen semántica de tipo de valor.
2. No se puede utilizar una definición de tipo recursiva con una unión discriminada de estructura multicaso.
3. Debe proporcionar nombres de caso únicos para una unión discriminada de estructura multicaso.

Uso de uniones discriminadas en lugar de jerarquías de objetos

A menudo puede utilizar una unión discriminada como una alternativa más sencilla a una jerarquía de objetos pequeños. Por ejemplo, la siguiente unión discriminada `Shape` podría utilizarse en lugar de una clase base que tenga tipos derivados para circle, square, etc.

```

type Shape =
    // The value here is the radius.
    | Circle of float
    // The value here is the side length.
    | EquilateralTriangle of double
    // The value here is the side length.
    | Square of double
    // The values here are the height and width.
    | Rectangle of double * double

```

En lugar de un método virtual para calcular un área o perímetro, como usaría en una implementación orientada a objetos, puede usar la coincidencia de patrones para bifurcar las fórmulas adecuadas para calcular estas cantidades. En el ejemplo siguiente, se utilizan diferentes fórmulas para calcular el área, dependiendo de la forma.

```

let pi = 3.141592654

let area myShape =
    match myShape with
    | Circle radius -> pi * radius * radius
    | EquilateralTriangle s -> (sqrt 3.0) / 4.0 * s * s
    | Square s -> s * s
    | Rectangle (h, w) -> h * w

let radius = 15.0
let myCircle = Circle(radius)
printfn "Area of circle that has radius %f: %f" radius (area myCircle)

let squareSide = 10.0
let mySquare = Square(squareSide)
printfn "Area of square that has side %f: %f" squareSide (area mySquare)

let height, width = 5.0, 10.0
let myRectangle = Rectangle(height, width)
printfn "Area of rectangle that has height %f and width %f is %f" height width (area myRectangle)

```

La salida es como sigue:

```

Area of circle that has radius 15.000000: 706.858347
Area of square that has side 10.000000: 100.000000
Area of rectangle that has height 5.000000 and width 10.000000 is 50.000000

```

Uso de uniones discriminadas para estructuras de datos de árboles

Las uniones discriminadas pueden ser recursivas, lo que significa que la propia unión puede incluirse en el tipo de uno o más casos. Las uniones discriminadas recursivas se pueden utilizar para crear estructuras de árbol, que se utilizan para modelar expresiones en lenguajes de programación. En el código siguiente, se usa una unión discriminada recursiva para crear una estructura de datos de árbol binario. La unión consta `Node` de dos casos, , que es un nodo con `Tip` un valor entero y subárboles izquierdo y derecho, y , que termina el árbol.

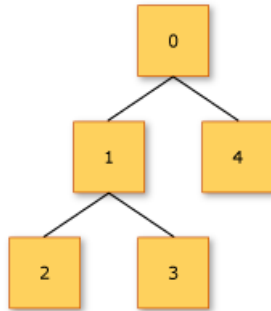
```

type Tree =
  | Tip
  | Node of int * Tree * Tree

let rec sumTree tree =
  match tree with
  | Tip -> 0
  | Node(value, left, right) ->
    value + sumTree(left) + sumTree(right)
let myTree = Node(0, Node(1, Node(2, Tip, Tip), Node(3, Tip, Tip)), Node(4, Tip, Tip))
let resultSumTree = sumTree myTree

```

En el código `resultSumTree` anterior, tiene el valor 10. En la ilustración siguiente `myTree` se muestra la estructura de árbol para .



Las uniones discriminadas funcionan bien si los nodos del árbol son heterogéneos. En el código siguiente, el tipo `Expression` representa el árbol de sintaxis abstracta de una expresión en un lenguaje de programación simple que admite la adición y multiplicación de números y variables. Algunos de los casos de unión no `Number` son recursivos y representan números () o variables (`Variable`). Otros casos son recursivos `Add` y `Multiply` representan operaciones (y), donde los operandos también son expresiones. La `Evaluate` función utiliza una expresión de coincidencia para procesar recursivamente el árbol de sintaxis.

```

type Expression =
  | Number of int
  | Add of Expression * Expression
  | Multiply of Expression * Expression
  | Variable of string

let rec Evaluate (env:Map<string,int>) exp =
  match exp with
  | Number n -> n
  | Add (x, y) -> Evaluate env x + Evaluate env y
  | Multiply (x, y) -> Evaluate env x * Evaluate env y
  | Variable id -> env.[id]

let environment = Map.ofList [ "a", 1 ;
                              "b", 2 ;
                              "c", 3 ]

// Create an expression tree that represents
// the expression: a + 2 * b.
let expressionTree1 = Add(Variable "a", Multiply(Number 2, Variable "b"))

// Evaluate the expression a + 2 * b, given the
// table of values for the variables.
let result = Evaluate environment expressionTree1

```

Cuando se ejecuta este código, el valor de `result` es 5.

Members

Es posible definir miembros en uniones discriminadas. En el ejemplo siguiente se muestra cómo definir una propiedad e implementar una interfaz:

```
open System

type IPrintable =
    abstract Print: unit -> unit

type Shape =
    | Circle of float
    | EquilateralTriangle of float
    | Square of float
    | Rectangle of float * float

    member this.Area =
        match this with
        | Circle r -> 2.0 * Math.PI * r
        | EquilateralTriangle s -> s * s * sqrt 3.0 / 4.0
        | Square s -> s * s
        | Rectangle(l, w) -> l * w

    interface IPrintable with
        member this.Print () =
            match this with
            | Circle r -> printfn "Circle with radius %f" r
            | EquilateralTriangle s -> printfn "Equilateral Triangle of side %f" s
            | Square s -> printfn "Square with side %f" s
            | Rectangle(l, w) -> printfn "Rectangle with length %f and width %f" l w
```

Atributos comunes

Los siguientes atributos se ven comúnmente en uniones discriminadas:

- [`<RequireQualifiedAccess>`]
- [`<NoEquality>`]
- [`<NoComparison>`]
- [`<Struct>`]

Consulte también

- [Referencia del lenguaje f](#)

Enumeraciones

23/10/2019 • 4 minutes to read • [Edit Online](#)

Las enumeraciones, también conocidas como enumeraciones, son tipos enteros donde las etiquetas se asignan a un subconjunto de los valores. Se pueden usar en lugar de los literales para que el código sea más fácil de leer y mantener.

Sintaxis

```
type enum-name =  
| value1 = integer-literal1  
| value2 = integer-literal2  
...
```

Comentarios

Una enumeración es muy similar a una Unión discriminada que tiene valores simples, con la excepción de que se pueden especificar los valores. Los valores suelen ser enteros que comienzan en 0 o 1, o enteros que representan posiciones de bits. Si una enumeración está pensada para representar posiciones de bits, debe usar también el atributo [Flags](#).

El tipo subyacente de la enumeración se determina a partir del literal que se utiliza, de modo que, por ejemplo, puede usar literales con un sufijo, `1u` como `2u`, etc., para un tipo entero (`uint32`) sin signo.

Cuando se hace referencia a los valores con nombre, debe usar el nombre del tipo de enumeración como calificador, es decir `enum-name.value1`, no `value1` solo. Este comportamiento difiere del de las uniones discriminadas. Esto se debe a que las enumeraciones siempre tienen el atributo [RequireQualifiedAccess](#).

En el código siguiente se muestra la declaración y el uso de una enumeración.

```
// Declaration of an enumeration.  
type Color =  
| Red = 0  
| Green = 1  
| Blue = 2  
// Use of an enumeration.  
let col1 : Color = Color.Red
```

Puede convertir fácilmente las enumeraciones en el tipo subyacente mediante el operador adecuado, tal y como se muestra en el código siguiente.

```
// Conversion to an integral type.  
let n = int col1
```

Los tipos enumerados pueden tener uno de los siguientes tipos subyacentes `byte`, `int16`, `sbyte`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `char` y. Los tipos de enumeración se representan en el .NET Framework como tipos que `System.Enum` se heredan de, que a `System.ValueType` su vez se hereda de. Por lo tanto, son tipos de valor que se encuentran en la pila o insertados en el objeto contenedor, y cualquier valor del tipo subyacente es un valor válido de la enumeración. Esto es importante cuando la coincidencia de patrones en los valores de enumeración, porque tiene que proporcionar un patrón que detecte los valores sin nombre.

La `enum` función de la F# biblioteca se puede usar para generar un valor de enumeración, incluso un valor distinto de uno de los valores con nombre predefinidos. La `enum` función se usa como se indica a continuación.

```
let col2 = enum<Color>(3)
```

La función `enum` predeterminada funciona con el `int32` tipo. Por lo tanto, no se puede utilizar con tipos de enumeración que tienen otros tipos subyacentes. En su lugar, use lo siguiente.

```
type uColor =  
    | Red = 0u  
    | Green = 1u  
    | Blue = 2u  
let col3 = Microsoft.FSharp.Core.LanguagePrimitives.EnumOfValue<uint32, uColor>(2u)
```

Además, los casos de enumeración siempre se emiten como `public`. Esto es para que se alineen C# con y el resto de la plataforma .net.

Vea también

- [Referencia del lenguaje F#](#)
- [Conversiones](#)

Abreviaturas de tipo

23/10/2019 • 2 minutes to read • [Edit Online](#)

Una *abreviatura de tipo* es un alias o un nombre alternativo para un tipo.

Sintaxis

```
type [accessibility-modifier] type-abbreviation = type-name
```

Comentarios

Puede usar las abreviaturas de tipo para asignar un nombre más significativo a un tipo, con el fin de facilitar la lectura del código. También puede usarlos para crear un nombre fácil de usar para un tipo que, de otro modo, es engorroso de escribir. Además, puede usar las abreviaturas de tipo para facilitar la modificación de un tipo subyacente sin cambiar todo el código que usa el tipo. La siguiente es una abreviatura de tipo simple.

La accesibilidad de las abreviaturas de tipo `public` tiene como valor predeterminado.

```
type SizeType = uint32
```

Las abreviaturas de tipo pueden incluir parámetros genéricos, como en el código siguiente.

```
type Transform<'a> = 'a -> 'a
```

En el código anterior, `Transform` es una abreviatura de tipo que representa una función que toma un solo argumento de cualquier tipo y que devuelve un valor único del mismo tipo.

Las abreviaturas de tipo no se conservan en el código MSIL de .NET Framework. Por lo tanto, cuando se F# usa un ensamblado de otro lenguaje .NET Framework, se debe usar el nombre de tipo subyacente para una abreviatura de tipo.

Las abreviaturas de tipo también se pueden usar en unidades de medida. Para obtener más información, consulte [unidades de medida](#).

Vea también

- [Referencia del lenguaje F#](#)

Clases

23/10/2019 • 16 minutes to read • [Edit Online](#)

Las *clases* son tipos que representan objetos que pueden tener propiedades, métodos y eventos.

Sintaxis

```
// Class definition:
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =
[ class ]
[ inherit base-type-name(base-constructor-args) ]
[ let-bindings ]
[ do-bindings ]
member-list
...
[ end ]
// Mutually recursive class definitions:
type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

Comentarios

Las clases representan la descripción fundamental de los tipos de objeto .NET; la clase es el concepto de tipo principal que admite la programación orientada F# a objetos en.

En la sintaxis anterior, `type-name` es cualquier identificador válido. El `type-params` describe los parámetros de tipo genérico opcionales. Consta de los nombres de parámetro de tipo y las restricciones entre corchetes angulares `<` (`>` y). Para obtener más información, vea [Genéricos](#) y [Restricciones](#). Describe `parameter-list` los parámetros de constructor. El primer modificador de acceso pertenece al tipo; el segundo pertenece al constructor principal. En ambos casos, el valor predeterminado `public` es.

La clase base de una clase se especifica mediante la `inherit` palabra clave. Debe proporcionar argumentos, entre paréntesis, para el constructor de clase base.

Declare los campos o valores de función que son locales de la clase `let` mediante enlaces y debe seguir las reglas generales para `let` los enlaces. En `do-bindings` la sección se incluye el código que se va a ejecutar durante la construcción del objeto.

`member-list` Está formado por constructores adicionales, declaraciones de instancias y métodos estáticos, declaraciones de interfaz, enlaces abstractos y declaraciones de propiedades y eventos. Se describen en [miembros](#).

El `identifier` que se usa con la palabra `as` clave opcional proporciona un nombre a la variable de instancia, o Self Identifier, que se puede usar en la definición de tipo para hacer referencia a la instancia del tipo. Para obtener más información, vea la sección identificadores propios más adelante en este tema.

Las palabras `class` clave `end` y que marcan el inicio y el final de la definición son opcionales.

Los tipos mutuamente recursivos, que son tipos que hacen referencia entre sí, se unen junto `and` con la palabra clave, al igual que las funciones recursivas. Para obtener un ejemplo, vea la sección tipos mutuamente recursivos.

Constructores

El constructor es código que crea una instancia del tipo de clase. Los constructores de las clases funcionan de F# manera ligeramente diferente que en otros lenguajes .net. En una F# clase, siempre hay un constructor principal cuyos argumentos se describen en el `parameter-list` que sigue al nombre de tipo y cuyo cuerpo está compuesto de los `let` enlaces (y `let rec`) al principio de la declaración de clase y el `do` enlaces siguientes. Los argumentos del constructor principal están en el ámbito de la declaración de clase.

Puede Agregar constructores adicionales mediante la `new` palabra clave para agregar un miembro, como se indica a continuación:

```
new ( argument-list ) = constructor-body
```

El cuerpo del nuevo constructor debe invocar el constructor principal que se especifica en la parte superior de la declaración de clase.

En el ejemplo siguiente se muestra este concepto. En el código siguiente, `MyClass` tiene dos constructores, un constructor principal que toma dos argumentos y otro que no toma ningún argumento.

```
type MyClass1(x: int, y: int) =  
    do printfn "%d %d" x y  
    new() = MyClass1(0, 0)
```

permitir y realizar enlaces

Los `let` enlaces `do` y de una definición de clase forman el cuerpo del constructor de clase principal y, por lo tanto, se ejecutan cada vez que se crea una instancia de clase. Si un `let` enlace es una función, se compila en un miembro. Si el `let` enlace es un valor que no se usa en ninguna función o miembro, se compila en una variable local para el constructor. De lo contrario, se compila en un campo de la clase. Las `do` expresiones siguientes se compilan en el constructor principal y ejecutan el código de inicialización para cada instancia. Dado que los constructores adicionales siempre llaman al constructor principal, `let` los enlaces y `do` enlaces siempre se ejecutan independientemente del constructor al que se llame.

Se puede tener acceso a `let` los campos que se crean mediante enlaces a través de los métodos y propiedades de la clase; sin embargo, no se puede tener acceso a ellos desde métodos estáticos, incluso si los métodos estáticos toman una variable de instancia como parámetro. No se puede tener acceso a ellos mediante el identificador propio, si existe alguno.

Identificadores propios

Un *identificador propio* es un nombre que representa la instancia actual. Los identificadores propios se asemejan a `this` la palabra `Me` clave en C# o C++ o en Visual Basic. Puede definir un identificador propio de dos maneras diferentes, en función de si desea que el identificador propio esté en el ámbito de la definición de clase completa o solo para un método individual.

Para definir un identificador propio para toda la clase, utilice la `as` palabra clave después del paréntesis de cierre de la lista de parámetros del constructor y especifique el nombre del identificador.

Para definir un identificador propio solo para un método, proporcione el identificador propio en la declaración de miembro, justo antes del nombre del método y un punto (.) como separador.

En el ejemplo de código siguiente se muestran las dos maneras de crear un identificador propio. En la primera línea, la `as` palabra clave se usa para definir el identificador propio. En la quinta línea, el identificador `this` se usa para definir un identificador propio cuyo ámbito está restringido al método. `PrintMessage`

```
type MyClass2(dataIn) as self =
    let data = dataIn
    do
        self.PrintMessage()
    member this.PrintMessage() =
        printf "Creating MyClass2 with Data %d" data
```

A diferencia de otros lenguajes de .NET, puede asignar un nombre al propio identificador si lo desea; no está restringido a nombres como `self`, `Me` o `this`.

El identificador propio que se declara con la `as` palabra clave no se inicializa hasta que se ejecutan los `let` enlaces. Por lo tanto, no se puede usar `let` en los enlaces. Puede usar el identificador propio en la `do` sección de enlaces.

Parámetros de tipos genéricos

Los parámetros de tipo genérico se especifican entre corchetes angulares (`<` y `>`), en forma de comillas simples seguidos de un identificador. Varios parámetros de tipo genérico se separan mediante comas. El parámetro de tipo genérico está en el ámbito de la declaración. En el ejemplo de código siguiente se muestra cómo especificar parámetros de tipo genérico.

```
type MyGenericClass<'a> (x: 'a) =
    do printfn "%A" x
```

Los argumentos de tipo se deducen cuando se usa el tipo. En el código siguiente, el tipo deducido es una secuencia de tuplas.

```
let g1 = MyGenericClass( seq { for i in 1 .. 10 -> (i, i*i) } )
```

Especificar herencia

La `inherit` cláusula identifica la clase base directa, si hay alguna. En F#, solo se permite una clase base directa. Las interfaces que implementa una clase no se consideran clases base. Las interfaces se describen en el tema [interfaces](#).

Puede tener acceso a los métodos y propiedades de la clase base desde la clase derivada mediante el uso de `base` la palabra clave Language como identificador, seguido de un punto (.) y el nombre del miembro.

Para obtener más información, vea [Herencia](#).

Sección de miembros

Puede definir métodos estáticos o de instancia, propiedades, implementaciones de interfaz, miembros abstractos, declaraciones de eventos y constructores adicionales en esta sección. Los enlaces `Let` y `do` no pueden aparecer en esta sección. Dado que los miembros se pueden agregar a una F# variedad de tipos además de clases, se tratan en un tema independiente, [miembros](#).

Tipos recursivos mutuamente

Al definir tipos que se hacen referencia entre sí de forma circular, se concatenan las definiciones de tipos mediante la `and` palabra clave. La `and` palabra clave reemplaza `type` la palabra clave en todas las definiciones excepto la primera, como se indica a continuación.

```

open System.IO

type Folder(pathIn: string) =
    let path = pathIn
    let filenameArray : string array = Directory.GetFiles(path)
    member this.FileArray = Array.map (fun elem -> new File(elem, this)) filenameArray

and File(filename: string, containingFolder: Folder) =
    member this.Name = filename
    member this.ContainingFolder = containingFolder

let folder1 = new Folder(".")
for file in folder1.FileArray do
    printfn "%s" file.Name

```

La salida es una lista de todos los archivos del directorio actual.

Cuándo usar clases, uniones, registros y estructuras

Dada la variedad de tipos entre los que elegir, debe tener una buena comprensión de para qué se ha diseñado cada tipo para seleccionar el tipo adecuado para una situación determinada. Las clases están diseñadas para su uso en contextos de programación orientados a objetos. La programación orientada a objetos es el paradigma dominante utilizado en las aplicaciones escritas para el .NET Framework. Si el F# código tiene que trabajar estrechamente con el .NET Framework u otra biblioteca orientada a objetos, y especialmente si tiene que extender desde un sistema de tipos orientado a objetos como una biblioteca de interfaz de usuario, las clases probablemente sean adecuadas.

Si no está interoperando estrechamente con código orientado a objetos, o si está escribiendo código que es independiente y, por tanto, está protegido contra la interacción frecuente con código orientado a objetos, considere la posibilidad de utilizar registros y uniones discriminadas. A menudo, se puede usar una Unión discriminada única y bien pensada, junto con el código de coincidencia de patrones adecuado, como alternativa más sencilla a una jerarquía de objetos. Para obtener más información sobre las uniones discriminadas, consulte [uniones discriminadas](#).

Los registros tienen la ventaja de ser más sencillos que las clases, pero los registros no son adecuados cuando las demandas de un tipo superan lo que se puede lograr con su simplicidad. Los registros son básicamente agregados de valores sencillos, sin constructores independientes que pueden realizar acciones personalizadas, sin campos ocultos y sin las implementaciones de herencia o de interfaz. Aunque se pueden agregar miembros como propiedades y métodos a los registros para que su comportamiento sea más complejo, los campos almacenados en un registro siguen siendo un agregado simple de valores. Para obtener más información sobre los registros, consulte [registros](#).

Las estructuras también son útiles para los pequeños agregados de datos, pero difieren de las clases y los registros en que son tipos de valor de .NET. Las clases y los registros son tipos de referencia de .NET. La semántica de los tipos de valor y los tipos de referencia es diferente en que los tipos de valor se pasan por valor. Esto significa que se copian bit de bit cuando se pasan como un parámetro o se devuelven desde una función. También se almacenan en la pila o, si se usan como un campo, se insertan dentro del objeto primario en lugar de almacenarse en su propia ubicación independiente en el montón. Por lo tanto, las estructuras son adecuadas para los datos a los que se accede con frecuencia cuando la sobrecarga de tener acceso al montón es un problema. Para obtener más información sobre las estructuras, vea [estructuras](#).

Vea también

- [Referencia del lenguaje F#](#)
- [Miembros](#)
- [Herencia](#)

- [Interfaces](#)

Estructuras

19/03/2020 • 7 minutes to read • [Edit Online](#)

Una *estructura* es un tipo de objeto compacto que puede ser más eficaz que una clase para los tipos que tienen una pequeña cantidad de datos y un comportamiento simple.

Sintaxis

```
[ attributes ]
type [accessibility-modifier] type-name =
    struct
        type-definition-elements-and-members
    end
// or
[ attributes ]
[<StructAttribute>]
type [accessibility-modifier] type-name =
    type-definition-elements-and-members
```

Observaciones

Las estructuras son tipos de *valor*; lo que significa que se almacenan directamente en la pila o, cuando se utilizan como campos o elementos de matriz, en línea en el tipo primario. A diferencia de los registros y las clases, las estructuras tienen semántica de paso por valor. Esto significa que son útiles principalmente para pequeños agregados de datos a los que accede y que se copian con frecuencia.

En la sintaxis anterior, se muestran dos formularios. La primera no es la sintaxis ligera; sin embargo, se utiliza frecuentemente porque, cuando se usan las palabras clave `struct` y `end`, se puede omitir el atributo `StructAttribute`, que aparece en el segundo formulario. `StructAttribute` se puede abreviar como `Struct`.

El *type-definition-elements-and-members* de la sintaxis anterior representa las declaraciones y definiciones de miembros. Las estructuras pueden tener constructores y campos mutables e inmutables, y pueden declarar implementaciones de interfaces y miembros. Para obtener más información, consulte [Miembros](#).

Las estructuras no pueden participar en la herencia, no pueden contener enlaces `let` ni `do`, y no puede contener de forma recursiva campos de su propio tipo (aunque pueden contener celdas de referencia que hagan referencia a su propio tipo).

Dado que las estructuras no permiten enlaces `let`, debe declarar campos en estructuras mediante el uso de la palabra clave `val`. La palabra clave `val` define un campo y su tipo, pero no permite la inicialización. En su lugar, las declaraciones `val` se inicializan en cero o null. Por este motivo, las estructuras que tienen un constructor implícito (es decir, los parámetros que se proporcionan inmediatamente después del nombre de la estructura en la declaración) requieren que las declaraciones `val` se anoten con el atributo `DefaultValue`. Las estructuras que tienen un constructor definido todavía admiten la inicialización en cero. Por lo tanto, el atributo `DefaultValue` es una declaración de que ese valor cero es válido para el campo. Los constructores implícitos de las estructuras no realizan ninguna acción porque los enlaces `let` y `do` no están permitidos en el tipo, pero los valores de parámetro de constructor implícito pasados están disponibles como campos privados.

Los constructores explícitos podrían implicar la inicialización de los valores de campo. Cuando se tiene una estructura con un constructor explícito, se admite la inicialización en cero; sin embargo, no se utiliza el atributo `DefaultValue` en las declaraciones `val` porque entra en conflicto con el constructor explícito. Para obtener `val`

más información acerca de las declaraciones, vea [Campos explícitos](#): `val` la palabra clave.

Los atributos y modificadores de accesibilidad están permitidos en las estructuras y siguen las mismas reglas que las de otros tipos. Para obtener más información, consulte [Atributos](#) y control de [acceso](#).

Los siguientes ejemplos de código ilustran las definiciones de la estructura.

```
// In Point3D, three immutable values are defined.
// x, y, and z will be initialized to 0.0.
type Point3D =
    struct
        val x: float
        val y: float
        val z: float
    end

// In Point2D, two immutable values are defined.
// It also has a member which computes a distance between itself and another Point2D.
// Point2D has an explicit constructor.
// You can create zero-initialized instances of Point2D, or you can
// pass in arguments to initialize the values.
type Point2D =
    struct
        val X: float
        val Y: float
        new(x: float, y: float) = { X = x; Y = y }

        member this.GetDistanceFrom(p: Point2D) =
            let dX = (p.X - this.X) ** 2.0
            let dY = (p.Y - this.Y) ** 2.0

            dX + dY
            |> sqrt
    end
```

Estructuras ByRefLike

Puede definir sus propias estructuras `byref` que se pueden adherir a la semántica -como: consulte [Byrefs](#) para obtener más información. Esto se hace [IsByRefLikeAttribute](#) con el atributo:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` no implica `Struct`. Ambos deben estar presentes en el tipo.

Una `byref` estructura "-like" en F es un tipo de valor enlazado a la pila. Nunca se asigna en el montón administrado. Una `byref` estructura -like es útil para la programación de alto rendimiento, ya que se aplica con un conjunto de comprobaciones sólidas sobre la duración y la no captura. Las reglas son:

- Se pueden utilizar como parámetros de función, parámetros de método, variables locales, retornos de método.
- No pueden ser miembros estáticos o de instancia de una clase o estructura normal.
- No se pueden capturar mediante `async` ninguna construcción de cierre (métodos o expresiones lambda).
- No se pueden utilizar como parámetro genérico.

Aunque estas reglas restringen fuertemente el uso, lo hacen para cumplir la promesa de la informática de alto rendimiento de una manera segura.

Estructuras ReadOnly

Puede anotar estructuras [IsReadOnlyAttribute](#) con el atributo. Por ejemplo:

```
[<IsReadOnly; Struct>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsReadOnly` no implica `Struct`. Debe agregar ambos para `IsReadOnly` tener una estructura.

El uso de este atributo emite metadatos que permiten `inref<'T>` que `in ref` F- y C- sepan tratarlo como y , respectivamente.

Definir un valor mutable dentro de una estructura readonly produce un error.

Struct Records y Uniones Discriminadas

Puede representar [Registros](#) y [Uniones discriminadas](#) como estructuras con el `[<Struct>]` atributo. Consulte cada artículo para obtener más información.

Consulte también

- [Referencia del lenguaje f](#)
- [Clases](#)
- [Registros](#)
- [Miembros](#)

Herencia

19/03/2020 • 6 minutes to read • [Edit Online](#)

La herencia se utiliza para modelar la relación "is-a", o subtipando, en la programación orientada a objetos.

Especificación de relaciones de herencia

Las relaciones de `inherit` herencia se especifican mediante la palabra clave en una declaración de clase. La forma sintáctica básica se muestra en el ejemplo siguiente.

```
type MyDerived(...) =  
    inherit MyBase(...)
```

Una clase puede tener como máximo una clase base directa. Si no especifica una clase base `inherit` mediante la palabra clave, `System.Object` la clase hereda implícitamente de .

Miembros heredados

Si una clase hereda de otra clase, los métodos y miembros de la clase base están disponibles para los usuarios de la clase derivada como si fueran miembros directos de la clase derivada.

Cualquier let enlaces y constructor parámetros son privados a una clase y, por lo tanto, no se puede tener acceso desde las clases derivadas.

La `base` palabra clave está disponible en clases derivadas y hace referencia a la instancia de clase base. Se utiliza como el autoidentificador.

Métodos virtuales y modificaciones

Los métodos virtuales (y las propiedades) funcionan de forma algo diferente en F# - en comparación con otros lenguajes .NET. Para declarar un nuevo miembro `abstract` virtual, utilice la palabra clave. Esto se hace independientemente de si se proporciona una implementación predeterminada para ese método. Por lo tanto, una definición completa de un método virtual en una clase base sigue este patrón:

```
abstract member [method-name] : [type]  
  
default [self-identifier].[method-name] [argument-list] = [method-body]
```

Y en una clase derivada, una invalidación de este método virtual sigue este patrón:

```
override [self-identifier].[method-name] [argument-list] = [method-body]
```

Si omite la implementación predeterminada en la clase base, la clase base se convierte en una clase abstracta.

En el ejemplo de código siguiente se `function1` muestra la declaración de un nuevo método virtual en una clase base y cómo reemplazarlo en una clase derivada.


```

type MyClassBase1() =
  let mutable z = 0
  abstract member function1 : int -> int
  default u.function1(a : int) = z <- z + a; z

type MyClassDerived1() =
  inherit MyClassBase1()
  override u.function1(a: int) = a + 1

```

Constructores y herencia

El constructor de la clase base debe llamarse en la clase derivada. Los argumentos del constructor de la clase `inherit` base aparecen en la lista de argumentos de la cláusula. Los valores que se utilizan deben determinarse a partir de los argumentos proporcionados al constructor de clase derivada.

El código siguiente muestra una clase base y una clase derivada, donde la clase derivada llama al constructor de la clase base en la cláusula `inherit`:

```

type MyClassBase2(x: int) =
  let mutable z = x * x
  do for i in 1..z do printf "%d " i

type MyClassDerived2(y: int) =
  inherit MyClassBase2(y * 2)
  do for i in 1..y do printf "%d " i

```

En el caso de varios constructores, se puede usar el código siguiente. La primera línea de los constructores `inherit` de clase según lo éstos `val` es la cláusula y los campos aparecen como campos explícitos que se declaran con la palabra clave. Para obtener más información, consulte [Campos explícitos: `val` La palabra clave](#).

```

type BaseClass =
  val string1 : string
  new (str) = { string1 = str }
  new () = { string1 = "" }

type DerivedClass =
  inherit BaseClass

  val string2 : string
  new (str1, str2) = { inherit BaseClass(str1); string2 = str2 }
  new (str2) = { inherit BaseClass(); string2 = str2 }

let obj1 = DerivedClass("A", "B")
let obj2 = DerivedClass("A")

```

Alternativas a la herencia

En los casos en que se requiere una modificación menor de un tipo, considere la posibilidad de usar una expresión de objeto como alternativa a la herencia. En el ejemplo siguiente se muestra el uso de una expresión de objeto como alternativa a la creación de un nuevo tipo derivado:

```
open System
```

```
let object1 = { new Object() with  
    override this.ToString() = "This overrides object.ToString()"  
}
```

```
printfn "%s" (object1.ToString())
```

Para obtener más información acerca de las expresiones de objeto, vea [Expresiones de objeto](#).

Al crear jerarquías de objetos, considere la posibilidad de usar una unión discriminada en lugar de una herencia. Las uniones discriminadas también pueden modelar un comportamiento variado de diferentes objetos que comparten un tipo general común. Una sola unión discriminada a menudo puede eliminar la necesidad de un número de clases derivadas que son variaciones menores entre sí. Para obtener información sobre las uniones discriminadas, consulte [Uniones discriminadas](#).

Consulte también

- [Expresiones de objeto](#)
- [Referencia del lenguaje f](#)

Interfaces

23/10/2019 • 6 minutes to read • [Edit Online](#)

Las *interfaces* especifican conjuntos de miembros relacionados que otras clases implementan.

Sintaxis

```
// Interface declaration:
[ attributes ]
type [accessibility-modifier] interface-name =
  [ interface ]    [ inherit base-interface-name ...]
  abstract member1 : [ argument-types1 -> ] return-type1
  abstract member2 : [ argument-types2 -> ] return-type2
  ...
[ end ]

// Implementing, inside a class type definition:
interface interface-name with
  member self-identifier.member1argument-list = method-body1
  member self-identifier.member2argument-list = method-body2

// Implementing, by using an object expression:
[ attributes ]
let class-name (argument-list) =
  { new interface-name with
    member self-identifier.member1argument-list = method-body1
    member self-identifier.member2argument-list = method-body2
    [ base-interface-definitions ]
  }
  member-list
```

Comentarios

Las declaraciones de interfaz son similares a las declaraciones de clase, salvo que no se implementa ningún miembro. En su lugar, todos los miembros son abstractos, como se indica `abstract` en la palabra clave. No se proporciona un cuerpo de método para los métodos abstractos. Sin embargo, puede proporcionar una implementación predeterminada al incluir también una definición independiente del miembro como un método junto con la `default` palabra clave. Hacerlo es equivalente a crear un método virtual en una clase base en otros lenguajes .NET. Este tipo de método virtual se puede invalidar en las clases que implementan la interfaz.

La accesibilidad predeterminada para las interfaces `public` es.

Opcionalmente, puede asignar un nombre a cada parámetro de método F# usando la sintaxis normal:

```
type ISprintable =
  abstract member Print : format:string -> unit
```

En el ejemplo `ISprintable` anterior, el `Print` método tiene un único parámetro del tipo `string` con el nombre `format`.

Hay dos maneras de implementar interfaces: mediante el uso de expresiones de objeto y el uso de tipos de clase. En cualquier caso, el tipo de clase o la expresión de objeto proporciona cuerpos de método para los métodos abstractos de la interfaz. Las implementaciones son específicas de cada tipo que implementa la interfaz. Por lo tanto, los métodos de interfaz en tipos diferentes pueden ser diferentes entre sí.

Las palabras `interface` clave `end` y, que marcan el inicio y el final de la definición, son opcionales cuando se usa la sintaxis ligera. Si no usa estas palabras clave, el compilador intenta deducir si el tipo es una clase o una interfaz mediante el análisis de las construcciones que se usan. Si define un miembro o usa otra sintaxis de clase, el tipo se interpreta como una clase.

El estilo de codificación de .NET consiste en comenzar todas las interfaces `I` con una letra mayúscula.

Implementar interfaces mediante tipos de clase

Puede implementar una o más interfaces en un tipo de clase mediante la `interface` palabra clave, el nombre de la interfaz y la `with` palabra clave, seguida de las definiciones de miembro de interfaz, como se muestra en el código siguiente.

```
type IPrintable =  
    abstract member Print : unit -> unit  
  
type SomeClass1(x: int, y: float) =  
    interface IPrintable with  
        member this.Print() = printfn "%d %f" x y
```

Las implementaciones de interfaz se heredan, por lo que las clases derivadas no necesitan volver a implementarlas.

Llamar a métodos de interfaz

Solo se puede llamar a los métodos de interfaz a través de la interfaz, no a través de ningún objeto del tipo que implementa la interfaz. Por lo tanto, es posible que tenga que convertir al tipo de interfaz mediante `:>` el operador o `upcast` el operador para llamar a estos métodos.

Para llamar al método de interfaz cuando tiene un objeto de tipo `SomeClass`, debe convertir el objeto al tipo de interfaz, como se muestra en el código siguiente.

```
let x1 = new SomeClass1(1, 2.0)  
(x1 :> IPrintable).Print()
```

Una alternativa consiste en declarar un método en el objeto que convierte y llama al método de interfaz, como en el ejemplo siguiente.

```
type SomeClass2(x: int, y: float) =  
    member this.Print() = (this :> IPrintable).Print()  
    interface IPrintable with  
        member this.Print() = printfn "%d %f" x y  
  
let x2 = new SomeClass2(1, 2.0)  
x2.Print()
```

Implementar interfaces mediante expresiones de objeto

Las expresiones de objeto proporcionan una manera breve de implementar una interfaz. Son útiles cuando no es necesario crear un tipo con nombre y simplemente se desea un objeto que admita los métodos de interfaz, sin ningún método adicional. En el código siguiente se muestra una expresión de objeto.

```
let makePrintable(x: int, y: float) =  
    { new IPrintable with  
        member this.Print() = printfn "%d %f" x y }  
let x3 = makePrintable(1, 2.0)  
x3.Print()
```

Herencia de interfaz

Las interfaces pueden heredar de una o varias interfaces base.

```
type Interface1 =  
    abstract member Method1 : int -> int  
  
type Interface2 =  
    abstract member Method2 : int -> int  
  
type Interface3 =  
    inherit Interface1  
    inherit Interface2  
    abstract member Method3 : int -> int  
  
type MyClass() =  
    interface Interface3 with  
        member this.Method1(n) = 2 * n  
        member this.Method2(n) = n + 100  
        member this.Method3(n) = n / 10
```

Vea también

- [Referencia del lenguaje F#](#)
- [Expresiones de objeto](#)
- [Clases](#)

Clases abstractas

23/10/2019 • 8 minutes to read • [Edit Online](#)

Las *clases abstractas* son clases que dejan algunos o todos los miembros no implementados, de modo que las implementaciones pueden ser proporcionadas por clases derivadas.

Sintaxis

```
// Abstract class syntax.  
[<AbstractClass>]  
type [ accessibility-modifier ] abstract-class-name =  
[ inherit base-class-or-interface-name ]  
[ abstract-member-declarations-and-member-definitions ]  
  
// Abstract member syntax.  
abstract member member-name : type-signature
```

Comentarios

En la programación orientada a objetos, una clase abstracta se utiliza como una clase base de una jerarquía y representa la funcionalidad común de un conjunto diverso de tipos de objeto. Como el nombre "abstract" implica, las clases abstractas no suelen corresponder directamente con entidades concretas del dominio del problema. Sin embargo, representan las distintas entidades concretas que tienen en común.

Las clases abstractas deben `AbstractClass` tener el atributo. Pueden tener miembros implementados y no implementados. El uso del término *abstracto* cuando se aplica a una clase es igual que en otros lenguajes .net; sin embargo, el uso del término *abstracto* cuando se aplica a métodos (y propiedades) es un poco F# diferente en respecto a su uso en otros lenguajes .net. En F#, cuando un método se marca con la `abstract` palabra clave, indica que un miembro tiene una entrada, conocida como *ranura de envío virtual*, en la tabla interna de funciones virtuales para ese tipo. En otras palabras, el método es virtual, aunque la `virtual` palabra clave no se usa en F# el lenguaje. La palabra `abstract` clave se usa en métodos virtuales independientemente de si se implementa el método. La declaración de una ranura de envío virtual es independiente de la definición de un método para esa ranura de envío. Por lo tanto F# , el equivalente de una declaración y definición de método virtual en otro lenguaje .net es una combinación de una declaración de método abstracto y una definición independiente `default` , con la `override` palabra clave o la palabra clave. Para obtener más información y ejemplos, vea [métodos](#).

Una clase se considera abstracta solo si hay métodos abstractos que se declaran pero no se definen. Por lo tanto, las clases que tienen métodos abstractos no son necesariamente clases abstractas. A menos que una clase tenga métodos abstractos sin definir, no use el atributo **AbstractClass** .

En la sintaxis anterior, el *modificador de accesibilidad* puede `public` ser `private` , `internal` o. Para más información, vea [Access Control](#) (Control de acceso).

Como con otros tipos, las clases abstractas pueden tener una clase base y una o varias interfaces base. Cada clase base o interfaz aparece en una línea independiente junto con la `inherit` palabra clave.

La definición de tipo de una clase abstracta puede contener miembros totalmente definidos, pero también puede contener miembros abstractos. La sintaxis de los miembros abstractos se muestra por separado en la sintaxis anterior. En esta sintaxis, la *signatura de tipo* de un miembro es una lista que contiene los tipos de parámetro en orden y los tipos de valor `->` devuelto, separados por `*` tokens o tokens, según corresponda para los

parámetros currificados y de tupla. La sintaxis de las firmas de tipo de miembro abstracto es la misma que la que se usa en los archivos de firma y que se muestra en IntelliSense en el editor de Visual Studio Code.

En el código siguiente se muestra una forma de clase abstracta, que tiene dos clases derivadas no abstractas, Square y Circle. En el ejemplo se muestra cómo usar clases, métodos y propiedades abstractos. En el ejemplo, la forma de clase abstracta representa los elementos comunes del círculo y el cuadrado de las entidades concretas. Las características comunes de todas las formas (en un sistema de coordenadas bidimensional) se abstraen en la clase de forma: la posición en la cuadrícula, un ángulo de rotación y las propiedades del área y del perímetro. Se pueden invalidar, excepto en el caso de la posición, el comportamiento de las formas individuales que no pueden cambiar.

Se puede invalidar el método de rotación, como en la clase Circle, que es la rotación invariable debido a su simetría. Por lo tanto, en la clase Circle, el método de rotación se reemplaza por un método que no hace nada.

```
// An abstract class that has some methods and properties defined
// and some left abstract.
[<AbstractClass>]
type Shape2D(x0 : float, y0 : float) =
    let mutable x, y = x0, y0
    let mutable rotAngle = 0.0

    // These properties are not declared abstract. They
    // cannot be overridden.
    member this.CenterX with get() = x and set xval = x <- xval
    member this.CenterY with get() = y and set yval = y <- yval

    // These properties are abstract, and no default implementation
    // is provided. Non-abstract derived classes must implement these.
    abstract Area : float with get
    abstract Perimeter : float with get
    abstract Name : string with get

    // This method is not declared abstract. It cannot be
    // overridden.
    member this.Move dx dy =
        x <- x + dx
        y <- y + dy

    // An abstract method that is given a default implementation
    // is equivalent to a virtual method in other .NET languages.
    // Rotate changes the internal angle of rotation of the square.
    // Angle is assumed to be in degrees.
    abstract member Rotate: float -> unit
    default this.Rotate(angle) = rotAngle <- rotAngle + angle

type Square(x, y, sideLengthIn) =
    inherit Shape2D(x, y)
    member this.SideLength = sideLengthIn
    override this.Area = this.SideLength * this.SideLength
    override this.Perimeter = this.SideLength * 4.
    override this.Name = "Square"

type Circle(x, y, radius) =
    inherit Shape2D(x, y)
    let PI = 3.141592654
    member this.Radius = radius
    override this.Area = PI * this.Radius * this.Radius
    override this.Perimeter = 2. * PI * this.Radius
    // Rotating a circle does nothing, so use the wildcard
    // character to discard the unused argument and
    // evaluate to unit.
    override this.Rotate(_) = ()
    override this.Name = "Circle"

let square1 = new Square(0.0, 0.0, 10.0)
```

```

let circle1 = new Circle(0.0, 0.0, 5.0)
circle1.CenterX <- 1.0
circle1.CenterY <- -2.0
square1.Move -1.0 2.0
square1.Rotate 45.0
circle1.Rotate 45.0
printfn "Perimeter of square with side length %f is %f, %f"
        (square1.SideLength) (square1.Area) (square1.Perimeter)
printfn "Circumference of circle with radius %f is %f, %f"
        (circle1.Radius) (circle1.Area) (circle1.Perimeter)

let shapeList : list<Shape2D> = [ (square1 :> Shape2D);
                                   (circle1 :> Shape2D) ]
List.iter (fun (elem : Shape2D) ->
            printfn "Area of %s: %f" (elem.Name) (elem.Area))
        shapeList

```

Salida:

```

Perimeter of square with side length 10.000000 is 40.000000
Circumference of circle with radius 5.000000 is 31.415927
Area of Square: 100.000000
Area of Circle: 78.539816

```

Vea también

- [Clases](#)
- [Miembros](#)
- [Métodos](#)
- [Propiedades](#)

Miembros

04/11/2019 • 2 minutes to read • [Edit Online](#)

En esta sección se describen los miembros de los tipos de objeto de F#.

Comentarios

Los *miembros* son características que forman parte de una definición de tipo y se declaran con la palabra clave `member`. Los tipos de objeto de F#, como los registros, clases, uniones discriminadas, interfaces y estructuras, admiten miembros. Para más información, vea [Registros](#), [Clases](#), [Uniones discriminadas](#), [Interfaces](#) y [Estructuras](#).

Normalmente, los miembros componen la interfaz pública de un tipo, por lo que son públicos a menos que se especifique lo contrario. Los miembros también pueden declararse como privados o internos. Para más información, vea [Access Control](#) (Control de acceso). También se pueden usar firmas de tipos para exponer o no determinados miembros de un tipo. Para más información, vea [Signatures](#) (Firmas).

Los campos privados y los enlaces `do`, que se usan únicamente con las clases, no son miembros auténticos ya que nunca forman parte de la interfaz pública de un tipo y no se declaran con la palabra clave `member`, pero también se describen en esta sección.

Temas relacionados

TEMA	DESCRIPCIÓN
<code>let</code> Bindings in Classes (Enlaces <code>let</code> en clases)	Describe la definición de campos privados y funciones en las clases.
<code>do</code> Bindings in Classes (Enlaces <code>do</code> en clases)	Describe la especificación de código de inicialización de objetos.
Propiedades	Describe los miembros de propiedad de las clases y otros tipos.
Propiedades indexadas	Describe propiedades similares a matrices de las clases y otros tipos.
Métodos	Describe funciones que son miembros de un tipo.
Constructores	Describe funciones especiales que inicializan objetos de un tipo.
Sobrecarga de operadores	Describe la definición de operadores personalizados para tipos.
Eventos	Describe la definición y la compatibilidad con el control de eventos en F#.
Campos explícitos: palabra clave <code>val</code>	Describe la definición de campos no inicializados en un tipo.

Enlaces let en clases

23/10/2019 • 3 minutes to read • [Edit Online](#)

Puede definir campos privados y funciones privadas para F# las clases mediante el `let` uso de enlaces en la definición de clase.

Sintaxis

```
// Field.  
[static] let [ mutable ] binding1 [ and ... binding-n ]  
  
// Function.  
[static] let [ rec ] binding1 [ and ... binding-n ]
```

Comentarios

La sintaxis anterior aparece después del encabezado de clase y las declaraciones de herencia, pero antes de cualquier definición de miembro. La sintaxis es similar a la `let` de los enlaces fuera de las clases, pero los nombres definidos en una clase tienen un ámbito limitado a la clase. Un `let` enlace crea un campo o una función privados; para exponer datos o funciones públicamente, declare una propiedad o un método de miembro.

Un `let` enlace que no es estático se denomina enlace de `let` instancia. Los `let` enlaces de instancia se ejecutan cuando se crean los objetos. Los `let` enlaces estáticos forman parte del inicializador estático de la clase, que se garantiza que se ejecuta antes de que se use por primera vez el tipo.

El código incluido en `let` los enlaces de instancia puede usar los parámetros del constructor principal.

Los atributos y los modificadores de accesibilidad no `let` se permiten en los enlaces de clases.

En los siguientes ejemplos de código se muestran `let` varios tipos de enlaces en las clases.

```

type PointWithCounter(a: int, b: int) =
  // A variable i.
  let mutable i = 0

  // A let binding that uses a pattern.
  let (x, y) = (a, b)

  // A private function binding.
  let privateFunction x y = x * x + 2*y

  // A static let binding.
  static let mutable count = 0

  // A do binding.
  do
    count <- count + 1

  member this.Prop1 = x
  member this.Prop2 = y
  member this.CreatedCount = count
  member this.FunctionValue = privateFunction x y

let point1 = PointWithCounter(10, 52)

printfn "%d %d %d %d" (point1.Prop1) (point1.Prop2) (point1.CreatedCount) (point1.FunctionValue)

```

La salida es la siguiente.

```
10 52 1 204
```

Maneras alternativas de crear campos

También puede usar la `val` palabra clave para crear un campo privado. Cuando se usa `val` la palabra clave, el campo no recibe un valor cuando se crea el objeto, sino que se inicializa con un valor predeterminado. Para obtener más información, [vea campos explícitos: Palabra clave Val](#).

También puede definir campos privados en una clase utilizando una definición de miembro y agregando la palabra `private` clave a la definición. Esto puede ser útil si espera cambiar la accesibilidad de un miembro sin reescribir el código. Para más información, [vea Access Control](#) (Control de acceso).

Vea también

- [Miembros](#)
- `do` [Bindings in Classes](#) (Enlaces `do` en clases)
- `let` [Enlaces](#)

Enlaces do en clases

23/10/2019 • 4 minutes to read • [Edit Online](#)

Un `do` enlace en una definición de clase realiza acciones cuando se construye el objeto o, para un `do` enlace estático, cuando se usa por primera vez el tipo.

Sintaxis

```
[static] do expression
```

Comentarios

Un `do` enlace aparece junto con los enlaces `let` o después, pero antes de las definiciones de miembro en una definición de clase. Aunque la `do` palabra clave es opcional `do` para los enlaces en el nivel de módulo, no es opcional `do` para los enlaces en una definición de clase.

Para la construcción de cada objeto de un tipo determinado, los enlaces no `do` estáticos y los enlaces no `let` estáticos se ejecutan en el orden en que aparecen en la definición de clase. Pueden `do` producirse varios enlaces en un tipo. Los enlaces no estáticos `let` y los enlaces no estáticos `do` se convierten en el cuerpo del constructor principal. El código de la sección de enlaces `do` no estáticos puede hacer referencia a los parámetros del constructor principal y a los valores o funciones que `let` se definen en la sección de enlaces.

Los enlaces no `do` estáticos pueden tener acceso a los miembros de la clase siempre que la clase tenga un identificador propio definido por una `as` palabra clave en el encabezado de la clase, y siempre y cuando todos los usos de esos miembros estén calificados con el propio identificador de la clase.

Dado `let` que los enlaces inicializan los campos privados de una clase, que a menudo es necesario para garantizar que los miembros `do` se comportan según lo `let` esperado, los enlaces normalmente se colocan después de los enlaces para que el `do` código del enlace pueda Ejecute con un objeto completamente inicializado. Si el código intenta utilizar un miembro antes de que se complete la inicialización, se genera una excepción `InvalidOperationException`.

Los `do` enlaces estáticos pueden hacer referencia a los miembros o campos estáticos de la clase envolvente, pero no a los miembros o campos de instancia. Los `do` enlaces estáticos forman parte del inicializador estático de la clase, que se garantiza que se ejecuta antes de que se use por primera vez la clase.

Los atributos se omiten para `do` los enlaces de los tipos. Si se requiere un atributo para el código que se ejecuta en `do` un enlace, debe aplicarse al constructor principal.

En el código siguiente, una clase tiene un enlace `do` estático y un enlace no estático `do`. El objeto tiene un constructor que tiene dos parámetros, `a` y `b`, y dos campos privados se definen en los `let` enlaces de la clase. También se definen dos propiedades. Todos se encuentran en el ámbito de la sección de enlaces `do` no estáticos, como se muestra en la línea que imprime todos esos valores.

open System

```
type MyType(a:int, b:int) as this =  
    inherit Object()  
    let x = 2*a  
    let y = 2*b  
    do printfn "Initializing object %d %d %d %d %d %d"  
        a b x y (this.Prop1) (this.Prop2)  
    static do printfn "Initializing MyType."  
    member this.Prop1 = 4*x  
    member this.Prop2 = 4*y  
    override this.ToString() = System.String.Format("{0} {1}", this.Prop1, this.Prop2)  
  
let obj1 = new MyType(1, 2)
```

La salida es la siguiente.

```
Initializing MyType.  
Initializing object 1 2 2 4 8 16
```

Vea también

- [Miembros](#)
- [Clases](#)
- [Constructores](#)
- [let](#) [Bindings in Classes](#) (Enlaces [let](#) en clases)
- [do](#) [Enlaces](#)

Propiedades

23/10/2019 • 12 minutes to read • [Edit Online](#)

Las propiedades son miembros que representan valores asociados a un objeto.

Sintaxis

```
// Property that has both get and set defined.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with [accessibility-modifier] get() =
    get-function-body
and [accessibility-modifier] set parameter =
    set-function-body

// Alternative syntax for a property that has get and set.
[ attributes-for-get ]
[ static ] member [accessibility-modifier-for-get] [self-identifier.]PropertyName =
    get-function-body
[ attributes-for-set ]
[ static ] member [accessibility-modifier-for-set] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName =
    get-function-body

// Alternative syntax for property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with get() =
    get-function-body

// Property that has set only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Automatically implemented properties.
[ attributes ]
[ static ] member val [accessibility-modifier] PropertyName = initialization-expression [ with get, set ]
```

Comentarios

Las propiedades representan la relación "tiene una" en la programación orientada a objetos, que representa los datos que están asociados a instancias de objeto o, para las propiedades estáticas, con el tipo.

Puede declarar propiedades de dos maneras, en función de si desea especificar explícitamente el valor subyacente (también denominado memoria auxiliar) para la propiedad, o si desea permitir que el compilador genere automáticamente la memoria auxiliar. Por lo general, debe usar la manera más explícita si la propiedad tiene una implementación no trivial y la manera automática cuando la propiedad es simplemente un contenedor simple para un valor o una variable. Para declarar una propiedad explícitamente, use `member` la palabra clave. Esta sintaxis declarativa va seguida de la sintaxis que especifica los `get` métodos `set` y, también denominados

descriptores de acceso. Las distintas formas de sintaxis explícita que se muestran en la sección sintaxis se utilizan para las propiedades de lectura/escritura, de solo lectura y de solo escritura. En el caso de las propiedades de solo lectura, `get` solo se define un método; en el caso de las `set` propiedades de solo escritura, solo se define un método. Tenga en cuenta que cuando una propiedad `get` tiene `set` los descriptores de acceso y, la sintaxis alternativa le permite especificar los atributos y modificadores de accesibilidad que son diferentes para cada descriptor de acceso, como se muestra en el código siguiente.

```
// A read-only property.
member this.MyReadOnlyProperty = myInternalValue
// A write-only property.
member this.MyWriteOnlyProperty with set (value) = myInternalValue <- value
// A read-write property.
member this.MyReadWriteProperty
  with get () = myInternalValue
  and set (value) = myInternalValue <- value
```

En el caso de las propiedades de lectura y escritura `get`, `set` que tienen un método y `get`, `set` se puede invertir el orden de y. Como alternativa, puede proporcionar la sintaxis que se muestra `get` solo y la sintaxis que se `set` muestra solo en lugar de usar la sintaxis combinada. De este modo, resulta más fácil comentar el método `get` o `set` individual, en caso de que sea algo que sea necesario realizar. Esta alternativa al uso de la sintaxis combinada se muestra en el código siguiente.

```
member this.MyReadWriteProperty with get () = myInternalValue
member this.MyReadWriteProperty with set (value) = myInternalValue <- value
```

Los valores privados que contienen los datos de las propiedades se denominan *almacenes de respaldo*. Para que el compilador cree automáticamente la memoria auxiliar, use `member val` las palabras clave, omita el autoidentificador y proporcione una expresión para inicializar la propiedad. Si la propiedad va a ser mutable, incluya `with get, set`. Por ejemplo, el siguiente tipo de clase incluye dos propiedades implementadas automáticamente. `Property1` es de solo lectura y se inicializa en el argumento proporcionado al constructor principal, y `Property2` es una propiedad configurable inicializada en una cadena vacía:

```
type MyClass(property1 : int) =
  member val Property1 = property1
  member val Property2 = "" with get, set
```

Las propiedades implementadas automáticamente forman parte de la inicialización de un tipo, por lo que deben incluirse antes que cualquier otra `let` definición de miembro `do`, al igual que los enlaces y enlaces en una definición de tipo. Tenga en cuenta que la expresión que inicializa una propiedad implementada automáticamente solo se evalúa durante la inicialización y no cada vez que se tiene acceso a la propiedad. Este comportamiento es distinto del comportamiento de una propiedad implementada explícitamente. Lo que significa de forma eficaz es que el código para inicializar estas propiedades se agrega al constructor de una clase. Considere el siguiente código que muestra esta diferencia:

```

type MyClass() =
    let random = new System.Random()
    member val AutoProperty = random.Next() with get, set
    member this.ExplicitProperty = random.Next()

let class1 = new MyClass()

printfn "class1.AutoProperty = %d" class1.AutoProperty
printfn "class1.AutoProperty = %d" class1.AutoProperty
printfn "class1.ExplicitProperty = %d" class1.ExplicitProperty
printfn "class1.ExplicitProperty = %d" class1.ExplicitProperty

```

Salida

```

class1.AutoProperty = 1853799794
class1.AutoProperty = 1853799794
class1.ExplicitProperty = 978922705
class1.ExplicitProperty = 1131210765

```

La salida del código anterior muestra que el valor de `autoProperty` no cambia cuando se llama varias veces, mientras que `ExplicitProperty` cambia cada vez que se llama. Esto demuestra que la expresión para una propiedad implementada automáticamente no se evalúa cada vez, como es el método captador de la propiedad explícita.

WARNING

Hay algunas bibliotecas, como la Entity Framework (`System.Data.Entity`) que realizan operaciones personalizadas en constructores de clase base que no funcionan bien con la inicialización de propiedades implementadas automáticamente. En esos casos, pruebe a usar propiedades explícitas.

Las propiedades pueden ser miembros de clases, estructuras, uniones discriminadas, registros, interfaces y extensiones de tipo, y también pueden definirse en expresiones de objeto.

Los atributos se pueden aplicar a las propiedades. Para aplicar un atributo a una propiedad, escriba el atributo en una línea independiente antes de la propiedad. Para obtener más información, consulte [Atributos](#) (Atributos).

De forma predeterminada, las propiedades son públicas. Los modificadores de accesibilidad también se pueden aplicar a las propiedades. Para aplicar un modificador de accesibilidad, agréguelo inmediatamente antes del nombre de la propiedad `get` si está pensado para aplicarse a los métodos y `set` ; agréguelo antes de las palabras clave y `set` si la `get` accesibilidad es diferente obligatorio para cada descriptor de acceso. El *modificador Accessibility* puede ser uno de los siguientes: `public` , `private` , `internal` . Para más información, vea [Access Control](#) (Control de acceso).

Las implementaciones de propiedad se ejecutan cada vez que se tiene acceso a una propiedad.

Propiedades estáticas y de instancia

Las propiedades pueden ser propiedades estáticas o de instancia. Las propiedades estáticas se pueden invocar sin una instancia de y se usan para los valores asociados al tipo, no con los objetos individuales. En el caso de las propiedades estáticas, omita el identificador automático. El autoidentificador es necesario para las propiedades de instancia.

La siguiente definición de propiedad estática se basa en un escenario en el que tiene un campo `myStaticValue` estático que es la memoria auxiliar para la propiedad.


```
static member MyStaticProperty
  with get() = myStaticValue
  and set(value) = myStaticValue <- value
```

Las propiedades también pueden ser de tipo matriz, en cuyo caso se denominan *propiedades indizadas*. Para obtener más información, vea [propiedades indizadas](#).

Anotación de tipo para propiedades

En muchos casos, el compilador tiene información suficiente para inferir el tipo de una propiedad a partir del tipo de la memoria auxiliar, pero puede establecer el tipo explícitamente agregando una anotación de tipo.

```
// To apply a type annotation to a property that does not have an explicit
// get or set, apply the type annotation directly to the property.
member this.MyProperty1 : int = myInternalValue
// If there is a get or set, apply the type annotation to the get or set method.
member this.MyProperty2 with get() : int = myInternalValue
```

Usar descriptores de acceso del conjunto de propiedades

Puede establecer propiedades que proporcionen `set` descriptores de acceso `<-` mediante el operador.

```
// Assume that the constructor argument sets the initial value of the
// internal backing store.
let mutable myObject = new MyType(10)
myObject.MyProperty <- 20
printfn "%d" (myObject.MyProperty)
```

El resultado es 20.

Propiedades abstractas

Las propiedades pueden ser abstractas. Como con los métodos `abstract`, solo significa que hay un envío virtual asociado a la propiedad. Las propiedades abstractas pueden ser realmente abstractas, es decir, sin una definición en la misma clase. Por lo tanto, la clase que contiene una propiedad de este tipo es una clase abstracta. Como alternativa, `abstract` solo puede significar que una propiedad es virtual y, en ese caso, una definición debe estar presente en la misma clase. Tenga en cuenta que las propiedades abstractas no deben ser privadas y, si un descriptor de acceso es abstracto, el otro también debe ser abstracto. Para obtener más información sobre las clases abstractas, vea [clases abstractas](#).

```

// Abstract property in abstract class.
// The property is an int type that has a get and
// set method
[<AbstractClass>]
type AbstractBase() =
  abstract Property1 : int with get, set

// Implementation of the abstract property
type Derived1() =
  inherit AbstractBase()
  let mutable value = 10
  override this.Property1 with get() = value and set(v : int) = value <- v

// A type with a "virtual" property.
type Base1() =
  let mutable value = 10
  abstract Property1 : int with get, set
  default this.Property1 with get() = value and set(v : int) = value <- v

// A derived type that overrides the virtual property
type Derived2() =
  inherit Base1()
  let mutable value2 = 11
  override this.Property1 with get() = value2 and set(v) = value2 <- v

```

Vea también

- [Miembros](#)
- [Métodos](#)

Propiedades indizadas

25/11/2019 • 4 minutes to read • [Edit Online](#)

Al definir una clase que resume los datos ordenados, a veces puede resultar útil proporcionar un acceso indizado a esos datos sin exponer la implementación subyacente. Esto se hace con el miembro `Item`.

Sintaxis

```
// Indexed property that can be read and written to
member self-identifier.Item
    with get(index-values) =
        get-member-body
    and set index-values values-to-set =
        set-member-body

// Indexed property can only be read
member self-identifier.Item
    with get(index-values) =
        get-member-body

// Indexed property that can only be set
member self-identifier.Item
    with set index-values values-to-set =
        set-member-body
```

Comentarios

Las formas de la sintaxis anterior muestran cómo definir las propiedades indizadas que tienen un método `get` y un `set`, tienen un método `get` solamente o tienen un método `set` solo. También puede combinar la sintaxis que se muestra para Get only y la sintaxis que se muestra solo para Set, y generar una propiedad que tenga get y set. Este último formulario le permite colocar distintos modificadores y atributos de accesibilidad en los métodos GET y set.

Con el nombre `Item`, el compilador trata la propiedad como una propiedad indizada predeterminada. Una *propiedad indizada predeterminada* es una propiedad a la que se puede tener acceso mediante una sintaxis similar a la de la matriz en la instancia del objeto. Por ejemplo, si `o` es un objeto del tipo que define esta propiedad, se utiliza la sintaxis `o.[index]` para tener acceso a la propiedad.

La sintaxis para tener acceso a una propiedad indizada no predeterminada consiste en proporcionar el nombre de la propiedad y el índice entre paréntesis, al igual que un miembro normal. Por ejemplo, si se llama a la propiedad en `o.Ordinal`, se escribe `o.Ordinal(index)` para tener acceso a ella.

Independientemente de la forma que use, siempre debe usar la forma currificada para el método Set en una propiedad indizada. Para obtener información sobre las funciones currificadas, vea [funciones](#).

Ejemplo

En el ejemplo de código siguiente se muestra la definición y el uso de las propiedades indizadas predeterminadas y no predeterminadas que tienen los métodos GET y set.

```

type NumberStrings() =
  let mutable ordinals = [| "one"; "two"; "three"; "four"; "five";
                             "six"; "seven"; "eight"; "nine"; "ten" |]
  let mutable cardinals = [| "first"; "second"; "third"; "fourth";
                              "fifth"; "sixth"; "seventh"; "eighth";
                              "ninth"; "tenth" |]

  member this.Item
    with get(index) = ordinals.[index]
    and set index value = ordinals.[index] <- value
  member this.Ordinal
    with get(index) = ordinals.[index]
    and set index value = ordinals.[index] <- value
  member this.Cardinal
    with get(index) = cardinals.[index]
    and set index value = cardinals.[index] <- value

let nstrs = new NumberStrings()
nstrs.[0] <- "ONE"
for i in 0 .. 9 do
  printf "%s " (nstrs.[i])
printfn ""

nstrs.Cardinal(5) <- "6th"

for i in 0 .. 9 do
  printf "%s " (nstrs.Ordinal(i))
  printf "%s " (nstrs.Cardinal(i))
printfn ""

```

Resultados

```

ONE two three four five six seven eight nine ten
ONE first two second three third four fourth five fifth six 6th
seven seventh eight eighth nine ninth ten tenth

```

Propiedades indizadas con varios valores de índice

Las propiedades indizadas pueden tener más de un valor de índice. En ese caso, los valores se separan mediante comas cuando se usa la propiedad. El método Set de esta propiedad debe tener dos argumentos currificados, el primero es una tupla que contiene las claves y el segundo es el valor que se va a establecer.

En el código siguiente se muestra el uso de una propiedad indizada con varios valores de índice.

```

open System.Collections.Generic

/// Basic implementation of a sparse matrix based on a dictionary
type SparseMatrix() =
  let table = new Dictionary<(int * int), float>()
  member _.Item
    // Because the key is comprised of two values, 'get' has two index values
    with get(key1, key2) = table.[(key1, key2)]

    // 'set' has two index values and a new value to place in the key's position
    and set (key1, key2) value = table.[(key1, key2)] <- value

let sm = new SparseMatrix()
for i in 1..1000 do
  sm.[i, i] <- float i * float i

```

Vea también

- [Miembros](#)

Métodos

19/03/2020 • 12 minutes to read • [Edit Online](#)

Un *método* es una función que está asociada a un tipo. En la programación orientada a objetos, se utilizan métodos para exponer e implementar la funcionalidad y el comportamiento de objetos y tipos.

Sintaxis

```
// Instance method definition.
[ attributes ]
member [inline] self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Static method definition.
[ attributes ]
static member [inline] method-name parameter-list [ : return-type ] =
    method-body

// Abstract method declaration or virtual dispatch slot.
[ attributes ]
abstract member method-name : type-signature

// Virtual method declaration and default implementation.
[ attributes ]
abstract member method-name : type-signature
[ attributes ]
default self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Override of inherited virtual method.
[ attributes ]
override self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Optional and DefaultParameterValue attributes on input parameters
[ attributes ]
[ modifier ] member [inline] self-identifier.method-name ([<Optional; DefaultParameterValue( default-value
)>] input) [ : return-type ]
```

Observaciones

En la sintaxis anterior, puede ver las diversas formas de declaraciones y definiciones de método. En los cuerpos de método más largos, un salto de línea sigue el signo igual (=) y se sangra todo el cuerpo del método.

Los atributos se pueden aplicar a cualquier declaración de método. Preceden a la sintaxis de una definición de método y normalmente se enumeran en una línea independiente. Para obtener más información, consulte [Atributos](#) (Atributos).

Los métodos `inline` se pueden marcar. Para más información sobre `inline`, vea [Inline Functions](#) (Funciones insertadas).

Los métodos no en línea se pueden utilizar de forma recursiva dentro del tipo; no hay necesidad de usar `rec` explícitamente la palabra clave.

Métodos de instancia

Los métodos de `member` instancia se declaran con la palabra clave y un *autoidentificador*, seguido de un punto (.) y el nombre y los parámetros del método. Como es el `let` caso de los enlaces, la *lista de parámetros* puede ser un patrón. Normalmente, se incluyen los parámetros de método entre paréntesis en un formulario de tupla, que es la forma en que aparecen los métodos en F - cuando se crean en otros lenguajes de .NET Framework. Sin embargo, la forma curried (parámetros separados por espacios) también es común, y también se admiten otros patrones.

En el ejemplo siguiente se muestra la definición y el uso de un método de instancia no abstracto.

```
type SomeType(factor0: int) =  
    let factor = factor0  
    member this.SomeMethod(a, b, c) =  
        (a + b + c) * factor  
  
    member this.SomeOtherMethod(a, b, c) =  
        this.SomeMethod(a, b, c) * factor
```

Dentro de los métodos de instancia, no utilice el identificador propio para tener acceso a los campos definidos mediante el uso de enlaces `let`. Utilice el identificador automático al acceder a otros miembros y propiedades.

Métodos estáticos

La `static` palabra clave se utiliza para especificar que se puede llamar a un método sin una instancia y no está asociado a una instancia de objeto. De lo contrario, los métodos son métodos de instancia.

En el ejemplo de la siguiente `let` sección se muestran `member` los campos declarados con `static` la palabra clave, los miembros de propiedad declarados con la palabra clave y un método estático declarado con la palabra clave.

En el ejemplo siguiente se muestra la definición y el uso de métodos estáticos. Supongamos que estas definiciones `SomeType` de método están en la clase de la sección anterior.

```
static member SomeStaticMethod(a, b, c) =  
    (a + b + c)  
  
static member SomeOtherStaticMethod(a, b, c) =  
    SomeType.SomeStaticMethod(a, b, c) * 100
```

Métodos abstractos y virtuales

La `abstract` palabra clave indica que un método tiene una ranura de envío virtual y es posible que no tenga una definición en la clase. Una ranura de *distribución virtual* es una entrada en una tabla de funciones mantenida internamente que se utiliza en tiempo de ejecución para buscar llamadas de función virtual en un tipo orientado a objetos. El mecanismo de envío virtual es el mecanismo que implementa el *polimorfismo*, una característica importante de la programación orientada a objetos. Una clase que tiene al menos un método abstracto sin una definición es una *clase abstracta*, lo que significa que no se puede crear ninguna instancia de esa clase. Para obtener más información acerca de las clases abstractas, vea [Clases abstractas](#).

Las declaraciones de método abstracto no incluyen un cuerpo de método. En su lugar, el nombre del método va seguido de dos puntos (:) y una firma de tipo para el método. La firma de tipo de un método es la misma que la que muestra IntelliSense al pausar el puntero del mouse sobre un nombre de método en el Editor de código de Visual Studio, excepto sin nombres de parámetro. El intérprete, `fsi.exe`, también muestra las firmas de tipo cuando se trabaja de forma interactiva. La firma de tipo de un método se forma enumerando los tipos de los parámetros, seguido por el tipo de valor devuelto, con los símbolos de separador adecuados. Los parámetros curridos están separados por `->` y los parámetros de tupla están separados por `*`. El valor devuelto siempre está separado

-> de los argumentos por un símbolo. Los paréntesis se pueden utilizar para agrupar parámetros complejos, como cuando un tipo de función es un parámetro, o para indicar cuándo una tupla se trata como un único parámetro en lugar de como dos parámetros.

También puede dar definiciones predeterminadas de métodos abstractos agregando la definición a la clase y usando la `default` palabra clave, como se muestra en el bloque de sintaxis de este tema. Un método abstracto que tiene una definición en la misma clase es equivalente a un método virtual en otros lenguajes de .NET Framework. Independientemente de si existe `abstract` o no una definición, la palabra clave crea una nueva ranura de distribución en la tabla de funciones virtuales para la clase.

Independientemente de si una clase base implementa sus métodos abstractos, las clases derivadas pueden proporcionar implementaciones de métodos abstractos. Para implementar un método abstracto en una clase derivada, defina un método que tenga `override` el `default` mismo nombre y firma en la clase derivada, excepto use la palabra clave `or` y proporcione el cuerpo del método. Las `override` palabras `default` clave significan exactamente lo mismo. Se `override` usa si el nuevo método reemplaza una implementación de clase base; se `default` usa cuando se crea una implementación en la misma clase que la declaración abstracta original. No utilice `abstract` la palabra clave en el método que implementa el método que se declaró abstracto en la clase base.

En el ejemplo siguiente `Rotate` se muestra un método abstracto que tiene una implementación predeterminada, el equivalente a un método virtual de .NET Framework.

```
type Ellipse(a0 : float, b0 : float, theta0 : float) =
    let mutable axis1 = a0
    let mutable axis2 = b0
    let mutable rotAngle = theta0
    abstract member Rotate: float -> unit
    default this.Rotate(delta : float) = rotAngle <- rotAngle + delta
```

En el ejemplo siguiente se muestra una clase derivada que reemplaza un método de clase base. En este caso, la invalidación cambia el comportamiento para que el método no haga nada.

```
type Circle(radius : float) =
    inherit Ellipse(radius, radius, 0.0)
    // Circles are invariant to rotation, so do nothing.
    override this.Rotate(_) = ()
```

Métodos sobrecargados

Los métodos sobrecargados son métodos que tienen nombres idénticos en un tipo determinado pero que tienen argumentos diferentes. Normalmente se usan argumentos opcionales en lugar de métodos sobrecargados. Sin embargo, se permiten métodos sobrecargados en el lenguaje, siempre que los argumentos estén en forma de tupla, no en forma de cuajada.

Argumentos opcionales

Comenzando con F 4.1, también puede tener argumentos opcionales con un valor de parámetro predeterminado en los métodos. Esto es para ayudar a facilitar la interoperación con el código de C. En el ejemplo siguiente se muestra la sintaxis:

```
// A class with a method M, which takes in an optional integer argument.
type C() =
    _.<M([<Optional; DefaultValue(12)>] i) = i + 1
```


Tenga en cuenta que `DefaultParameterValue` el valor pasado para debe coincidir con el tipo de entrada. En la muestra anterior, `int` es un archivo. Si se intenta pasar un `DefaultParameterValue` valor no entero, se produciría un error de compilación.

Ejemplo: Propiedades y métodos

El ejemplo siguiente contiene un tipo que tiene ejemplos de campos, funciones privadas, propiedades y un método estático.

```
type RectangleXY(x1 : float, y1: float, x2: float, y2: float) =
    // Field definitions.
    let height = y2 - y1
    let width = x2 - x1
    let area = height * width
    // Private functions.
    static let maxFloat (x: float) (y: float) =
        if x >= y then x else y
    static let minFloat (x: float) (y: float) =
        if x <= y then x else y
    // Properties.
    // Here, "this" is used as the self identifier,
    // but it can be any identifier.
    member this.X1 = x1
    member this.Y1 = y1
    member this.X2 = x2
    member this.Y2 = y2
    // A static method.
    static member intersection(rect1 : RectangleXY, rect2 : RectangleXY) =
        let x1 = maxFloat rect1.X1 rect2.X1
        let y1 = maxFloat rect1.Y1 rect2.Y1
        let x2 = minFloat rect1.X2 rect2.X2
        let y2 = minFloat rect1.Y2 rect2.Y2
        let result : RectangleXY option =
            if ( x2 > x1 && y2 > y1) then
                Some (RectangleXY(x1, y1, x2, y2))
            else
                None
        result

// Test code.
let testIntersection =
    let r1 = RectangleXY(10.0, 10.0, 20.0, 20.0)
    let r2 = RectangleXY(15.0, 15.0, 25.0, 25.0)
    let r3 : RectangleXY option = RectangleXY.intersection(r1, r2)
    match r3 with
    | Some(r3) -> printfn "Intersection rectangle: %f %f %f %f" r3.X1 r3.Y1 r3.X2 r3.Y2
    | None -> printfn "No intersection found."

testIntersection
```

Consulte también

- [Miembros](#)

Constructores

23/10/2019 • 11 minutes to read • [Edit Online](#)

En este artículo se describe cómo definir y usar constructores para crear e inicializar objetos de clase y estructura.

Construcción de objetos de clase

Los objetos de tipos de clase tienen constructores. Hay dos tipos de constructores. Uno es el constructor principal, cuyos parámetros aparecen entre paréntesis justo después del nombre de tipo. Especifique otros constructores adicionales opcionales mediante la `new` palabra clave. Cualquier constructor adicional de este tipo debe llamar al constructor principal.

El constructor principal contiene `let` enlaces `do` y que aparecen al principio de la definición de clase. Un `let` enlace declara los campos y métodos privados de la clase; un `do` enlace ejecuta código. Para obtener más información `let` sobre los enlaces de los constructores de clase, vea [let enlaces en clases](#). Para obtener más información `do` sobre los enlaces en constructores, vea [do enlaces en clases](#).

Independientemente de si el constructor al que desea llamar es un constructor principal o un constructor adicional, puede crear objetos mediante una `new` expresión, con o sin la palabra clave opcional. `new` Los objetos se inicializan junto con los argumentos del constructor, ya sea mediante una lista de los argumentos en orden y se separan mediante comas y entre paréntesis, o usando argumentos y valores con nombre entre paréntesis. También puede establecer las propiedades de un objeto durante la construcción del objeto utilizando los nombres de propiedad y asignando valores de la misma forma que se usan los argumentos del constructor con nombre.

En el código siguiente se muestra una clase que tiene un constructor y varias maneras de crear objetos:

```
// This class has a primary constructor that takes three arguments
// and an additional constructor that calls the primary constructor.
type MyClass(x0, y0, z0) =
  let mutable x = x0
  let mutable y = y0
  let mutable z = z0
  do
    printfn "Initialized object that has coordinates (%d, %d, %d)" x y z
  member this.X with get() = x and set(value) = x <- value
  member this.Y with get() = y and set(value) = y <- value
  member this.Z with get() = z and set(value) = z <- value
  new() = MyClass(0, 0, 0)

// Create by using the new keyword.
let myObject1 = new MyClass(1, 2, 3)
// Create without using the new keyword.
let myObject2 = MyClass(4, 5, 6)
// Create by using named arguments.
let myObject3 = MyClass(x0 = 7, y0 = 8, z0 = 9)
// Create by using the additional constructor.
let myObject4 = MyClass()
```

La salida es la siguiente:

```
Initialized object that has coordinates (1, 2, 3)
Initialized object that has coordinates (4, 5, 6)
Initialized object that has coordinates (7, 8, 9)
Initialized object that has coordinates (0, 0, 0)
```

Construcción de estructuras

Las estructuras siguen todas las reglas de las clases. Por lo tanto, puede tener un constructor principal y puede proporcionar constructores `new` adicionales mediante. Sin embargo, hay una diferencia importante entre las estructuras y las clases: las estructuras pueden tener un constructor sin parámetros (es decir, uno sin argumentos) aunque no se haya definido ningún constructor principal. El constructor sin parámetros inicializa todos los campos en el valor predeterminado de ese tipo, normalmente cero o su equivalente. Los constructores que defina para las estructuras deben tener al menos un argumento para que no entren en conflicto con el constructor sin parámetros.

Además, las estructuras suelen tener campos que se crean mediante la `val` palabra clave; las clases también pueden tener estos campos. Las estructuras y las clases que tienen campos definidos mediante `val` la palabra clave también se pueden inicializar en constructores adicionales mediante el uso de expresiones de registro, como se muestra en el código siguiente.

```
type MyStruct =  
  struct  
    val X : int  
    val Y : int  
    val Z : int  
    new(x, y, z) = { X = x; Y = y; Z = z }  
  end  
  
let myStructure1 = new MyStruct(1, 2, 3)
```

Para obtener más información, [vea campos explícitos](#): `val` Palabra clave.

Ejecutar efectos secundarios en constructores

Un constructor principal de una clase puede ejecutar código en un `do` enlace. Sin embargo, ¿qué ocurre si tiene que ejecutar código en un constructor adicional, `do` sin un enlace? Para ello, use la `then` palabra clave.

```
// Executing side effects in the primary constructor and  
// additional constructors.  
type Person(nameIn : string, idIn : int) =  
  let mutable name = nameIn  
  let mutable id = idIn  
  do printfn "Created a person object."  
  member this.Name with get() = name and set(v) = name <- v  
  member this.ID with get() = id and set(v) = id <- v  
  new() =  
    Person("Invalid Name", -1)  
  then  
    printfn "Created an invalid person object."  
  
let person1 = new Person("Humberto Acevedo", 123458734)  
let person2 = new Person()
```

Los efectos secundarios del constructor principal todavía se ejecutan. Por lo tanto, la salida es la siguiente:

```
Created a person object.  
Created a person object.  
Created an invalid person object.
```

Identificadores propios en constructores

En otros miembros, se proporciona un nombre para el objeto actual en la definición de cada miembro. También

puede colocar el identificador propio en la primera línea de la definición de clase mediante la `as` palabra clave inmediatamente después de los parámetros del constructor. En el ejemplo siguiente se muestra esta sintaxis.

```
type MyClass1(x) as this =  
  // This use of the self identifier produces a warning - avoid.  
  let x1 = this.X  
  // This use of the self identifier is acceptable.  
  do printfn "Initializing object with X =%d" this.X  
  member this.X = x
```

En los constructores adicionales, también puede definir un identificador propio colocando la `as` cláusula justo después de los parámetros del constructor. En el ejemplo siguiente se muestra esta sintaxis:

```
type MyClass2(x : int) =  
  member this.X = x  
  new() as this = MyClass2(0) then printfn "Initializing with X = %d" this.X
```

Pueden producirse problemas al intentar usar un objeto antes de que se defina por completo. Por lo tanto, los usos del propio identificador pueden hacer que el compilador emita una advertencia e inserte comprobaciones adicionales para asegurarse de que no se tiene acceso a los miembros de un objeto antes de que se inicialice el objeto. Solo se debe usar el identificador propio en los `do` enlaces del constructor principal o después de la `then` palabra clave en constructores adicionales.

No es necesario que el nombre del propio identificador sea `this`. Puede ser cualquier identificador válido.

Asignar valores a propiedades en la inicialización

Puede asignar valores a las propiedades de un objeto de clase en el código de inicialización anexando una lista de asignaciones del formulario `property = value` a la lista de argumentos de un constructor. Esto se muestra en el ejemplo de código siguiente:

```
type Account() =  
  let mutable balance = 0.0  
  let mutable number = 0  
  let mutable firstName = ""  
  let mutable lastName = ""  
  member this.AccountNumber  
    with get() = number  
    and set(value) = number <- value  
  member this.FirstName  
    with get() = firstName  
    and set(value) = firstName <- value  
  member this.LastName  
    with get() = lastName  
    and set(value) = lastName <- value  
  member this.Balance  
    with get() = balance  
    and set(value) = balance <- value  
  member this.Deposit(amount: float) = this.Balance <- this.Balance + amount  
  member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount  
  
let account1 = new Account(AccountNumber=8782108,  
                           FirstName="Darren", LastName="Parker",  
                           Balance=1543.33)
```

La siguiente versión del código anterior muestra la combinación de argumentos ordinarios, argumentos opcionales y valores de propiedad en una llamada de constructor:

```

type Account(accountNumber : int, ?first: string, ?last: string, ?bal : float) =
  let mutable balance = defaultArg bal 0.0
  let mutable number = accountNumber
  let mutable firstName = defaultArg first ""
  let mutable lastName = defaultArg last ""
  member this.AccountNumber
    with get() = number
    and set(value) = number <- value
  member this.FirstName
    with get() = firstName
    and set(value) = firstName <- value
  member this.LastName
    with get() = lastName
    and set(value) = lastName <- value
  member this.Balance
    with get() = balance
    and set(value) = balance <- value
  member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
  member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount

let account1 = new Account(8782108, bal = 543.33,
                          FirstName="Raman", LastName="Iyer")

```

Constructores en la clase heredada

Al heredar de una clase base que tiene un constructor, debe especificar sus argumentos en la cláusula inherit. Para obtener más información, vea [constructores y herencia](#).

Constructores estáticos o constructores de tipo

Además de especificar el código para crear objetos, se pueden `let` crear `do` enlaces estáticos y en tipos de clase que se ejecutan antes de que el tipo se use por primera vez para realizar la inicialización en el nivel de tipo. Para obtener más información, vea [let enlaces en clases](#) y [do enlaces en clases](#).

Vea también

- [Miembros](#)

Eventos

21/02/2020 • 11 minutes to read • [Edit Online](#)

NOTE

Los vínculos de la referencia de API de este artículo le llevarán a MSDN. La referencia de API de docs.microsoft.com no está completa.

Los eventos permiten asociar las llamadas de función a las acciones del usuario y son importantes para la programación de GUI. Los eventos también los pueden desencadenar las aplicaciones o el sistema operativo.

Controlar eventos

Cuando se utiliza una biblioteca de GUI como Windows Forms o Windows Presentation Foundation (WPF), gran parte del código de la aplicación se ejecuta en respuesta a los eventos predefinidos por la biblioteca. Estos eventos predefinidos son miembros de clases de GUI, como formularios y controles. Se puede agregar comportamiento personalizado a un evento preexistente, como un clic del botón, haciendo referencia al evento con nombre en cuestión (por ejemplo, el evento `Click` de la clase `Form`) e invocando el método `Add`, tal y como se muestra en el código siguiente. Si se ejecuta en F# Interactive, se omitirá la llamada a

```
System.Windows.Forms.Application.Run(System.Windows.Forms.Form) .
```

```
open System.Windows.Forms

let form = new Form(Text="F# Windows Form",
                    Visible = true,
                    TopMost = true)

form.Click.Add(fun evArgs -> System.Console.Beep())
Application.Run(form)
```

El tipo del método `Add` es `('a -> unit) -> unit`. Por consiguiente, el método de control de eventos toma un parámetro, que suele ser los argumentos del evento, y devuelve `unit`. En el ejemplo anterior se muestra el controlador de eventos como una expresión lambda. El controlador de eventos también puede ser un valor de función, como en el ejemplo de código siguiente. En el ejemplo de código siguiente también se muestra el uso de los parámetros de controlador de eventos, que proporcionan información específica del tipo de evento. Para un evento `MouseMove`, el sistema pasa un objeto `System.Windows.Forms.MouseEventArgs`, que contiene la posición `x` e `y` del puntero.

```

open System.Windows.Forms

let Beep evArgs =
    System.Console.Beep( )

let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)

let MouseMoveEventHandler (evArgs : System.Windows.Forms.MouseEventArgs) =
    form.Text <- System.String.Format("{0},{1}", evArgs.X, evArgs.Y)

form.Click.Add(Beep)
form.MouseMove.Add(MouseMoveEventHandler)
Application.Run(form)

```

Crear eventos personalizados

F# los eventos se representan mediante F# la clase de [eventos](#), que implementa la interfaz [IEvent](#). [IEvent](#) es en sí misma una interfaz que combina la funcionalidad de otras dos interfaces, [System.IObservable<'T>](#) e [IDelegateEvent](#). Por consiguiente, la clase [Event](#) tiene una funcionalidad de delegados equivalente a la de otros lenguajes, además de la funcionalidad de [IObservable](#), lo que significa que los eventos de F# admiten el filtrado así como el uso de expresiones lambda y funciones de primera clase de F# como controladores de eventos. Esta funcionalidad se proporciona en el [módulo de eventos](#).

Para crear en una clase un evento que actúe como cualquier otro evento de .NET Framework, agregue a la clase un enlace [let](#) que defina un objeto [Event](#) como un campo de una clase. Puede especificar el tipo de argumento de evento que desee como tipo de argumento o puede dejarlo en blanco y dejar que el compilador infiera el tipo adecuado. Además, debe definir un miembro de evento que exponga el evento como un evento de CLI. Este miembro debe tener el atributo [CLIEvent](#). Se declara como una propiedad y su implementación es simplemente una llamada a la propiedad [Publish](#) del evento. Los usuarios de la clase pueden usar el método [Add](#) del evento publicado para agregar un controlador. El argumento del método [Add](#) puede ser una expresión lambda. Puede usar la propiedad [Trigger](#) del evento para generar el evento pasando los argumentos a la función del controlador. En el siguiente ejemplo código se muestra cómo hacerlo. En este ejemplo, el argumento de tipo inferido para el evento es una tupla, que representa los argumentos de la expresión lambda.

```

open System.Collections.Generic

type MyClassWithCLIEvent() =

    let event1 = new Event<_>()

    [<CLIEvent>]
    member this.Event1 = event1.Publish

    member this.TestEvent(arg) =
        event1.Trigger(this, arg)

let classWithEvent = new MyClassWithCLIEvent()
classWithEvent.Event1.Add(fun (sender, arg) ->
    printfn "Event1 occurred! Object data: %s" arg)

classWithEvent.TestEvent("Hello World!")

System.Console.ReadLine() |> ignore

```

La salida es la siguiente.

```
Event1 occurred! Object data: Hello World!
```

Aquí se muestra la funcionalidad adicional que proporciona el módulo `Event`. En el siguiente ejemplo de código, se muestra el uso básico de `Event.create` para crear un evento y un método desencadenador, agregar dos controladores de eventos en forma de expresiones lambda y, a continuación, desencadenar el evento para ejecutar ambas expresiones lambda.

```
type MyType() =  
  let myEvent = new Event<_>()  
  
  member this.AddHandlers() =  
    Event.add (fun string1 -> printfn "%s" string1) myEvent.Publish  
    Event.add (fun string1 -> printfn "Given a value: %s" string1) myEvent.Publish  
  
  member this.Trigger(message) =  
    myEvent.Trigger(message)  
  
let myMyType = MyType()  
myMyType.AddHandlers()  
myMyType.Trigger("Event occurred.")
```

La salida del código anterior es la siguiente.

```
Event occurred.  
Given a value: Event occurred.
```

Procesar secuencias de eventos

En lugar de agregar simplemente un controlador de eventos para un evento mediante la función [Event.Add](#), puede usar las funciones del módulo `Event` para procesar secuencias de eventos de maneras muy personalizadas. Para ello, se utiliza la canalización hacia delante (`|>`) junto con el evento como primer valor en una serie de llamadas de función, y las funciones del módulo `Event` como llamadas de función subsiguientes.

En el siguiente ejemplo de código, se muestra cómo configurar un evento cuyo controlador se invoca únicamente en determinadas condiciones.

```
let form = new Form(Text = "F# Windows Form",  
                    Visible = true,  
                    TopMost = true)  
  
form.MouseMove  
  |> Event.filter ( fun evArgs -> evArgs.X > 100 && evArgs.Y > 100)  
  |> Event.add ( fun evArgs ->  
    form.BackColor <- System.Drawing.Color.FromArgb(  
      evArgs.X, evArgs.Y, evArgs.X ^^ evArgs.Y ) )
```

El [módulo observable](#) contiene funciones similares que operan en objetos observables. Los objetos observables son similares a los eventos pero solo se suscriben activamente a los eventos si ellos mismos son objeto de una suscripción.

Implementar un evento de interfaz

Cuando se desarrollan componentes de interfaz de usuario, suele empezarse por crear un nuevo formulario o control que herede de un formulario o control ya existente. A menudo los eventos se definen en una interfaz y, en ese caso, debe implementar la interfaz para implementar el evento. La interfaz

`System.ComponentModel.INotifyPropertyChanged` define un único evento

`System.ComponentModel.INotifyPropertyChanged.PropertyChanged` . En el código siguiente se muestra cómo implementar el evento definido por esta interfaz heredada:

```
module CustomForm

open System.Windows.Forms
open System.ComponentModel

type AppForm() as this =
    inherit Form()

    // Define the PropertyChanged event.
    let PropertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs ->
            this.Property1 <- "text2"
            this.Property2 <- "text3")

    // This property does not have the property-changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property-changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            PropertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    // Expose the PropertyChanged event as a first class .NET event.
    [<CLIEvent>]
    member this.PropertyChanged = PropertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = PropertyChanged.Publish.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = PropertyChanged.Publish.RemoveHandler(handler)

    // This is the event-handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property %s changed its value to %s" args.PropertyName newValue

    // Create a form, hook up the event handler, and start the application.
    let appForm = new AppForm()
    let inpc = appForm :> INotifyPropertyChanged
    inpc.PropertyChanged.Add(appForm.OnPropertyChanged)
    Application.Run(appForm)
```

Si desea enlazar el evento en el constructor, el código es algo más complejo porque el enlace del evento debe estar en un bloque `then` de un constructor adicional, como en el ejemplo siguiente:

```

module CustomForm

open System.Windows.Forms
open System.ComponentModel

// Create a private constructor with a dummy argument so that the public
// constructor can have no arguments.
type AppForm private (dummy) as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs ->
            this.Property1 <- "text2"
            this.Property2 <- "text3")

    // This property does not have the property changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

[<CLIEvent>]
member this.PropertyChanged = propertyChanged.Publish

// Define the add and remove methods to implement this interface.
interface INotifyPropertyChanged with
    member this.add_PropertyChanged(handler) = this.PropertyChanged.AddHandler(handler)
    member this.remove_PropertyChanged(handler) = this.PropertyChanged.RemoveHandler(handler)

// This is the event handler method.
member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
    let newProperty = this.GetType().GetProperty(args.PropertyName)
    let newValue = newProperty.GetValue(this :> obj) :?> string
    printfn "Property %s changed its value to %s" args.PropertyName newValue

new() as this =
    new AppForm(0)
    then
        let inpc = this :> INotifyPropertyChanged
        inpc.PropertyChanged.Add(this.OnPropertyChanged)

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
Application.Run(appForm)

```

Consulte también

- [Miembros](#)
- [Control y generación de eventos](#)
- [Expresiones lambda: palabra clave](#) `fun`
- [Control.event \(módulo\)](#)
- [Control.event<\(> clase\)](#)
- [Control.event<' Delegate, '> args \(clase\)](#)

Campos explícitos: La palabra clave Val

23/10/2019 • 8 minutes to read • [Edit Online](#)

La palabra clave `val` se usa para declarar una ubicación para almacenar un valor en un tipo de clase o estructura sin inicializarlo. Las ubicaciones de almacenamiento declaradas de esta manera se denominan *campos explícitos*. Otra manera de usar la palabra clave `val` es conjuntamente con la palabra clave `member` para declarar una propiedad implementada automáticamente. Para obtener más información sobre las propiedades implementadas automáticamente, vea [propiedades](#).

Sintaxis

```
val [ mutable ] [ access-modifier ] field-name : type-name
```

Comentarios

La forma más habitual de definir campos en un tipo de clase o estructura es usar un enlace `let`. Sin embargo, los enlaces `let` deben inicializarse como parte del constructor de clase, lo que no siempre es posible, necesario o deseable. Puede usar la palabra clave `val` cuando quiera un campo que no se haya inicializado.

Los campos explícitos pueden ser estáticos o no estáticos. El *modificador de acceso* puede ser `public`, `private` o `internal`. De forma predeterminada, los campos explícitos son públicos. Esto difiere de los enlaces `let` de las clases, que siempre son privados.

El atributo [DefaultValue](#) es necesario en los campos explícitos de los tipos de clase que tienen un constructor primario. Este atributo especifica que el campo se inicializa en cero. El tipo del campo debe admitir la inicialización en cero. Los tipos que admiten la inicialización en cero son los siguientes:

- Un tipo primitivo que tenga un valor de cero.
- Un tipo que admita un valor nulo, ya sea como valor normal, como valor anormal o como representación de un valor. Esto incluye clases, tuplas, registros, funciones, interfaces, tipos de referencia .NET, el tipo `unit` y tipos de unión discriminada.
- Un tipo de valor .NET.
- Una estructura cuyos campos admitan el valor cero predeterminado.

Por ejemplo, un campo inmutable denominado `someField` tiene un campo de respaldo en la representación compilada de .NET con el nombre `someField@` y el usuario accede al valor almacenado con una propiedad denominada `someField`.

Para un campo mutable, la representación compilada de .NET es un campo .NET.

WARNING

El espacio de nombres `System.ComponentModel` .NET Framework contiene un atributo con el mismo nombre. Para obtener más información sobre el atributo, vea [DefaultValueAttribute](#).

El código siguiente muestra el uso de campos explícitos y, para la comparación, un `let` de enlace en una clase que tiene un constructor primario. Tenga en cuenta que el campo enlazado a `let`myInt1` es privado. Si se hace referencia al campo enlazado a `let`myInt1` desde un método de miembro, el identificador propio `this` no es

necesario. Pero si hace referencia a los campos explícitos `myInt2` y `myString`, se requiere el identificador propio.

```
type MyType() =
  let mutable myInt1 = 10
  [<DefaultValue>] val mutable myInt2 : int
  [<DefaultValue>] val mutable myString : string
  member this.SetValsAndPrint( i: int, str: string) =
    myInt1 <- i
    this.myInt2 <- i + 1
    this.myString <- str
    printfn "%d %d %s" myInt1 (this.myInt2) (this.myString)

let myObject = new MyType()
myObject.SetValsAndPrint(11, "abc")
// The following line is not allowed because let bindings are private.
// myObject.myInt1 <- 20
myObject.myInt2 <- 30
myObject.myString <- "def"

printfn "%d %s" (myObject.myInt2) (myObject.myString)
```

La salida es la siguiente:

```
11 12 abc
30 def
```

En el código siguiente se muestra el uso de campos explícitos en una clase que no tiene constructor primario. En este caso, el atributo `DefaultValue` no es necesario, pero todos los campos deben estar inicializados en los constructores definidos para el tipo.

```
type MyClass =
  val a : int
  val b : int
  // The following version of the constructor is an error
  // because b is not initialized.
  // new (a0, b0) = { a = a0; }
  // The following version is acceptable because all fields are initialized.
  new(a0, b0) = { a = a0; b = b0; }

let myClassObj = new MyClass(35, 22)
printfn "%d %d" (myClassObj.a) (myClassObj.b)
```

El resultado es `35 22`

En el código siguiente se muestra el uso de campos explícitos en una estructura. Dado que una estructura es un tipo de valor, tiene automáticamente un constructor sin parámetros que establece los valores de sus campos en cero. Por lo tanto, el atributo `DefaultValue` no es necesario.

```
type MyStruct =
  struct
    val mutable myInt : int
    val mutable myString : string
  end

let mutable myStructObj = new MyStruct()
myStructObj.myInt <- 11
myStructObj.myString <- "xyz"

printfn "%d %s" (myStructObj.myInt) (myStructObj.myString)
```

El resultado es `11 xyz`

Tenga **cuidado**, si va a inicializar la estructura con `mutable` campos sin `mutable` palabra clave, las asignaciones funcionarán en una copia de la estructura que se descartará justo después de la asignación. Por lo tanto, la estructura no cambiará.

```
[<Struct>]
type Foo =
    val mutable bar: string
    member self.ChangeBar bar = self.bar <- bar
    new (bar) = {bar = bar}

let foo = Foo "1"
foo.ChangeBar "2" //make implicit copy of Foo, changes the copy, discards the copy, foo remains unchanged
printfn "%s" foo.bar //prints 1

let mutable foo' = Foo "1"
foo'.ChangeBar "2" //changes foo'
printfn "%s" foo'.bar //prints 2
```

Los campos explícitos no están pensados para un uso rutinario. En general, siempre que sea posible debe usar un enlace `let` en una clase en lugar de un campo explícito. Los campos explícitos son útiles en determinados escenarios de interoperabilidad, como cuando se necesita definir una estructura que se usará en una llamada de invocación a una plataforma API nativa o en escenarios de interoperabilidad COM. Para obtener más información, vea [funciones externas](#). Otra situación en la que podría ser necesario un campo explícito es cuando se trabaja con un generador de código F # que emite clases sin un constructor primario. Los campos explícitos también son útiles para las variables de subproceso estático o construcciones similares. Para obtener más información, consulta `System.ThreadStaticAttribute`.

Cuando las palabras clave `member val` aparecen juntas en una definición de tipo, se trata de la definición de una propiedad implementada automáticamente. Para obtener más información, consulta [Propiedades](#).

Vea también

- [Propiedades](#)
- [Miembros](#)
- `let` [Bindings in Classes](#) (Enlaces `let` en clases)

Extensiones de tipo

10/02/2020 • 10 minutes to read • [Edit Online](#)

Las extensiones de tipo (también denominadas *aumentos*) son una familia de características que permiten agregar nuevos miembros a un tipo de objeto definido previamente. Las tres características son:

- Extensiones de tipo intrínsecas
- Extensiones de tipo opcionales
- Métodos de extensión

Cada se puede usar en escenarios diferentes y tiene diferentes inconvenientes.

Sintaxis

```
// Intrinsic and optional extensions
type typename with
    member self-identifier.member-name =
        body
    ...

// Extension methods
open System.Runtime.CompilerServices

[<Extension>]
type Extensions() =
    [<Extension>]
    static member self-identifier.extension-name (ty: typename, [args]) =
        body
    ...
```

Extensiones de tipo intrínsecas

Una extensión de tipo intrínseco es una extensión de tipo que extiende un tipo definido por el usuario.

Las extensiones de tipo intrínseco se deben definir en el mismo archivo y en el mismo espacio de nombres o módulo que el tipo que están extendiendo. Cualquier otra definición dará como resultado extensiones de [tipo opcionales](#).

Las extensiones de tipo intrínseco a veces son una forma más limpia de separar la funcionalidad de la declaración de tipos. En el ejemplo siguiente se muestra cómo definir una extensión de tipo intrínseco:

```

namespace Example

type Variant =
    | Num of int
    | Str of string

module Variant =
    let print v =
        match v with
        | Num n -> printf "Num %d" n
        | Str s -> printf "Str %s" s

// Add a member to Variant as an extension
type Variant with
    member x.Print() = Variant.print x

```

El uso de una extensión de tipo le permite separar cada uno de los elementos siguientes:

- Declaración de un tipo de `Variant`
- Funcionalidad para imprimir la clase `Variant` en función de su "forma"
- Una manera de tener acceso a la funcionalidad de impresión con la notación de `.` de estilo de objeto

Esta es una alternativa a la definición de todo como miembro en `Variant`. Aunque no es un enfoque bastante mejor, puede ser una representación más limpia de la funcionalidad en algunas situaciones.

Las extensiones de tipo intrínseco se compilan como miembros del tipo que aumentan y aparecen en el tipo cuando la reflexión examina el tipo.

Extensiones de tipo opcionales

Una extensión de tipo opcional es una extensión que aparece fuera del módulo, espacio de nombres o ensamblado original del tipo que se va a extender.

Las extensiones de tipo opcionales son útiles para extender un tipo que no se ha definido. Por ejemplo:

```

module Extensions

type IEnumerable<'T> with
    /// Repeat each element of the sequence n times
    member xs.RepeatElements(n: int) =
        seq {
            for x in xs do
                for _ in 1 .. n -> x
        }

```

Ahora puede tener acceso a `RepeatElements` como si es miembro de `IEnumerable<T>` siempre que el módulo de `Extensions` se abra en el ámbito en el que está trabajando.

Las extensiones opcionales no aparecen en el tipo extendido cuando se examinan por reflexión. Las extensiones opcionales deben estar en módulos y solo están en el ámbito cuando el módulo que contiene la extensión está abierto o está en el ámbito de otra forma.

Los miembros de extensión opcionales se compilan en miembros estáticos para los que la instancia de objeto se pasa implícitamente como el primer parámetro. Sin embargo, actúan como si fueran miembros de instancia o miembros estáticos según cómo se declaran.

Los miembros de extensión opcionales tampoco son visibles C# para los consumidores o Visual Basic. Solo se pueden consumir en otro F# código.

Limitación genérica de las extensiones de tipo intrínsecas y opcionales

Es posible declarar una extensión de tipo en un tipo genérico en el que la variable de tipo está restringida. El requisito es que la restricción de la declaración de extensión coincide con la restricción del tipo declarado.

Sin embargo, incluso cuando se hace coincidir las restricciones entre un tipo declarado y una extensión de tipo, es posible que una restricción se infiera por el cuerpo de un miembro extendido que impone un requisito diferente en el parámetro de tipo que el tipo declarado. Por ejemplo:

```
open System.Collections.Generic

// NOT POSSIBLE AND FAILS TO COMPILE!
//
// The member 'Sum' has a different requirement on 'T' than the type IEnumerable<'T>
type IEnumerable<'T> with
    member this.Sum() = Seq.sum this
```

No hay ninguna manera de obtener este código para trabajar con una extensión de tipo opcional:

- Tal y como está, el miembro de `Sum` tiene una restricción diferente en `'T` (`static member get_Zero` y `static member (+)`) de la que define la extensión de tipo.
- La modificación de la extensión de tipo para que tenga la misma restricción que `Sum` ya no coincidirá con la restricción definida en `IEnumerable<'T>`.
- Al cambiar `member this.Sum` a `member inline this.Sum`, se producirá un error que indica que las restricciones de tipo no coinciden.

Lo que se desea son los métodos estáticos que "flotan espacio" y se pueden presentar como si estuvieran extendiendo un tipo. Aquí es donde los métodos de extensión son necesarios.

Métodos de extensión

Por último, los métodos de extensión (C# a veces denominados "miembros de extensión de F# estilo") se pueden declarar en como un método de miembro estático en una clase.

Los métodos de extensión son útiles cuando se desea definir extensiones en un tipo genérico que restrinja la variable de tipo. Por ejemplo:

```
namespace Extensions

open System.Runtime.CompilerServices

[<Extension>]
type IEnumerableExtensions =
    [<Extension>]
    static member inline Sum(xs: IEnumerable<'T>) = Seq.sum xs
```

Cuando se usa, este código aparecerá como si `Sum` se define en `IEnumerable<T>`, siempre y cuando `Extensions` se haya abierto o esté en el ámbito.

Para que la extensión esté disponible para el código VB.NET, se requiere un `ExtensionAttribute` adicional en el nivel de ensamblado:

```
module AssemblyInfo
open System.Runtime.CompilerServices
[<assembly:Extension>]
do ()
```


Otros comentarios

Las extensiones de tipo también tienen los siguientes atributos:

- Se puede extender cualquier tipo al que se pueda tener acceso.
- Las extensiones de tipo intrínseca y opcional pueden definir *cualquier* tipo de miembro, no solo los métodos. Por ejemplo, las propiedades de extensión también son posibles.
- El token de `self-identifier` en la [Sintaxis](#) representa la instancia del tipo que se invoca, al igual que los miembros ordinarios.
- Los miembros extendidos pueden ser miembros estáticos o de instancia.
- Las variables de tipo en una extensión de tipo deben coincidir con las restricciones del tipo declarado.

También existen las siguientes limitaciones para las extensiones de tipo:

- Las extensiones de tipo no admiten métodos virtuales o abstractos.
- Las extensiones de tipo no admiten métodos de invalidación como aumentos.
- Las extensiones de tipo no admiten [parámetros de tipo resueltos estáticamente](#).
- Las extensiones de tipo opcionales no admiten constructores como aumentos.
- Las extensiones de tipo no se pueden definir en las [abreviaturas de tipo](#).
- Las extensiones de tipo no son válidas para `byref<'T>` (aunque se pueden declarar).
- Las extensiones de tipo no son válidas para los atributos (aunque se pueden declarar).
- Puede definir extensiones que sobrecarguen otros métodos del mismo nombre, pero el F# compilador da preferencia a los métodos que no son de extensión si hay una llamada ambigua.

Por último, si existen varias extensiones de tipo intrínsecas para un tipo, todos los miembros deben ser únicos. En el caso de las extensiones de tipo opcionales, los miembros de diferentes extensiones de tipo al mismo tipo pueden tener los mismos nombres. Los errores de ambigüedad solo se producen si el código de cliente abre dos ámbitos diferentes que definen los mismos nombres de miembro.

Consulte también

- [Referencia del lenguaje F#](#)
- [Miembros](#)

Parámetros y argumentos

19/03/2020 • 18 minutes to read • [Edit Online](#)

En este tema se describe la compatibilidad del lenguaje para definir parámetros y pasar argumentos a funciones, métodos y propiedades. Incluye información sobre cómo pasar por referencia y cómo definir y usar métodos que pueden tomar un número variable de argumentos.

Parámetros y argumentos

El *parámetro* term se utiliza para describir los nombres de los valores que se espera que se suminisen. El *término argumento* se utiliza para los valores proporcionados para cada parámetro.

Los parámetros se pueden especificar en forma de tupla o cuajada, o en alguna combinación de los dos. Puede pasar argumentos mediante un nombre de parámetro explícito. Los parámetros de los métodos se pueden especificar como opcionales y se les da un valor predeterminado.

Patrones de parámetros

Los parámetros proporcionados a funciones y métodos son, en general, patrones separados por espacios. Esto significa que, en principio, cualquiera de los patrones descritos en [Expresiones](#) de coincidencia se puede utilizar en una lista de parámetros para una función o miembro.

Los métodos suelen utilizar la forma de tupla de pasar argumentos. Esto logra un resultado más claro desde la perspectiva de otros lenguajes .NET porque el formulario de tupla coincide con la forma en que se pasan los argumentos en los métodos .NET.

La forma curried se utiliza con `let` mayor frecuencia con funciones creadas mediante enlaces.

El siguiente pseudocódigo muestra ejemplos de tupla y argumentos currudos.

```
// Tuple form.
member this.SomeMethod(param1, param2) = ...
// Curried form.
let function1 param1 param2 = ...
```

Los formularios combinados son posibles cuando algunos argumentos están en tuplas y otros no.

```
let function2 param1 (param2a, param2b) param3 = ...
```

Otros patrones también se pueden utilizar en las listas de parámetros, pero si el patrón de parámetros no coincide con todas las entradas posibles, puede haber una coincidencia incompleta en tiempo de ejecución. La

`MatchFailureException` excepción se genera cuando el valor de un argumento no coincide con los patrones especificados en la lista de parámetros. El compilador emite una advertencia cuando un patrón de parámetros permite coincidencias incompletas. Al menos otro patrón suele ser útil para las listas de parámetros, y ese es el patrón de comodín. Utilice el patrón de comodín en una lista de parámetros cuando simplemente desee omitir los argumentos que se proporcionan. El código siguiente ilustra el uso del patrón de comodín en una lista de argumentos.

```
let makeList _ = [ for i in 1 .. 100 -> i * i ]
// The arguments 100 and 200 are ignored.
let list1 = makeList 100
let list2 = makeList 200
```

El patrón de comodín puede ser útil siempre que no necesite los argumentos pasados, como en el punto de entrada principal a un programa, cuando no está interesado en los argumentos de línea de comandos que normalmente se proporcionan como una matriz de cadenas, como en el código siguiente.

```
[<EntryPoint>]
let main _ =
    printfn "Entry point!"
    0
```

Otros patrones que a veces se `as` usan en argumentos son el patrón y los patrones de identificador asociados con uniones discriminadas y patrones activos. Puede utilizar el patrón de unión discriminada de un solo caso de la siguiente manera.

```
type Slice = Slice of int * int * string

let GetSubstring1 (Slice(p0, p1, text)) =
    printfn "Data begins at %d and ends at %d in string %s" p0 p1 text
    text.[p0..p1]

let substring = GetSubstring1 (Slice(0, 4, "Et tu, Brute?"))
printfn "Substring: %s" substring
```

La salida es la siguiente.

```
Data begins at 0 and ends at 4 in string Et tu, Brute?
Et tu
```

Los patrones activos pueden ser útiles como parámetros, por ejemplo, al transformar un argumento en un formato deseado, como en el ejemplo siguiente:

```
type Point = { x : float; y : float }

let (| Polar |) { x = x; y = y } =
    ( sqrt (x*x + y*y), System.Math.Atan (y/ x) )

let radius (Polar(r, _)) = r
let angle (Polar(_, theta)) = theta
```

Puede usar `as` el patrón para almacenar un valor coincidente como un valor local, como se muestra en la siguiente línea de código.

```
let GetSubstring2 (Slice(p0, p1, text) as s) = s
```

Otro patrón que se usa ocasionalmente es una función que deja el último argumento sin nombre proporcionando, como el cuerpo de la función, una expresión lambda que realiza inmediatamente una coincidencia de patrón en el argumento implícito. Un ejemplo de esto es la siguiente línea de código.

```
let isNil = function [] -> true | _::_ -> false
```

Este código define una función que `true` toma una lista `false` genérica y devuelve si la lista está vacía y, en caso contrario. El uso de estas técnicas puede hacer que el código sea más difícil de leer.

Ocasionalmente, los patrones que implican coincidencias incompletas son útiles, por ejemplo, si sabe que las listas del programa tienen solo tres elementos, puede usar un patrón como el siguiente en una lista de parámetros.

```
let sum [a; b; c;] = a + b + c
```

El uso de patrones que tienen coincidencias incompletas se reserva mejor para la creación rápida de prototipos y otros usos temporales. El compilador emitirá una advertencia para dicho código. Estos patrones no pueden abarcar el caso general de todas las entradas posibles y, por lo tanto, no son adecuados para las API de componentes.

Argumentos con nombre

Los argumentos de los métodos se pueden especificar por posición en una lista de argumentos separados por comas, o se pueden pasar a un método explícitamente proporcionando el nombre, seguido de un signo igual y el valor que se va a pasar. Si se especifica proporcionando el nombre, pueden aparecer en un orden diferente del utilizado en la declaración.

Los argumentos con nombre pueden hacer que el código sea más legible y más adaptable a ciertos tipos de cambios en la API, como un reordenamiento de los parámetros del método.

Los argumentos con nombre solo `let` se permiten para métodos, no para funciones enlazadas, valores de función o expresiones lambda.

En el ejemplo de código siguiente se muestra el uso de argumentos con nombre.

```
type SpeedingTicket() =
    member this.GetMPHOver(speed: int, limit: int) = speed - limit

let CalculateFine (ticket : SpeedingTicket) =
    let delta = ticket.GetMPHOver(limit = 55, speed = 70)
    if delta < 20 then 50.0 else 100.0

let ticket1 : SpeedingTicket = SpeedingTicket()
printfn "%f" (CalculateFine ticket1)
```

En una llamada a un constructor de clase, puede establecer los valores de las propiedades de la clase mediante una sintaxis similar a la de los argumentos con nombre. En el ejemplo siguiente se muestra esta sintaxis.

```

type Account() =
  let mutable balance = 0.0
  let mutable number = 0
  let mutable firstName = ""
  let mutable lastName = ""
  member this.AccountNumber
    with get() = number
    and set(value) = number <- value
  member this.FirstName
    with get() = firstName
    and set(value) = firstName <- value
  member this.LastName
    with get() = lastName
    and set(value) = lastName <- value
  member this.Balance
    with get() = balance
    and set(value) = balance <- value
  member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
  member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount

let account1 = new Account(AccountNumber=8782108,
                             FirstName="Darren", LastName="Parker",
                             Balance=1543.33)

```

Para obtener más información, vea [Constructores \(F-\)](#).

Parámetros opcionales

Puede especificar un parámetro opcional para un método mediante un signo de interrogación delante del nombre del parámetro. Los parámetros opcionales se interpretan como el tipo de opción de F, por lo `match` que `Some` `None` puede consultarlos de la manera regular en que se consultan los tipos de opción, mediante una expresión con `y`. Los parámetros opcionales solo se permiten `let` en los miembros, no en las funciones creadas mediante enlaces.

Puede pasar valores opcionales existentes al método `?arg=None` `?arg=Some(3)` por `?arg=arg` nombre de parámetro, como `o`. Esto puede ser útil al compilar un método que pasa argumentos opcionales a otro método.

También puede utilizar `defaultArg` una función, que establece un valor predeterminado de un argumento opcional. La `defaultArg` función toma el parámetro opcional como el primer argumento y el valor predeterminado como el segundo.

En el ejemplo siguiente se muestra el uso de parámetros opcionales.

```

type DuplexType =
    | Full
    | Half

type Connection(?rate0 : int, ?duplex0 : DuplexType, ?parity0 : bool) =
    let duplex = defaultArg duplex0 Full
    let parity = defaultArg parity0 false
    let mutable rate = match rate0 with
        | Some rate1 -> rate1
        | None -> match duplex with
            | Full -> 9600
            | Half -> 4800
    do printfn "Baud Rate: %d Duplex: %A Parity: %b" rate duplex parity

let conn1 = Connection(duplex0 = Full)
let conn2 = Connection(duplex0 = Half)
let conn3 = Connection(300, Half, true)
let conn4 = Connection(?duplex0 = None)
let conn5 = Connection(?duplex0 = Some(Full))

let optionalDuplexValue : option<DuplexType> = Some(Half)
let conn6 = Connection(?duplex0 = optionalDuplexValue)

```

La salida es la siguiente.

```

Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
Baud Rate: 300 Duplex: Half Parity: true
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false

```

Para los fines de la interoperabilidad de C- y Visual Basic, puede usar los atributos

[<Optional; DefaultValue<...>] en F#, de modo que los llamadores verán un argumento como opcional. Esto equivale a definir el argumento como opcional `MyMethod(int i = 3)` en C- como en .

```

open System
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultValue("Hello world")>] message) =
        printfn "%s" message

```

También puede especificar un nuevo objeto como valor de parámetro predeterminado. Por ejemplo, `Foo` el miembro `CancellationToken` podría tener un opcional como entrada en su lugar:

```

open System.Threading
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultValue(CancellationToken())>] ct: CancellationToken) =
        printfn "%A" ct

```

El valor dado `DefaultValue` como argumento para debe coincidir con el tipo del parámetro. Por ejemplo, no se permite lo siguiente:

```

type C =
    static member Wrong([<Optional; DefaultValue("string")>] i:int) = ()

```

En este caso, el compilador genera una advertencia e ignorará ambos atributos por completo. Tenga en cuenta

`null` que el valor predeterminado debe anotarse con el tipo, ya que de lo contrario el compilador deduce el tipo incorrecto, es decir, `[<Optional; DefaultParameterValue(null:obj)>] o:obj`.

Pasando por referencia

Pasar un valor de F por referencia implica [byrefs](#), que son tipos de puntero administrado. Las instrucciones para qué tipo utilizar son las siguientes:

- Utilícelo `inref<'T>` si solo necesita leer el puntero.
- Utilícelo `outref<'T>` si solo necesita escribir en el puntero.
- Utilícelo `byref<'T>` si necesita leer y escribir en el puntero.

```
let example1 (x: inref<int>) = printfn "It's %d" x

let example2 (x: outref<int>) = x <- x + 1

let example3 (x: byref<int>) =
    printfn "It'd %d" x
    x <- x + 1

let test () =
    // No need to make it mutable, since it's read-only
    let x = 1
    example1 &x

    // Needs to be mutable, since we write to it
    let mutable y = 2
    example2 &y
    example3 &y // Now 'y' is 3
```

Dado que el parámetro es un puntero y el valor es mutable, los cambios en el valor se conservan después de la ejecución de la función.

Puede usar una tupla como valor `out` devuelto para almacenar cualquier parámetro en los métodos de biblioteca de .NET. Como alternativa, puede `out` tratar el `byref` parámetro como un parámetro. En el ejemplo de código siguiente se muestran ambas formas.

```
// TryParse has a second parameter that is an out parameter
// of type System.DateTime.
let (b, dt) = System.DateTime.TryParse("12-20-04 12:21:00")

printfn "%b %A" b dt

// The same call, using an address of operator.
let mutable dt2 = System.DateTime.Now
let b2 = System.DateTime.TryParse("12-20-04 12:21:00", &dt2)

printfn "%b %A" b2 dt2
```

Matrices de parámetros

Ocasionalmente es necesario definir una función que toma un número arbitrario de parámetros de tipo heterogéneo. No sería práctico crear todos los métodos sobrecargados posibles para tener en cuenta todos los tipos que se podrían usar. Las implementaciones de .NET proporcionan compatibilidad con estos métodos a través de la característica de matriz de parámetros. Un método que toma una matriz de parámetros en su firma se puede proporcionar con un número arbitrario de parámetros. Los parámetros se colocan en una matriz. El tipo de los elementos de matriz determina los tipos de parámetro que se pueden pasar a la función. Si define la matriz

`System.Object` de parámetros con como el tipo de elemento, el código de cliente puede pasar valores de cualquier tipo.

En F, las matrices de parámetros solo se pueden definir en métodos. No se pueden utilizar en funciones independientes o funciones definidas en módulos.

Defina una matriz de `ParamArray` parámetros mediante el atributo. El `ParamArray` atributo solo se puede aplicar al último parámetro.

En el código siguiente se muestra tanto la llamada a un método .NET que toma una matriz de parámetros como la definición de un tipo en F- que tiene un método que toma una matriz de parámetros.

```
open System

type X() =
    member this.F([<ParamArray>] args: Object[]) =
        for arg in args do
            printfn "%A" arg

[<EntryPoint>]
let main _ =
    // call a .NET method that takes a parameter array, passing values of various types
    Console.WriteLine("a {0} {1} {2} {3} {4}", 1, 10.0, "Hello world", 1u, true)

    let xobj = new X()
    // call an F# method that takes a parameter array, passing values of various types
    xobj.F("a", 1, 10.0, "Hello world", 1u, true)
    0
```

Cuando se ejecuta en un proyecto, la salida del código anterior es la siguiente:

```
a 1 10 Hello world 1 True
"a"
1
10.0
"Hello world"
1u
true
```

Consulte también

- [Miembros](#)

Sobrecarga de operadores

23/10/2019 • 11 minutes to read • [Edit Online](#)

En este tema se describe cómo sobrecargar los operadores aritméticos de un tipo de clase o registro, así como en el nivel global.

Sintaxis

```
// Overloading an operator as a class or record member.  
static member (operator-symbols) (parameter-list) =  
    method-body  
  
// Overloading an operator at the global level  
let [inline] (operator-symbols) parameter-list = function-body
```

Comentarios

En la sintaxis anterior, el *símbolo del operador* es uno de `+`, `-`, `*`, `/`, `=`, etc. La *lista de parámetros* especifica los operandos en el orden en que aparecen en la sintaxis habitual de ese operador. El *cuerpo del método* crea el valor resultante.

Las sobrecargas de operador para los operadores deben ser estáticas. Las sobrecargas de operador para los operadores unarios `-`, `+` como y, deben usar `~` una tilde () en el *símbolo del operador* para indicar que el operador es un operador unario y no un operador binario, como se muestra a continuación relativa.

```
static member (~-) (v : Vector)
```

En el código siguiente se muestra una clase vectorial que solo tiene dos operadores, uno para el unario menos y uno para la multiplicación por un escalar. En el ejemplo, se necesitan dos sobrecargas para la multiplicación escalar porque el operador debe funcionar con independencia del orden en que aparezcan el vector y el escalar.

```
type Vector(x: float, y : float) =  
    member this.x = x  
    member this.y = y  
    static member (~-) (v : Vector) =  
        Vector(-1.0 * v.x, -1.0 * v.y)  
    static member (*) (v : Vector, a) =  
        Vector(a * v.x, a * v.y)  
    static member (*) (a, v: Vector) =  
        Vector(a * v.x, a * v.y)  
    override this.ToString() =  
        this.x.ToString() + " " + this.y.ToString()  
  
let v1 = Vector(1.0, 2.0)  
  
let v2 = v1 * 2.0  
let v3 = 2.0 * v1  
  
let v4 = - v2  
  
printfn "%s" (v1.ToString())  
printfn "%s" (v2.ToString())  
printfn "%s" (v3.ToString())  
printfn "%s" (v4.ToString())
```

Crear operadores nuevos

Se pueden sobrecargar todos los operadores estándar, pero también se pueden crear otros nuevos mediante secuencias de determinados caracteres. Los caracteres de operador permitidos son `*`, `+`, `.`, `"`, `=`, `-`, `/`, `>`, `<`, `%`, `?`, `,`, `@`, `,`, `^` y `~`. El carácter `~` tiene el significado especial de convertir en unario un operador y no forma parte de la secuencia de caracteres de operador. No todos los operadores se pueden convertir en unario.

Según la secuencia de caracteres exacta utilizada, el operador tendrá una prioridad y una asociatividad determinadas. La asociatividad puede ser de izquierda a derecha o de derecha a izquierda, y se utiliza siempre que aparecen en secuencia y sin paréntesis operadores que tienen el mismo nivel de prioridad.

El carácter operador `.` no afecta a la prioridad, de modo que, por ejemplo, para definir una versión propia de multiplicación que tenga la misma prioridad y asociatividad que la multiplicación ordinaria, se pueden crear operadores tales como `.*`.

Solo los operadores `?` y `?<-` pueden comenzar por `?`.

Una tabla que muestra la prioridad de todos los operadores F# de se puede encontrar en la [referencia de símbolos y operadores](#).

Nombres de operador sobrecargados

Cuando el compilador de F# compila una expresión de operador, genera un método que tiene un nombre generado por compilador para ese operador. Este es el nombre que aparece en el Lenguaje Intermedio de Microsoft (MSIL) para el método y también en reflexión e IntelliSense. Normalmente, no es necesario utilizar estos nombres en el código de F#.

En la tabla siguiente se muestran los operadores estándar y sus nombres generados correspondientes.

OPERADOR	NOMBRE GENERADO
<code>[]</code>	<code>op_Nil</code>
<code>::</code>	<code>op_Cons</code>
<code>+</code>	<code>op_Addition</code>
<code>-</code>	<code>op_Subtraction</code>
<code>*</code>	<code>op_Multiply</code>
<code>/</code>	<code>op_Division</code>
<code>@</code>	<code>op_Append</code>
<code>^</code>	<code>op_Concatenate</code>
<code>%</code>	<code>op_Modulus</code>
<code>&&&</code>	<code>op_BitwiseAnd</code>
<code> </code>	<code>op_BitwiseOr</code>

OPERADOR	NOMBRE GENERADO
^^^	op_ExclusiveOr
<<<	op_LeftShift
~~~	op_LogicalNot
>>>	op_RightShift
~+	op_UnaryPlus
~-	op_UnaryNegation
=	op_Equality
<=	op_LessThanOrEqual
>=	op_GreaterThanOrEqual
<	op_LessThan
>	op_GreaterThan
?	op_Dynamic
?<-	op_DynamicAssignment
>	op_PipeRight
<	op_PipeLeft
!	op_Dereference
>>	op_ComposeRight
<<	op_ComposeLeft
<@ @>	op_Quotation
<@@ @@@>	op_QuotationUntyped
+=	op_AdditionAssignment
-=	op_SubtractionAssignment
*=	op_MultiplyAssignment
/=	op_DivisionAssignment

OPERADOR	NOMBRE GENERADO
<code>..</code>	<code>op_Range</code>
<code>.. ..</code>	<code>op_RangeStep</code>

Hay otras combinaciones de caracteres de operador que no se muestran en este texto y que se pueden utilizar como operadores; sus nombres se crean a partir de la concatenación de los nombres de los caracteres individuales según la tabla siguiente. Por ejemplo, `+!` se `op_PlusBang` convierte en.

CARÁCTER DE OPERADOR	NAME
<code>&gt;</code>	<code>Greater</code>
<code>&lt;</code>	<code>Less</code>
<code>+</code>	<code>Plus</code>
<code>-</code>	<code>Minus</code>
<code>*</code>	<code>Multiply</code>
<code>/</code>	<code>Divide</code>
<code>=</code>	<code>Equals</code>
<code>~</code>	<code>Twiddle</code>
<code>%</code>	<code>Percent</code>
<code>.</code>	<code>Dot</code>
<code>&amp;</code>	<code>Amp</code>
<code> </code>	<code>Bar</code>
<code>@</code>	<code>At</code>
<code>^</code>	<code>Hat</code>
<code>!</code>	<code>Bang</code>
<code>?</code>	<code>Qmark</code>
<code>(</code>	<code>LParen</code>
<code>,</code>	<code>Comma</code>
<code>)</code>	<code>RParen</code>

CARÁCTER DE OPERADOR	NAME
[	LBrack
]	RBrack

## Operadores de prefijo e infijo

Se espera que los operadores de *prefijo* se colocan delante de los operandos u operandos, de forma muy parecida a una función. Se espera que los operadores de *infijo* se colocan entre los dos operandos.

Solo se pueden usar determinados operadores como operadores de prefijo. Algunos operadores siempre son operadores de prefijo, otros pueden ser de infijo o de prefijo, y el resto son siempre operadores de infijo. Los operadores que comienzan por `!`, excepto `!=`, el operador `~` y las secuencias repetidas de `~`, son siempre operadores de prefijo. Los operadores `+`, `-`, `++`, `--`, `&`, `&&`, `%` y `%%` pueden ser operadores de prefijo u operadores de infijo. La versión prefija de estos operadores se distingue de su versión infija agregando `~` al principio de un operador de prefijo cuando se define. `~` no se usa al utilizar el operador, solo al definirlo.

## Ejemplo

En el código siguiente se muestra el uso de la sobrecarga de operadores para implementar un tipo de fracción. Una fracción se representa mediante un numerador y un denominador. La función `hcf` se usa para determinar el factor común, que se utiliza para reducir fracciones.

```
// Determine the highest common factor between
// two positive integers, a helper for reducing
// fractions.
let rec hcf a b =
    if a = 0u then b
    elif a < b then hcf a (b - a)
    else hcf (a - b) b

// type Fraction: represents a positive fraction
// (positive rational number).
type Fraction =
{
    // n: Numerator of fraction.
    n : uint32
    // d: Denominator of fraction.
    d : uint32
}

// Produce a string representation. If the
// denominator is "1", do not display it.
override this.ToString() =
    if (this.d = 1u)
    then this.n.ToString()
    else this.n.ToString() + "/" + this.d.ToString()

// Add two fractions.
static member (+) (f1 : Fraction, f2 : Fraction) =
    let nTemp = f1.n * f2.d + f2.n * f1.d
    let dTemp = f1.d * f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Adds a fraction and a positive integer.
static member (+) (f1: Fraction, i : uint32) =
    let nTemp = f1.n + i * f1.d
    let dTemp = f1.d
    let hcfTemp = hcf nTemp dTemp
```

```

    let nTemp = n1 * nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Adds a positive integer and a fraction.
static member (+) (i : uint32, f2: Fraction) =
    let nTemp = f2.n + i * f2.d
    let dTemp = f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Subtract one fraction from another.
static member (-) (f1 : Fraction, f2 : Fraction) =
    if (f2.n * f1.d > f1.n * f2.d)
        then failwith "This operation results in a negative number, which is not supported."
    let nTemp = f1.n * f2.d - f2.n * f1.d
    let dTemp = f1.d * f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Multiply two fractions.
static member (*) (f1 : Fraction, f2 : Fraction) =
    let nTemp = f1.n * f2.n
    let dTemp = f1.d * f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Divide two fractions.
static member (/) (f1 : Fraction, f2 : Fraction) =
    let nTemp = f1.n * f2.d
    let dTemp = f2.n * f1.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// A full set of operators can be quite lengthy. For example,
// consider operators that support other integral data types,
// with fractions, on the left side and the right side for each.
// Also consider implementing unary operators.

let fraction1 = { n = 3u; d = 4u }
let fraction2 = { n = 1u; d = 2u }
let result1 = fraction1 + fraction2
let result2 = fraction1 - fraction2
let result3 = fraction1 * fraction2
let result4 = fraction1 / fraction2
let result5 = fraction1 + 1u
printfn "%s + %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result1.ToString())
printfn "%s - %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result2.ToString())
printfn "%s * %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result3.ToString())
printfn "%s / %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result4.ToString())
printfn "%s + 1 = %s" (fraction1.ToString()) (result5.ToString())

```

Salida:

```

3/4 + 1/2 = 5/4
3/4 - 1/2 = 1/4
3/4 * 1/2 = 3/8
3/4 / 1/2 = 3/2
3/4 + 1 = 7/4

```

## Operadores en el nivel global

También se pueden definir operadores en el nivel global. En el código siguiente se define `+` un operador.

```
let inline (+?) (x: int) (y: int) = x + 2*y
printf "%d" (10 +? 1)
```

La salida del código anterior es `12`.

De esta forma, es posible redefinir los operadores aritméticos normales porque las reglas de ámbito para F# dictan que los operadores recién definidos tienen prioridad respecto de los operadores integrados.

La palabra clave `inline` se suele utilizar con los operadores globales, que a menudo son pequeñas funciones que se integran mejor en el código de llamada. Hacer que las funciones de operador sean inline permite que se puedan usar con los parámetros de tipo que se resuelven estáticamente a fin de generar código genérico resuelto estáticamente. Para obtener más información, vea [funciones insertadas](#) y [parámetros de tipo resueltos estáticamente](#).

## Vea también

- [Miembros](#)

# Tipos flexibles

23/10/2019 • 5 minutes to read • [Edit Online](#)

Una *anotación de tipo flexible* indica que un parámetro, una variable o un valor tiene un tipo que es compatible con un tipo especificado, donde la compatibilidad se determina por posición en una jerarquía orientada a objetos de clases o interfaces. Los tipos flexibles son especialmente útiles cuando la conversión automática en tipos situados más arriba en la jerarquía de tipos no se produce, pero aún desea habilitar la funcionalidad para trabajar con cualquier tipo de la jerarquía o con cualquier tipo que implemente una interfaz.

## Sintaxis

```
#type
```

## Comentarios

En la sintaxis anterior, *Type* representa un tipo base o una interfaz.

Un tipo flexible es equivalente a un tipo genérico que tiene una restricción que limita los tipos permitidos a tipos que son compatibles con la base o el tipo de interfaz. Es decir, las dos líneas de código siguientes son equivalentes.

```
#SomeType

'T when 'T :> SomeType
```

Los tipos flexibles son útiles en varios tipos de situaciones. Por ejemplo, si tiene una función de orden superior (una función que toma una función como argumento), a menudo resulta útil que la función devuelva un tipo flexible. En el ejemplo siguiente, el uso de un tipo flexible con un argumento de secuencia `iterate2` en permite que la función de orden superior funcione con funciones que generan secuencias, matrices, listas y cualquier otro tipo `Enumerable`.

Tenga en cuenta las dos funciones siguientes, una de las cuales devuelve una secuencia, la otra devuelve un tipo flexible.

```
let iterate1 (f : unit -> seq<int>) =
    for e in f() do printfn "%d" e
let iterate2 (f : unit -> #seq<int>) =
    for e in f() do printfn "%d" e

// Passing a function that takes a list requires a cast.
iterate1 (fun () -> [1] :> seq<int>)

// Passing a function that takes a list to the version that specifies a
// flexible type as the return value is OK as is.
iterate2 (fun () -> [1])
```

Como otro ejemplo, considere la función de biblioteca [Seq.concat](#) :

```
val concat: sequences:seq<#seq<'T>> -> seq<'T>
```

Puede pasar cualquiera de las secuencias enumerables siguientes a esta función:



- Una lista de listas
- Una lista de matrices
- Una matriz de listas
- Una matriz de secuencias
- Cualquier otra combinación de secuencias enumerables

En el código siguiente `Seq.concat` se usa para mostrar los escenarios que se pueden admitir mediante el uso de tipos flexibles.

```
let list1 = [1;2;3]
let list2 = [4;5;6]
let list3 = [7;8;9]

let concat1 = Seq.concat [ list1; list2; list3 ]
printfn "%A" concat1

let array1 = [|1;2;3|]
let array2 = [|4;5;6|]
let array3 = [|7;8;9|]

let concat2 = Seq.concat [ array1; array2; array3 ]
printfn "%A" concat2

let concat3 = Seq.concat [| list1; list2; list3 |]
printfn "%A" concat3

let concat4 = Seq.concat [| array1; array2; array3 |]
printfn "%A" concat4

let seq1 = { 1 .. 3 }
let seq2 = { 4 .. 6 }
let seq3 = { 7 .. 9 }

let concat5 = Seq.concat [| seq1; seq2; seq3 |]

printfn "%A" concat5
```

La salida es la siguiente.

```
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
```

En F#, como en otros lenguajes orientados a objetos, hay contextos en los que los tipos derivados o tipos que implementan interfaces se convierten automáticamente en un tipo base o de interfaz. Estas conversiones automáticas se producen en argumentos directos, pero no cuando el tipo está en una posición subordinada, como parte de un tipo más complejo, como un tipo de valor devuelto de un tipo de función, o como un argumento de tipo. Por lo tanto, la notación de tipos flexibles es principalmente útil cuando el tipo al que se aplica es parte de un tipo más complejo.

## Vea también

- [Referencia del lenguaje F#](#)
- [Genéricos](#)

# Delegados

23/10/2019 • 5 minutes to read • [Edit Online](#)

Un delegado representa una llamada de función como un objeto. En F#, normalmente debería usar valores de función para representar funciones como valores de primera clase; sin embargo, los delegados se usan en el .NET Framework y, por tanto, son necesarios cuando se interoperan con las API que los esperan. También se pueden usar al crear bibliotecas diseñadas para su uso desde otros lenguajes .NET Framework.

## Sintaxis

```
type delegate-typename = delegate of type1 -> type2
```

## Comentarios

En la sintaxis anterior, `type1` representa el tipo de argumento o los `type2` tipos y representa el tipo de valor devuelto. Los tipos de argumento que se representan `type1` mediante se convierten automáticamente en currificados. Esto sugiere que, para este tipo, se usa un formato de tupla si los argumentos de la función de destino son currificados y una tupla entre paréntesis para los argumentos que ya están en el formato de tupla. La currificación automática quita un conjunto de paréntesis y mantiene un argumento de tupla que coincide con el método de destino. Consulte el ejemplo de código para la sintaxis que se debe usar en cada caso.

Los delegados se F# pueden adjuntar a valores de función y métodos estáticos o de instancia. F# los valores de función se pueden pasar directamente como argumentos a los constructores de delegado. Para un método estático, se construye el delegado utilizando el nombre de la clase y el método. Para un método de instancia, debe proporcionar la instancia de objeto y el método en un argumento. En ambos casos, se utiliza el operador de `.` acceso a miembros `()`.

El `Invoke` método en el tipo de delegado llama a la función encapsulada. Además, los delegados se pueden pasar como valores de función haciendo referencia al nombre del método de invocación sin los paréntesis.

En el código siguiente se muestra la sintaxis para crear delegados que representan varios métodos de una clase. Dependiendo de si el método es un método estático o un método de instancia, y si tiene argumentos en el formato de tupla o en el formato currificados, la sintaxis para declarar y asignar el delegado es un poco diferente.

```

type Test1() =
    static member add(a : int, b : int) =
        a + b
    static member add2 (a : int) (b : int) =
        a + b

    member x.Add(a : int, b : int) =
        a + b
    member x.Add2 (a : int) (b : int) =
        a + b

// Delegate1 works with tuple arguments.
type Delegate1 = delegate of (int * int) -> int
// Delegate2 works with curried arguments.
type Delegate2 = delegate of int * int -> int

let InvokeDelegate1 (dlg : Delegate1) (a : int) (b: int) =
    dlg.Invoke(a, b)
let InvokeDelegate2 (dlg : Delegate2) (a : int) (b: int) =
    dlg.Invoke(a, b)

// For static methods, use the class name, the dot operator, and the
// name of the static method.
let del1 : Delegate1 = new Delegate1( Test1.add )
let del2 : Delegate2 = new Delegate2( Test1.add2 )

let testObject = Test1()

// For instance methods, use the instance value name, the dot operator, and the instance method name.
let del3 : Delegate1 = new Delegate1( testObject.Add )
let del4 : Delegate2 = new Delegate2( testObject.Add2 )

for (a, b) in [ (100, 200); (10, 20) ] do
    printfn "%d + %d = %d" a b (InvokeDelegate1 del1 a b)
    printfn "%d + %d = %d" a b (InvokeDelegate2 del2 a b)
    printfn "%d + %d = %d" a b (InvokeDelegate1 del3 a b)
    printfn "%d + %d = %d" a b (InvokeDelegate2 del4 a b)

```

En el código siguiente se muestran algunas de las distintas formas en que puede trabajar con los delegados.

```

type Delegate1 = delegate of int * char -> string

let replicate n c = String.replicate n (string c)

// An F# function value constructed from an unapplied let-bound function
let function1 = replicate

// A delegate object constructed from an F# function value
let delObject = new Delegate1(function1)

// An F# function value constructed from an unapplied .NET member
let functionValue = delObject.Invoke

List.map (fun c -> functionValue(5,c)) ['a'; 'b'; 'c']
|> List.iter (printfn "%s")

// Or if you want to get back the same curried signature
let replicate' n c = delObject.Invoke(n,c)

// You can pass a lambda expression as an argument to a function expecting a compatible delegate type
// System.Array.ConvertAll takes an array and a converter delegate that transforms an element from
// one type to another according to a specified function.
let stringArray = System.Array.ConvertAll(['a';'b'], fun c -> replicate' 3 c)
printfn "%A" stringArray

```

La salida del ejemplo de código anterior es la siguiente.

```

aaaaa
bbbbbb
ccccc
[|"aaa"; "bbb"|]

```

## Vea también

- [Referencia del lenguaje F#](#)
- [Parámetros y argumentos](#)
- [Eventos](#)

# Expresiones de objeto

23/10/2019 • 2 minutes to read • [Edit Online](#)

Un *objeto expresión* es una expresión que crea una nueva instancia de un tipo de objeto anónimo creado dinámicamente que se basa en un tipo base existente, una interfaz o un conjunto de interfaces.

## Sintaxis

```
// When typename is a class:
{ new typename [type-params]arguments with
  member-definitions
  [ additional-interface-definitions ]
}
// When typename is not a class:
{ new typename [generic-type-args] with
  member-definitions
  [ additional-interface-definitions ]
}
```

## Comentarios

En la sintaxis anterior, el *typename* representa un tipo de clase existente o un tipo de interfaz. *parámetros de tipo* se describen los parámetros de tipo genérico opcionales. El *argumentos* se usan únicamente para los tipos de clase, que requieren los parámetros del constructor. El *definiciones de miembros* son reemplazos de métodos de clase base o implementaciones de los métodos abstractos de una clase base o una interfaz.

El ejemplo siguiente muestra los distintos tipos de expresiones de objeto.

```

// This object expression specifies a System.Object but overrides the
// ToString method.
let obj1 = { new System.Object() with member x.ToString() = "F#" }
printfn "%A" obj1

// This object expression implements the IFormattable interface.
let delimiter(delim1: string, delim2: string, value: string) =
    { new System.IFormattable with
        member x.ToString(format: string, provider: System.IFormatProvider) =
            if format = "D" then
                delim1 + value + delim2
            else
                value }

let obj2 = delimiter("{","}", "Bananas!");

printfn "%A" (System.String.Format("{0:D}", obj2))

// Define two interfaces
type IFirst =
    abstract F : unit -> unit
    abstract G : unit -> unit

type ISecond =
    inherit IFirst
    abstract H : unit -> unit
    abstract J : unit -> unit

// This object expression implements both interfaces.
let implementer() =
    { new ISecond with
        member this.H() = ()
        member this.J() = ()
        interface IFirst with
            member this.F() = ()
            member this.G() = () }

```

## Uso de expresiones de objeto

Usar expresiones de objeto cuando desea evitar el código adicional y la sobrecarga que se necesita para crear un nuevo tipo con nombre. Si se usan expresiones de objeto para minimizar el número de tipos creados en un programa, puede reducir el número de líneas de código y evitar la proliferación innecesaria de tipos. En lugar de crear muchos tipos simplemente para controlar situaciones específicas, puede usar una expresión de objeto que se personaliza un tipo existente o proporcione una implementación apropiada de una interfaz para el caso en cuestión.

## Vea también

- [Referencia del lenguaje F#](#)

# Conversiones (F#)

21/02/2020 • 11 minutes to read • [Edit Online](#)

En este tema se describe la compatibilidad con las conversiones de tipos en F#.

## Tipos aritméticos

F#proporciona operadores de conversión para las conversiones aritméticas entre varios tipos primitivos, como entre tipos de punto flotante y enteros. Los operadores de conversión integral y char tienen formularios comprobados y no comprobados; los operadores de punto flotante y el operador de conversión `enum` no. Los formularios no comprobados se definen en `Microsoft.FSharp.Core.Operators` y los formularios marcados se definen en `Microsoft.FSharp.Core.Operators.Checked`. Los formularios comprobados comprueban si hay desbordamiento y generan una excepción en tiempo de ejecución si el valor resultante supera los límites del tipo de destino.

Cada uno de estos operadores tiene el mismo nombre que el nombre del tipo de destino. Por ejemplo, en el código siguiente, en el que los tipos se anotan explícitamente, `byte` aparece con dos significados diferentes. La primera aparición es el tipo y el segundo es el operador de conversión.

```
let x : int = 5

let b : byte = byte x
```

En la tabla siguiente se muestran los operadores F#de conversión definidos en.

OPERATOR	DESCRIPCIÓN
<code>byte</code>	Convertir en byte, un tipo sin signo de 8 bits.
<code>sbyte</code>	Convertir en bytes con signo.
<code>int16</code>	Convertir en un entero de 16 bits con signo.
<code>uint16</code>	Convertir en un entero de 16 bits sin signo.
<code>int32, int</code>	Convertir en un entero de 32 bits con signo.
<code>uint32</code>	Convertir en un entero de 32 bits sin signo.
<code>int64</code>	Convertir en un entero de 64 bits con signo.
<code>uint64</code>	Convertir en un entero de 64 bits sin signo.
<code>nativeint</code>	Convertir en un entero nativo.
<code>unativeint</code>	Convertir en un entero nativo sin signo.

OPERATOR	DESCRIPCIÓN
<code>float, double</code>	Convertir en un número de punto flotante de IEEE de doble precisión de 64 bits.
<code>float32, single</code>	Convertir en un número de punto flotante de IEEE de precisión sencilla de 32 bits.
<code>decimal</code>	Convertir en <code>System.Decimal</code> .
<code>char</code>	Convertir en <code>System.Char</code> , un carácter Unicode.
<code>enum</code>	Convertir en un tipo enumerado.

Además de los tipos primitivos integrados, puede usar estos operadores con tipos que implementan `op_Explicit` o `op_Implicit` métodos con las firmas adecuadas. Por ejemplo, el operador de conversión `int` funciona con cualquier tipo que proporcione un método estático `op_Explicit` que toma el tipo como parámetro y devuelve `int`. Como excepción especial a la regla general de que los métodos no se pueden sobrecargar por tipo de valor devuelto, puede hacerlo para `op_Explicit` y `op_Implicit`.

## Tipos enumerados

El operador `enum` es un operador genérico que toma un parámetro de tipo que representa el tipo del `enum` al que se va a convertir. Cuando se convierte a un tipo enumerado, la inferencia de tipos intenta determinar el tipo de `enum` al que se desea convertir. En el ejemplo siguiente, la variable `col1` no se anota explícitamente, pero su tipo se deduce de la prueba de igualdad posterior. Por consiguiente, el compilador puede deducir que se está convirtiendo en una enumeración `Color`. Como alternativa, puede proporcionar una anotación de tipo, como con `col2` en el ejemplo siguiente.

```
type Color =
    | Red = 1
    | Green = 2
    | Blue = 3

// The target type of the conversion cannot be determined by type inference, so the type parameter must be explicit.
let col1 = enum<Color> 1

// The target type is supplied by a type annotation.
let col2 : Color = enum 2
```

También puede especificar explícitamente el tipo de enumeración de destino como un parámetro de tipo, como en el código siguiente:

```
let col3 = enum<Color> 3
```

Tenga en cuenta que las conversiones de enumeración funcionan solo si el tipo subyacente de la enumeración es compatible con el tipo que se va a convertir. En el código siguiente, la conversión no se compila debido a la falta de coincidencia entre `int32` y `uint32`.

```
// Error: types are incompatible
let col4 : Color = enum 2u
```



Para obtener más información, vea [enumeraciones](#).

## Convertir tipos de objeto

La conversión entre tipos en una jerarquía de objetos es fundamental para la programación orientada a objetos. Hay dos tipos básicos de conversiones: conversión hacia arriba (conversión) y conversión hacia abajo (downcasting). La conversión de una jerarquía significa la conversión de una referencia de objeto derivada a una referencia de objeto base. Se garantiza que este tipo de conversión funciona siempre y cuando la clase base se encuentra en la jerarquía de herencia de la clase derivada. La conversión de una jerarquía de una referencia de objeto base a una referencia de objeto derivada solo se realiza correctamente si el objeto es realmente una instancia del tipo de destino (derivado) correcto o un tipo derivado del tipo de destino.

F#proporciona operadores para estos tipos de conversiones. El operador `:` convierte hacia arriba la jerarquía y el operador `?:>` convierte en la jerarquía.

### Convertir

En muchos lenguajes orientados a objetos, la conversión cruzada es implícita; en F#, las reglas son ligeramente diferentes. La conversión se aplica automáticamente cuando se pasan argumentos a métodos en un tipo de objeto. Sin embargo, en el caso de las funciones enlazadas a un módulo, la conversión cruzada no es automática, a menos que el tipo de parámetro se declare como un tipo flexible. Para obtener más información, vea [tipos flexibles](#).

El operador `:` realiza una conversión estática, lo que significa que el éxito de la conversión se determina en tiempo de compilación. Si una conversión que usa `:` se compila correctamente, es una conversión válida y no tiene posibilidad de errores en tiempo de ejecución.

También puede usar el operador `upcast` para realizar esta conversión. La expresión siguiente especifica una conversión hacia arriba en la jerarquía:

```
upcast expression
```

Cuando se usa el operador de conversión, el compilador intenta deducir el tipo al que se va a convertir desde el contexto. Si el compilador no puede determinar el tipo de destino, el compilador informa de un error. Es posible que se requiera una anotación de tipo.

### Downcasting

El operador `?:>` realiza una conversión dinámica, lo que significa que el éxito de la conversión se determina en tiempo de ejecución. Una conversión que usa el operador `?:>` no se comprueba en tiempo de compilación; pero en tiempo de ejecución, se realiza un intento de conversión al tipo especificado. Si el objeto es compatible con el tipo de destino, la conversión se realiza correctamente. Si el objeto no es compatible con el tipo de destino, el motor en tiempo de ejecución genera un `InvalidCastException`.

También puede usar el operador `downcast` para realizar una conversión de tipos dinámica. La expresión siguiente especifica una conversión hacia abajo en la jerarquía hasta un tipo que se deduce del contexto del programa:

```
downcast expression
```

Como en el caso del operador `upcast`, si el compilador no puede inferir un tipo de destino específico del contexto, notifica un error. Es posible que se requiera una anotación de tipo.

En el código siguiente se muestra el uso de los operadores `:` y `?:>`. El código muestra que el operador de `?:>` se utiliza mejor cuando sabe que la conversión se realizará correctamente, ya que produce `InvalidCastException` si se produce un error en la conversión. Si no sabe que una conversión se realizará correctamente, una prueba de

tipo que use una expresión `match` es mejor porque evita la sobrecarga de generar una excepción.

```
type Base1() =
    abstract member F : unit -> unit
    default u.F() =
        printfn "F Base1"

type Derived1() =
    inherit Base1()
    override u.F() =
        printfn "F Derived1"

let d1 : Derived1 = Derived1()

// Upcast to Base1.
let base1 = d1 :> Base1

// This might throw an exception, unless
// you are sure that base1 is really a Derived1 object, as
// is the case here.
let derived1 = base1 :?> Derived1

// If you cannot be sure that b1 is a Derived1 object,
// use a type test, as follows:
let downcastBase1 (b1 : Base1) =
    match b1 with
    | :? Derived1 as derived1 -> derived1.F()
    | _ -> ()

downcastBase1 base1
```

Dado que los operadores genéricos `downcast` y `upcast` dependen de la inferencia de tipos para determinar el argumento y el tipo de valor devuelto, en el código anterior, puede reemplazar

```
let base1 = d1 :> Base1
```

con

```
let base1: Base1 = upcast d1
```

Tenga en cuenta que se requiere una anotación de tipo, ya que `upcast` por sí misma no pudo determinar la clase base.

## Consulte también

- [Referencia del lenguaje F#](#)

# Control de acceso

04/11/2019 • 5 minutes to read • [Edit Online](#)

El *control de acceso* hace referencia a la declaración de los clientes que pueden usar determinados elementos de programa, como tipos, métodos y funciones.

## Aspectos básicos de Access Control

En F#, los especificadores de control de acceso `public`, `internal` y `private` pueden aplicarse a módulos, tipos, métodos, definiciones de valores, funciones, propiedades y campos explícitos.

- `public` indica que todos los llamadores pueden tener acceso a la entidad.
- `internal` indica que solo se puede tener acceso a la entidad desde el mismo ensamblado.
- `private` indica que solo se puede tener acceso a la entidad desde el tipo o módulo envolvente.

### NOTE

El especificador de acceso `protected` no se utiliza F#, aunque es aceptable si se usan tipos creados en lenguajes que admiten el acceso a `protected`. Por consiguiente, si invalida un método protegido, el método permanece accesible solo dentro de la clase y sus descendientes.

En general, el especificador se coloca delante del nombre de la entidad, excepto cuando se usa un especificador `mutable` o `inline`, que aparece después del especificador de control de acceso.

Si no se usa ningún especificador de acceso, el valor predeterminado es `public`, excepto para los enlaces de `let` en un tipo, que siempre se `private` al tipo.

Las firmas de F# proporcionan otro mecanismo para controlar el acceso F# a los elementos del programa. No se necesitan firmas para el control de acceso. Para más información, vea [Signatures](#) (Firmas).

## Reglas para Access Control

El control de acceso está sujeto a las siguientes reglas:

- Las declaraciones de herencia (es decir, el uso de `inherit` para especificar una clase base para una clase), las declaraciones de interfaz (es decir, especificando que una clase implementa una interfaz) y los miembros abstractos siempre tienen la misma accesibilidad que el tipo envolvente. Por lo tanto, no se puede usar un especificador de control de acceso en estas construcciones.
- La accesibilidad de los casos individuales en una Unión discriminada viene determinada por la accesibilidad de la propia Unión discriminada. Es decir, un caso de Unión determinado no es menos accesible que la propia Unión.
- La accesibilidad del propio registro determina la accesibilidad de los campos individuales de un tipo de registro. Es decir, una etiqueta de registro determinada no es menos accesible que el propio registro.

## Ejemplo

En el código siguiente se muestra el uso de especificadores de control de acceso. Hay dos archivos en el proyecto, `Module1.fs` y `Module2.fs`. Cada archivo es implícitamente un módulo. Por lo tanto, hay dos

módulos, `Module1` y `Module2`. En `Module1` se definen un tipo privado y un tipo interno. No se puede tener acceso al tipo privado desde `Module2`, pero el tipo interno puede.

```
// Module1.fs

module Module1

// This type is not usable outside of this file
type private MyPrivateType() =
    // x is private since this is an internal let binding
    let x = 5
    // X is private and does not appear in the QuickInfo window
    // when viewing this type in the Visual Studio editor
    member private this.X() = 10
    member this.Z() = x * 100

type internal MyInternalType() =
    let x = 5
    member private this.X() = 10
    member this.Z() = x * 100

// Top-level let bindings are public by default,
// so "private" and "internal" are needed here since a
// value cannot be more accessible than its type.
let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

// let bindings at the top level are public by default,
// so result1 and result2 are public.
let result1 = myPrivateObj.Z
let result2 = myInternalObj.Z
```

En el código siguiente se prueba la accesibilidad de los tipos creados en `Module1.fs`.

```
// Module2.fs
module Module2

open Module1

// The following line is an error because private means
// that it cannot be accessed from another file or module
// let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

let result = myInternalObj.Z
```

## Vea también

- [Referencia del lenguaje F#](#)
- [Firmas](#)

# Expresiones condicionales: `if...then...else`

23/10/2019 • 2 minutes to read • [Edit Online](#)

La `if...then...else` expresión ejecuta distintas bifurcaciones de código y también se evalúa como un valor diferente en función de la expresión booleana dada.

## Sintaxis

```
if boolean-expression then expression1 [ else expression2 ]
```

## Comentarios

En la sintaxis anterior, *expression1* se ejecuta cuando la expresión booleana se evalúa `true` como; de lo contrario, *expression2* se ejecuta.

A diferencia de otros lenguajes, `if...then...else` la construcción es una expresión, no una instrucción. Esto significa que genera un valor, que es el valor de la última expresión de la bifurcación que se ejecuta. Los tipos de los valores generados en cada bifurcación deben coincidir. Si no hay ninguna rama `else` explícita, su tipo es `unit`. Por lo tanto, si el tipo `then` de la bifurcación es cualquier `unit` tipo que no sea, `else` debe haber una rama con el mismo tipo de valor devuelto. Al encadenar `if...then...else` expresiones juntas, puede usar la palabra `elif` clave en lugar `else if` de; son equivalentes.

## Ejemplo

En el ejemplo siguiente se muestra cómo usar la `if...then...else` expresión.

```
let test x y =
    if x = y then "equals"
    elif x < y then "is less than"
    else "is greater than"

printfn "%d %s %d." 10 (test 10 20) 20

printfn "What is your name? "
let nameString = System.Console.ReadLine()

printfn "What is your age? "
let ageString = System.Console.ReadLine()
let age = System.Int32.Parse(ageString)

if age < 10
then printfn "You are only %d years old and already learning F#? Wow!" age
```

```
10 is less than 20
What is your name? John
How old are you? 9
You are only 9 years old and already learning F#? Wow!
```

Vea también

- [Referencia del lenguaje F#](#)

# Expresiones de coincidencia

23/10/2019 • 5 minutes to read • [Edit Online](#)

La `match` expresión proporciona control de bifurcación que se basa en la comparación de una expresión con un conjunto de patrones.

## Sintaxis

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...

// Pattern matching function.
function
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

## Comentarios

Las expresiones de coincidencia de patrones permiten la bifurcación compleja basada en la comparación de una expresión de prueba con un conjunto de patrones. En la `match` expresión, *Test-Expression* se compara con cada patrón a su vez y, cuando se encuentra una coincidencia, se evalúa la *expresión de resultado* correspondiente y el valor resultante se devuelve como el valor de la expresión de coincidencia.

La función de coincidencia de patrones mostrada en la sintaxis anterior es una expresión lambda en la que la coincidencia de patrones se realiza inmediatamente en el argumento. La función de coincidencia de patrones que se muestra en la sintaxis anterior es equivalente a la siguiente.

```
fun arg ->
  match arg with
  | pattern1 [ when condition ] -> result-expression1
  | pattern2 [ when condition ] -> result-expression2
  | ...
```

Para obtener más información sobre las expresiones lambda, [vea expresiones lambda](#): `fun` **Palabra clave**.

Todo el conjunto de patrones debe cubrir todas las posibles coincidencias de la variable de entrada. Con frecuencia, se usa el patrón de `_` carácter comodín () como último patrón para buscar coincidencias con los valores de entrada no coincidentes previamente.

En el código siguiente se muestran algunas de las formas en que `match` se usa la expresión. Para obtener una referencia y ejemplos de todos los patrones posibles que se pueden usar, [vea coincidencia de patrones](#).

```

let list1 = [ 1; 5; 100; 450; 788 ]

// Pattern matching by using the cons pattern and a list
// pattern that tests for an empty list.
let rec printList listx =
  match listx with
  | head :: tail -> printf "%d " head; printList tail
  | [] -> printfn ""

printList list1

// Pattern matching with multiple alternatives on the same line.
let filter123 x =
  match x with
  | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
  | a -> printfn "%d" a

// The same function written with the pattern matching
// function syntax.
let filterNumbers =
  function | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
           | a -> printfn "%d" a

```

## Protecciones en patrones

Puede usar una `when` cláusula para especificar una condición adicional que la variable debe cumplir para que coincida con un patrón. Este tipo de cláusula se conoce como *protección*. La expresión que sigue `when` a la palabra clave no se evalúa a menos que se haga una coincidencia con el patrón asociado a dicha protección.

En el ejemplo siguiente se muestra el uso de una protección para especificar un intervalo numérico para un patrón de variable. Tenga en cuenta que se combinan varias condiciones mediante el uso de operadores booleanos.

```

let rangeTest testValue mid size =
  match testValue with
  | var1 when var1 >= mid - size/2 && var1 <= mid + size/2 -> printfn "The test value is in range."
  | _ -> printfn "The test value is out of range."

rangeTest 10 20 5
rangeTest 10 20 10
rangeTest 10 20 40

```

Tenga en cuenta que, dado que los valores distintos de los literales no se pueden usar en `when` el modelo, debe utilizar una cláusula si tiene que comparar alguna parte de la entrada con un valor. Esto se muestra en el código siguiente:

```

// This example uses patterns that have when guards.
let detectValue point target =
  match point with
  | (a, b) when a = target && b = target -> printfn "Both values match target %d." target
  | (a, b) when a = target -> printfn "First value matched target in (%d, %d)" target b
  | (a, b) when b = target -> printfn "Second value matched target in (%d, %d)" a target
  | _ -> printfn "Neither value matches target."

detectValue (0, 0) 0
detectValue (1, 0) 0
detectValue (0, 10) 0
detectValue (10, 15) 0

```

Tenga en cuenta que cuando un modelo de Unión está incluido en una protección, la protección se aplica a



todos los patrones, no solo al último. Por ejemplo, según el código siguiente, la protección `when a > 12` se aplica tanto `A a` a `B a` como a:

```
type Union =  
    | A of int  
    | B of int  
  
let foo() =  
    let test = A 42  
    match test with  
    | A a  
    | B a when a > 41 -> a // the guard applies to both patterns  
    | _ -> 1  
  
foo() // returns 42
```

## Vea también

- [Referencia del lenguaje F#](#)
- [Patrones activos](#)
- [Coincidencia de patrones](#)

# Coincidencia de modelos

29/10/2019 • 22 minutes to read • [Edit Online](#)

Los patrones son reglas para transformar los datos de entrada. Se usan en todo el F# lenguaje para comparar los datos con una estructura lógica o estructuras, descomponer los datos en partes constituyentes o extraer información de los datos de varias maneras.

## Comentarios

Los patrones se usan en muchas construcciones de lenguaje, como la expresión `match`. Se utilizan cuando se procesan argumentos para funciones en enlaces de `let`, expresiones lambda y en los controladores de excepción asociados a la expresión `try...with`. Para obtener más información, [Vea expresiones de coincidencia](#), [enlaces Let](#), [expresiones Lambda: la palabra clave fun](#) y [excepciones: la expresión try...with](#).

Por ejemplo, en la expresión `match`, el *patrón* es lo que sigue al símbolo de canalización.

```
match expression with
| pattern [ when condition ] -> result-expression
...
```

Cada patrón actúa como una regla para transformar la entrada de alguna manera. En la expresión `match`, cada patrón se examina a su vez para ver si los datos de entrada son compatibles con el patrón. Si se encuentra una coincidencia, se ejecuta la expresión de resultado. Si no se encuentra ninguna coincidencia, se prueba la siguiente regla de patrón. La parte de *condición* opcional `when` se explica en [expresiones de coincidencia](#).

Los patrones admitidos se muestran en la tabla siguiente. En tiempo de ejecución, la entrada se prueba con cada uno de los patrones siguientes en el orden indicado en la tabla y los patrones se aplican de forma recursiva, de la primera a la última, como aparecen en el código y de izquierda a derecha para los patrones de cada línea.

NAME	DESCRIPCIÓN	EJEMPLO
Patrón de constante	Cualquier literal numérico, de carácter o de cadena, una constante de enumeración o un identificador literal definido	<code>1.0</code> , <code>"test"</code> , <code>30</code> , <code>Color.Red</code>
Patrón de identificador	Un valor de caso de una Unión discriminada, una etiqueta de excepción o un caso de modelo activo	<code>Some(x)</code> <code>Failure(msg)</code>
Patrón de variable	<i>identifier</i>	<code>a</code>
patrón de <code>as</code>	<i>patrón como identificador</i>	<code>(a, b) as tuple1</code>
Patrón OR	<i>pattern1   pattern2</i>	<code>([h]   [h; _])</code>
Y patrón	<i>pattern1 &amp; pattern2</i>	<code>(a, b) &amp; (_, "test")</code>
Patrón de cons	<i>Identifier :: List-Identifier</i>	<code>h :: t</code>

NAME	DESCRIPCIÓN	EJEMPLO
Patrón de lista	[ <i>pattern_1</i> ;...; <i>pattern_n</i> ]	[ a; b; c ]
Patrón de matriz	[  <i>pattern_1</i> ;... <i>pattern_n</i>  ]	[  a; b; c  ]
Patrón entre paréntesis	( <i>patrón</i> )	( a )
Patrón de tupla	( <i>pattern_1</i> ,..., <i>pattern_n</i> )	( a, b )
Patrón de registro	{ <i>identificador1</i> = <i>pattern_1</i> ;...; <i>identifier_n</i> = <i>pattern_n</i> }	{ Name = name; }
Patrón de carácter comodín		_
Patrón junto con anotación de tipo	<i>patrón</i> : <i>tipo</i>	a : int
Modelo de prueba de tipo	:? <i>tipo</i> [como <i>identificador</i> ]	:? System.DateTime as dt
Patrón null	nulo	null

## Patrones constantes

Los patrones constantes son los literales numéricos, de carácter y de cadena, las constantes de enumeración (con el nombre de tipo de enumeración incluido). Una expresión `match` que solo tiene patrones constantes puede compararse con una instrucción Case en otros lenguajes. La entrada se compara con el valor literal y el patrón coincide si los valores son iguales. El tipo del literal debe ser compatible con el tipo de la entrada.

En el ejemplo siguiente se muestra el uso de patrones literales y también se usa un patrón de variable y un patrón o.

```
[<Literal>]
let Three = 3

let filter123 x =
    match x with
    // The following line contains literal patterns combined with an OR pattern.
    | 1 | 2 | Three -> printfn "Found 1, 2, or 3!"
    // The following line contains a variable pattern.
    | var1 -> printfn "%d" var1

for x in 1..10 do filter123 x
```

Otro ejemplo de un patrón literal es un modelo basado en constantes de enumeración. Debe especificar el nombre del tipo de enumeración al usar constantes de enumeración.

```

type Color =
  | Red = 0
  | Green = 1
  | Blue = 2

let printColorName (color:Color) =
  match color with
  | Color.Red -> printfn "Red"
  | Color.Green -> printfn "Green"
  | Color.Blue -> printfn "Blue"
  | _ -> ()

printColorName Color.Red
printColorName Color.Green
printColorName Color.Blue

```

## Patrones de identificador

Si el patrón es una cadena de caracteres que forma un identificador válido, el formato del identificador determina cómo coincide el patrón. Si el identificador es más largo que un carácter único y comienza con un carácter en mayúsculas, el compilador intenta establecer una coincidencia con el patrón de identificador. El identificador de este patrón puede ser un valor marcado con el atributo literal, un caso de Unión discriminada, un identificador de excepción o un caso de modelo activo. Si no se encuentra ningún identificador coincidente, se produce un error en la coincidencia y la siguiente regla de patrón, el patrón de variable, se compara con la entrada.

Los patrones de Unión discriminada pueden ser casos con nombre sencillos o pueden tener un valor o una tupla que contiene varios valores. Si hay un valor, debe especificar un identificador para el valor. En el caso de una tupla, debe proporcionar un patrón de tupla con un identificador para cada elemento de la tupla o un identificador con un nombre de campo para uno o más campos de unión con nombre. Vea los ejemplos de código de esta sección para obtener ejemplos.

El `option` tipo es una Unión discriminada que tiene dos casos, `Some` y `None`. Un caso (`Some`) tiene un valor, pero el otro (`None`) es simplemente un caso con nombre. Por lo tanto, `Some` necesita tener una variable para el valor asociado al `Some` caso, pero `None` debe aparecer por sí sola. En el código siguiente, a la variable `var1` se le proporciona el valor que se obtiene al buscar coincidencias con el `Some` caso.

```

let printOption (data : int option) =
  match data with
  | Some var1 -> printfn "%d" var1
  | None -> ()

```

En el ejemplo siguiente, la Unión discriminada `PersonName` contiene una mezcla de cadenas y caracteres que representan las posibles formas de los nombres. Los casos de la Unión discriminada son `FirstOnly`, `LastOnly` y `FirstLast`.

```

type PersonName =
  | FirstOnly of string
  | LastOnly of string
  | FirstLast of string * string

let constructQuery personName =
  match personName with
  | FirstOnly(firstName) -> printf "May I call you %s?" firstName
  | LastOnly(lastName) -> printf "Are you Mr. or Ms. %s?" lastName
  | FirstLast(firstName, lastName) -> printf "Are you %s %s?" firstName lastName

```

En el caso de las uniones discriminadas que tienen campos con nombre, use el signo igual (=) para extraer el valor de un campo con nombre. Por ejemplo, considere una Unión discriminada con una declaración como la siguiente.

```
type Shape =  
  | Rectangle of height : float * width : float  
  | Circle of radius : float
```

Puede usar los campos con nombre en una expresión de coincidencia de patrones como se indica a continuación.

```
let matchShape shape =  
  match shape with  
  | Rectangle(height = h) -> printfn "Rectangle with length %f" h  
  | Circle(r) -> printfn "Circle with radius %f" r
```

El uso del campo con nombre es opcional, por lo que en el ejemplo anterior, tanto `Circle(r)` como `Circle(radius = r)` tienen el mismo efecto.

Cuando especifique varios campos, use el punto y coma (;) como separador.

```
match shape with  
| Rectangle(height = h; width = w) -> printfn "Rectangle with height %f and width %f" h w  
| _ -> ()
```

Los modelos activos permiten definir una coincidencia de patrones personalizada más compleja. Para obtener más información sobre los patrones activos, vea [patrones activos](#).

El caso en el que el identificador es una excepción se usa en la coincidencia de patrones en el contexto de los controladores de excepciones. Para obtener información sobre la coincidencia de patrones en el control de excepciones, vea [excepciones: la expresión](#) `try...with`.

## Patrones de variables

El patrón variable asigna el valor que coincide con un nombre de variable, que está disponible para su uso en la expresión de ejecución a la derecha del símbolo `->`. Un patrón variable solo coincide con cualquier entrada, pero los modelos variables suelen aparecer dentro de otros patrones, lo que permite que las estructuras más complejas, como las tuplas y las matrices, se descompongan en variables.

En el ejemplo siguiente se muestra un patrón de variable dentro de un patrón de tupla.

```
let function1 x =  
  match x with  
  | (var1, var2) when var1 > var2 -> printfn "%d is greater than %d" var1 var2  
  | (var1, var2) when var1 < var2 -> printfn "%d is less than %d" var1 var2  
  | (var1, var2) -> printfn "%d equals %d" var1 var2  
  
function1 (1,2)  
function1 (2, 1)  
function1 (0, 0)
```

## como patrón

El patrón `as` es un patrón que tiene una cláusula `as` anexada a él. La cláusula `as` enlaza el valor coincidente con un nombre que se puede usar en la expresión de ejecución de una expresión `match`, o bien, en el caso en

el que este patrón se usa en un enlace de `let`, el nombre se agrega como un enlace al ámbito local.

En el ejemplo siguiente se usa un patrón de `as`.

```
let (var1, var2) as tuple1 = (1, 2)
printfn "%d %d %A" var1 var2 tuple1
```

## Patrón OR

El patrón `or` se utiliza cuando los datos de entrada pueden coincidir con varios patrones y desea ejecutar el mismo código como resultado. Los tipos de ambos lados del patrón `or` deben ser compatibles.

En el siguiente ejemplo se muestra el patrón `or`.

```
let detectZeroOR point =
    match point with
    | (0, 0) | (0, _) | (_, 0) -> printfn "Zero found."
    | _ -> printfn "Both nonzero."
detectZeroOR (0, 0)
detectZeroOR (1, 0)
detectZeroOR (0, 10)
detectZeroOR (10, 15)
```

## Y patrón

El patrón `and` requiere que la entrada coincida con dos patrones. Los tipos de ambos lados del patrón `and` deben ser compatibles.

El ejemplo siguiente es como `detectZeroTuple` se muestra en la sección [modelo de tupla](#) más adelante en este tema, pero aquí tanto `var1` como `var2` se obtienen como valores mediante el patrón `and`.

```
let detectZeroAND point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (var1, var2) & (0, _) -> printfn "First value is 0 in (%d, %d)" var1 var2
    | (var1, var2) & (_, 0) -> printfn "Second value is 0 in (%d, %d)" var1 var2
    | _ -> printfn "Both nonzero."
detectZeroAND (0, 0)
detectZeroAND (1, 0)
detectZeroAND (0, 10)
detectZeroAND (10, 15)
```

## Patrón de cons

El patrón `cons` se usa para descomponer una lista en el primer elemento, el *encabezado* y una lista que contiene los elementos restantes, la *cola*.

```
let list1 = [ 1; 2; 3; 4 ]

// This example uses a cons pattern and a list pattern.
let rec printList l =
    match l with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1
```

## Patrón de lista

El patrón de lista permite descomponer listas en varios elementos. El propio patrón de lista solo puede coincidir con las listas de un número específico de elementos.

```
// This example uses a list pattern.
let listLength list =
    match list with
    | [] -> 0
    | [ _ ] -> 1
    | [ _; _ ] -> 2
    | [ _; _; _ ] -> 3
    | _ -> List.length list

printfn "%d" (listLength [ 1 ])
printfn "%d" (listLength [ 1; 1 ])
printfn "%d" (listLength [ 1; 1; 1; ])
printfn "%d" (listLength [ ] )
```

## Patrón de matriz

El patrón de matriz se asemeja al patrón de lista y se puede usar para descomponer las matrices de una longitud específica.

```
// This example uses array patterns.
let vectorLength vec =
    match vec with
    | [| var1 |] -> var1
    | [| var1; var2 |] -> sqrt (var1*var1 + var2*var2)
    | [| var1; var2; var3 |] -> sqrt (var1*var1 + var2*var2 + var3*var3)
    | _ -> failwith (sprintf "vectorLength called with an unsupported array size of %d." (vec.Length))

printfn "%f" (vectorLength [| 1. |])
printfn "%f" (vectorLength [| 1.; 1. |])
printfn "%f" (vectorLength [| 1.; 1.; 1.; |])
printfn "%f" (vectorLength [| |] )
```

## Patrón entre paréntesis

Los paréntesis se pueden agrupar en torno a patrones para lograr la asociatividad deseada. En el ejemplo siguiente, los paréntesis se usan para controlar la asociatividad entre un patrón AND y un patrón cons.

```
let countValues list value =
    let rec checkList list acc =
        match list with
        | (elem1 & head) :: tail when elem1 = value -> checkList tail (acc + 1)
        | head :: tail -> checkList tail acc
        | [] -> acc
    checkList list 0

let result = countValues [ for x in -10..10 -> x*x - 4 ] 0
printfn "%d" result
```

## Patrón de tupla

El patrón de tupla asocia la entrada en forma de tupla y permite descomponer la tupla en sus elementos constituyentes mediante el uso de variables de coincidencia de patrones para cada posición de la tupla.

En el ejemplo siguiente se muestra el patrón de tupla y también se usan patrones literales, patrones de variables y el patrón de caracteres comodín.

```
let detectZeroTuple point =
  match point with
  | (0, 0) -> printfn "Both values zero."
  | (0, var2) -> printfn "First value is 0 in (0, %d)" var2
  | (var1, 0) -> printfn "Second value is 0 in (%d, 0)" var1
  | _ -> printfn "Both nonzero."
detectZeroTuple (0, 0)
detectZeroTuple (1, 0)
detectZeroTuple (0, 10)
detectZeroTuple (10, 15)
```

## Patrón de registro

El patrón de registro se usa para descomponer registros para extraer los valores de los campos. El patrón no tiene que hacer referencia a todos los campos del registro; los campos omitidos simplemente no participan en la coincidencia y no se extraen.

```
// This example uses a record pattern.

type MyRecord = { Name: string; ID: int }

let IsMatchByName record1 (name: string) =
  match record1 with
  | { MyRecord.Name = nameFound; MyRecord.ID = _; } when nameFound = name -> true
  | _ -> false

let recordX = { Name = "Parker"; ID = 10 }
let isMatched1 = IsMatchByName recordX "Parker"
let isMatched2 = IsMatchByName recordX "Hartono"
```

## Patrón de carácter comodín

El patrón de caracteres comodín se representa mediante el carácter de subrayado (`_`) y coincide con cualquier entrada, al igual que el patrón de variable, con la excepción de que la entrada se descarta en lugar de asignarla a una variable. El patrón de caracteres comodín se usa a menudo en otros patrones como un marcador de posición para los valores que no son necesarios en la expresión a la derecha del símbolo `->`. El patrón de caracteres comodín también se usa con frecuencia al final de una lista de patrones para buscar coincidencias con cualquier entrada no coincidente. El patrón de caracteres comodín se muestra en muchos ejemplos de código de este tema. Vea el código anterior para ver un ejemplo.

## Patrones que tienen anotaciones de tipo

Los patrones pueden tener anotaciones de tipo. Estos se comportan como otras anotaciones de tipo e inferencia de la guía como otras anotaciones de tipo. Se requieren paréntesis en torno a las anotaciones de tipo en los patrones. En el código siguiente se muestra un patrón que tiene una anotación de tipo.

```
let detect1 x =
  match x with
  | 1 -> printfn "Found a 1!"
  | (var1 : int) -> printfn "%d" var1
detect1 0
detect1 1
```



# Modelo de prueba de tipo

El modelo de prueba de tipo se usa para hacer coincidir la entrada con un tipo. Si el tipo de entrada es una coincidencia con (o un tipo derivado de) el tipo especificado en el patrón, la coincidencia se realiza correctamente.

En el siguiente ejemplo se muestra el modelo de prueba de tipo.

```
open System.Windows.Forms

let RegisterControl(control:Control) =
    match control with
    | :? Button as button -> button.Text <- "Registered."
    | :? CheckBox as checkbox -> checkbox.Text <- "Registered."
    | _ -> ()
```

Si solo está comprobando si un identificador es de un tipo derivado determinado, no necesita la parte `as identifier` del patrón, tal como se muestra en el ejemplo siguiente:

```
type A() = class end
type B() = inherit A()
type C() = inherit A()

let m (a: A) =
    match a with
    | :? B -> printfn "It's a B"
    | :? C -> printfn "It's a C"
    | _ -> ()
```

## Patrón null

El patrón null coincide con el valor null que puede aparecer cuando se trabaja con tipos que permiten un valor null. Los patrones NULL se utilizan con frecuencia al interoperar con código de .NET Framework. Por ejemplo, el valor devuelto de una API de .NET podría ser la entrada a una expresión `match`. Puede controlar el flujo del programa en función de si el valor devuelto es NULL y también de otras características del valor devuelto. Puede usar el patrón null para evitar que los valores NULL se propaguen al resto del programa.

En el ejemplo siguiente se usa el patrón NULL y el patrón variable.

```
let ReadFromFile (reader : System.IO.StreamReader) =
    match reader.ReadLine() with
    | null -> printfn "\n"; false
    | line -> printfn "%s" line; true

let fs = System.IO.File.Open("../Program.fs", System.IO.FileMode.Open)
let sr = new System.IO.StreamReader(fs)
while ReadFromFile(sr) = true do ()
sr.Close()
```

## Vea también

- [Expresiones de coincidencia](#)
- [Patrones activos](#)
- [Referencia del lenguaje F#](#)

# Patrones activos

23/07/2020 • 10 minutes to read • [Edit Online](#)

Los *modelos activos* permiten definir particiones con nombre que subdividen los datos de entrada, de modo que puede usar estos nombres en una expresión de coincidencia de patrones de la misma forma que lo haría para una Unión discriminada. Se pueden usar patrones activos para descomponer los datos de manera personalizada para cada partición.

## Sintaxis

```
// Active pattern of one choice.
let (|identifier|) [arguments] valueToMatch = expression

// Active Pattern with multiple choices.
// Uses a FSharp.Core.Choice<_,...,> based on the number of case names. In F#, the limitation n <= 7
// applies.
let (|identifier1|identifier2|...|) valueToMatch = expression

// Partial active pattern definition.
// Uses a FSharp.Core.option<_> to represent if the type is satisfied at the call site.
let (|identifier|_|) [arguments ] valueToMatch = expression
```

## Comentarios

En la sintaxis anterior, los identificadores son nombres para las particiones de los datos de entrada que se representan mediante *argumentoso*, en otras palabras, nombres para subconjuntos del conjunto de todos los valores de los argumentos. Puede haber hasta siete particiones en una definición de modelo activo. La *expresión* describe el formulario en el que se descomponen los datos. Puede usar una definición de modelo activa para definir las reglas para determinar a qué particiones con nombre se encuentran los valores especificados como argumentos. Los símbolos (| y |) se conocen como *clips de banana* y la función creada por este tipo de enlace Let se denomina *reconocedor activo*.

Como ejemplo, considere el siguiente patrón activo con un argumento.

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd
```

Puede usar el modelo activo en una expresión de coincidencia de patrones, como en el ejemplo siguiente.

```
let TestNumber input =
    match input with
    | Even -> printfn "%d is even" input
    | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

La salida de este programa es la siguiente:

```
7 is odd
11 is odd
32 is even
```

Otro uso de los patrones activos es descomponer los tipos de datos de varias maneras, por ejemplo, cuando los mismos datos subyacentes tienen varias representaciones posibles. Por ejemplo, un `Color` objeto se puede descomponer en una representación RGB o en una representación HSB.

```
open System.Drawing

let (|RGB|) (col : System.Drawing.Color) =
    ( col.R, col.G, col.B )

let (|HSB|) (col : System.Drawing.Color) =
    ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let printRGB (col: System.Drawing.Color) =
    match col with
    | RGB(r, g, b) -> printfn " Red: %d Green: %d Blue: %d" r g b

let printHSB (col: System.Drawing.Color) =
    match col with
    | HSB(h, s, b) -> printfn " Hue: %f Saturation: %f Brightness: %f" h s b

let printAll col colorString =
    printfn "%s" colorString
    printRGB col
    printHSB col

printAll Color.Red "Red"
printAll Color.Black "Black"
printAll Color.White "White"
printAll Color.Gray "Gray"
printAll Color.BlanchedAlmond "BlanchedAlmond"
```

La salida del programa anterior es la siguiente:

```
Red
Red: 255 Green: 0 Blue: 0
Hue: 360.000000 Saturation: 1.000000 Brightness: 0.500000
Black
Red: 0 Green: 0 Blue: 0
Hue: 0.000000 Saturation: 0.000000 Brightness: 0.000000
White
Red: 255 Green: 255 Blue: 255
Hue: 0.000000 Saturation: 0.000000 Brightness: 1.000000
Gray
Red: 128 Green: 128 Blue: 128
Hue: 0.000000 Saturation: 0.000000 Brightness: 0.501961
BlanchedAlmond
Red: 255 Green: 235 Blue: 205
Hue: 36.000000 Saturation: 1.000000 Brightness: 0.901961
```

En combinación, estas dos maneras de utilizar los modelos activos permiten particionar y descomponer los datos en el formulario adecuado y realizar los cálculos adecuados en los datos adecuados en el formulario más conveniente para el cálculo.

Las expresiones de coincidencia de patrones resultantes permiten escribir datos de una manera cómoda que sea muy legible, lo que simplifica considerablemente la bifurcación y el código de análisis de datos potencialmente complejos.

# Modelos activos parciales

A veces, solo necesita crear particiones de parte del espacio de entrada. En ese caso, se escribe un conjunto de patrones parciales cada uno de los cuales coinciden con algunas entradas pero no coinciden con otras entradas. Los modelos activos que no siempre generan un valor se denominan *modelos activos parciales*. tienen un valor devuelto que es un tipo de opción. Para definir un modelo activo parcial, se usa un carácter comodín ( `_` ) al final de la lista de patrones dentro de los clips de banana. En el código siguiente se muestra el uso de un modelo activo parcial.

```
let (|Integer|_|) (str: string) =
    let mutable intvalue = 0
    if System.Int32.TryParse(str, &intvalue) then Some(intvalue)
    else None

let (|Float|_|) (str: string) =
    let mutable floatvalue = 0.0
    if System.Double.TryParse(str, &floatvalue) then Some(floatvalue)
    else None

let parseNumeric str =
    match str with
    | Integer i -> printfn "%d : Integer" i
    | Float f -> printfn "%f : Floating point" f
    | _ -> printfn "%s : Not matched." str

parseNumeric "1.1"
parseNumeric "0"
parseNumeric "0.0"
parseNumeric "10"
parseNumeric "Something else"
```

La salida del ejemplo anterior es la siguiente:

```
1.100000 : Floating point
0 : Integer
0.000000 : Floating point
10 : Integer
Something else : Not matched.
```

Cuando se usan modelos activos parciales, algunas veces las opciones individuales pueden ser disjuntas o excluidas mutuamente, pero no es necesario. En el ejemplo siguiente, el cuadrado de patrón y el cubo de patrón no están separados, porque algunos números son cuadrados y cubos, como 64. En el siguiente programa se usa el patrón y para combinar los patrones de cuadrados y de cubo. Imprime todos los enteros hasta 1000 que son cuadrados y cubos, así como aquellos que son solo cubos.

```

let err = 1.e-10

let isNearlyIntegral (x:float) = abs (x - round(x)) < err

let (|Square|_|) (x : int) =
    if isNearlyIntegral (sqrt (float x)) then Some(x)
    else None

let (|Cube|_|) (x : int) =
    if isNearlyIntegral ((float x) ** ( 1.0 / 3.0)) then Some(x)
    else None

let findSquareCubes x =
    match x with
    | Cube x & Square _ -> printfn "%d is a cube and a square" x
    | Cube x -> printfn "%d is a cube" x
    | _ -> ()

[ 1 .. 1000 ] |> List.iter (fun elem -> findSquareCubes elem)

```

La salida es como sigue:

```

1 is a cube and a square
8 is a cube
27 is a cube
64 is a cube and a square
125 is a cube
216 is a cube
343 is a cube
512 is a cube
729 is a cube and a square
1000 is a cube

```

## Modelos activos con parámetros

Los modelos activos siempre toman al menos un argumento para el elemento que se está coincidentando, pero también pueden tomar argumentos adicionales, en cuyo caso se aplica el nombre del *modelo activo parametrizado*. Los argumentos adicionales permiten que un patrón general esté especializado. Por ejemplo, los patrones activos que usan expresiones regulares para analizar cadenas suelen incluir la expresión regular como parámetro adicional, como en el código siguiente, que también usa el modelo activo parcial `Integer` definido en el ejemplo de código anterior. En este ejemplo, se proporcionan cadenas que usan expresiones regulares para varios formatos de fecha para personalizar el modelo activo `ParseRegex` general. El modelo activo entero se usa para convertir las cadenas coincidentes en enteros que se pueden pasar al constructor `DateTime`.

```

open System.Text.RegularExpressions

// ParseRegex parses a regular expression and returns a list of the strings that match each group in
// the regular expression.
// List.tail is called to eliminate the first element in the list, which is the full matched expression,
// since only the matches for each group are wanted.
let (|ParseRegex|_|) regex str =
    let m = Regex(regex).Match(str)
    if m.Success
    then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None

// Three different date formats are demonstrated here. The first matches two-
// digit dates and the second matches full dates. This code assumes that if a two-digit
// date is provided, it is an abbreviation, not a year in the first century.
let parseDate str =
    match str with
    | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{1,2})$" [Integer m; Integer d; Integer y]
      -> new System.DateTime(y + 2000, m, d)
    | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{3,4})" [Integer m; Integer d; Integer y]
      -> new System.DateTime(y, m, d)
    | ParseRegex "(\d{1,4})-(\d{1,2})-(\d{1,2})" [Integer y; Integer m; Integer d]
      -> new System.DateTime(y, m, d)
    | _ -> new System.DateTime()

let dt1 = parseDate "12/22/08"
let dt2 = parseDate "1/1/2009"
let dt3 = parseDate "2008-1-15"
let dt4 = parseDate "1995-12-28"

printfn "%s %s %s %s" (dt1.ToString()) (dt2.ToString()) (dt3.ToString()) (dt4.ToString())

```

La salida del código anterior es la siguiente:

```

12/22/2008 12:00:00 AM 1/1/2009 12:00:00 AM 1/15/2008 12:00:00 AM 12/28/1995 12:00:00 AM

```

Los modelos activos no se limitan solo a las expresiones de coincidencia de patrones, sino que también se pueden usar en los enlaces Let.

```

let (|Default|) onNone value =
    match value with
    | None -> onNone
    | Some e -> e

let greet (Default "random citizen" name) =
    printfn "Hello, %s!" name

greet None
greet (Some "George")

```

La salida del código anterior es la siguiente:

```

Hello, random citizen!
Hello, George!

```

## Consulte también:

- [Referencia del lenguaje F#](#)
- [Expresiones de coincidencia](#)



# Bucles: expresión for...to

27/11/2019 • 2 minutes to read • [Edit Online](#)

La expresión `for...to` se usa para iterar en un bucle en un intervalo de valores de una variable de bucle.

## Sintaxis

```
for identifier = start [ to | downto ] finish do
  body-expression
```

## Comentarios

El tipo del identificador se deduce del tipo de las expresiones de *Inicio* y *finalización*. Los tipos de estas expresiones deben ser enteros de 32 bits.

A pesar de ser técnicamente una expresión, `for...to` es más similar a una instrucción tradicional en un lenguaje de programación imperativo. El tipo de valor devuelto para la *expresión Body* debe ser `unit`. En los ejemplos siguientes se muestran varios usos de la expresión `for...to`.

```
// A simple for...to loop.
let function1() =
  for i = 1 to 10 do
    printf "%d " i
  printfn ""

// A for...to loop that counts in reverse.
let function2() =
  for i = 10 downto 1 do
    printf "%d " i
  printfn ""

function1()
function2()

// A for...to loop that uses functions as the start and finish expressions.
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function3 x y =
  for i = (beginning x y) to (ending x y) do
    printf "%d " i
  printfn ""

function3 10 4
```

La salida del código anterior es la siguiente.

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

Vea también



- Referencia del lenguaje F#
- Bucles: expresión `for...in`
- Bucles: expresión `while...do`

# Bucles: expresión for...in

23/10/2019 • 6 minutes to read • [Edit Online](#)

Esta construcción de bucle se usa para iterar por las coincidencias de un patrón en una colección Enumerable como una expresión de intervalo, una secuencia, una lista, una matriz u otra construcción que admita la enumeración.

## Sintaxis

```
for pattern in enumerable-expression do
    body-expression
```

## Comentarios

La `for...in` expresión se puede comparar con la `for each` instrucción en otros lenguajes .net porque se usa para recorrer los valores de una colección Enumerable. Sin embargo `for...in`, también admite la coincidencia de patrones en la colección en lugar de simplemente la iteración en toda la colección.

La expresión Enumerable se puede especificar como una colección Enumerable o, mediante `..` el operador, como un intervalo en un tipo entero. Las colecciones enumerables incluyen listas, secuencias, matrices, conjuntos, asignaciones, etc. Se `System.Collections.IEnumerable` puede usar cualquier tipo que implemente.

Al expresar un intervalo mediante el `..` operador, puede usar la sintaxis siguiente.

*iniciar .. finish*

También puede usar una versión que incluya un incremento denominado *SKIP*, como en el código siguiente.

*iniciar .. omitir ... finish*

Cuando se usan intervalos enteros y una variable de contador simple como patrón, el comportamiento típico es incrementar la variable de contador en 1 en cada iteración, pero si el intervalo incluye un valor de omisión, el contador se incrementa en su lugar por el valor de omisión.

Los valores coincidentes en el patrón también se pueden utilizar en la expresión del cuerpo.

En los siguientes ejemplos de código se muestra el `for...in` uso de la expresión.

```
// Looping over a list.
let list1 = [ 1; 5; 100; 450; 788 ]
for i in list1 do
    printfn "%d" i
```

La salida es la siguiente.

```
1
5
100
450
788
```

En el ejemplo siguiente se muestra cómo crear un bucle sobre una secuencia y cómo usar un patrón de tupla en

lugar de una variable simple.

```
let seq1 = seq { for i in 1 .. 10 -> (i, i*i) }
for (a, asqr) in seq1 do
    printfn "%d squared is %d" a asqr
```

La salida es la siguiente.

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

En el ejemplo siguiente se muestra cómo recorrer un intervalo entero simple.

```
let function1() =
    for i in 1 .. 10 do
        printf "%d " i
    printfn ""
function1()
```

La salida de function1 es como se indica a continuación.

```
1 2 3 4 5 6 7 8 9 10
```

En el ejemplo siguiente se muestra cómo recorrer en bucle un intervalo con un salto de 2, que incluye todos los demás elementos del intervalo.

```
let function2() =
    for i in 1 .. 2 .. 10 do
        printf "%d " i
    printfn ""
function2()
```

La salida de `function2` es como se indica a continuación.

```
1 3 5 7 9
```

En el ejemplo siguiente se muestra cómo usar un intervalo de caracteres.

```
let function3() =
    for c in 'a' .. 'z' do
        printf "%c " c
    printfn ""
function3()
```

La salida de `function3` es como se indica a continuación.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

En el ejemplo siguiente se muestra cómo usar un valor de SKIP negativo para una iteración inversa.

```
let function4() =  
  for i in 10 .. -1 .. 1 do  
    printf "%d " i  
  printfn " ... Lift off!"  
function4()
```

La salida de `function4` es como se indica a continuación.

```
10 9 8 7 6 5 4 3 2 1 ... Lift off!
```

El principio y el final del intervalo también pueden ser expresiones, como las funciones, como en el código siguiente.

```
let beginning x y = x - 2*y  
let ending x y = x + 2*y  
  
let function5 x y =  
  for i in (beginning x y) .. (ending x y) do  
    printf "%d " i  
  printfn ""  
  
function5 10 4
```

La salida de `function5` con esta entrada es como se indica a continuación.

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

En el ejemplo siguiente se muestra el uso de un carácter comodín () cuando el elemento no es necesario en el bucle.

```
let mutable count = 0  
for _ in list1 do  
  count <- count + 1  
printfn "Number of elements in list1: %d" count
```

La salida es la siguiente.

```
Number of elements in list1: 5
```

**Note** Puede usar `for...in` en expresiones de secuencia y otras expresiones de cálculo, en cuyo caso se utiliza una versión personalizada `for...in` de la expresión. Para obtener más información, vea [secuencias](#), [flujos de trabajo asincrónicos](#) y [expresiones de cálculo](#).

## Vea también

- [Referencia del lenguaje F#](#)
- [Bucles](#) `for...to` [Expresiones](#)
- [Bucles](#) `while...do` [Expresiones](#)



# Bucles: expresión while...do

23/10/2019 • 2 minutes to read • [Edit Online](#)

La `while...do` expresión se usa para realizar la ejecución iterativa (en bucle) mientras se cumple una condición de prueba especificada.

## Sintaxis

```
while test-expression do
    body-expression
```

## Comentarios

Se evalúa *Test-Expression*; Si es `true` así, se ejecuta la *expresión Body* y se vuelve a evaluar la expresión de prueba. El *cuerpo-expresión* debe tener el `unit` tipo. Si la expresión de prueba `false` es, finaliza la iteración.

En el ejemplo siguiente se muestra el uso de `while...do` la expresión.

```
open System

let lookForValue value maxValue =
    let mutable continueLooping = true
    let randomNumberGenerator = new Random()
    while continueLooping do
        // Generate a random number between 1 and maxValue.
        let rand = randomNumberGenerator.Next(maxValue)
        printf "%d " rand
        if rand = value then
            printfn "\nFound a %d!" value
            continueLooping <- false

lookForValue 10 20
```

La salida del código anterior es un flujo de números aleatorios entre 1 y 20, el último de los cuales es 10.

```
13 19 8 18 16 2 10
Found a 10!
```

### NOTE

Puede usar `while...do` en expresiones de secuencia y otras expresiones de cálculo, en cuyo caso se utiliza una versión personalizada `while...do` de la expresión. Para obtener más información, vea [secuencias](#), [flujos de trabajo asíncronos](#) y [expresiones de cálculo](#).

## Vea también

- [Referencia del lenguaje F#](#)
- Bucles `for...in` [Expresiones](#)
- Bucles `for...to` [Expresiones](#)

# Aserciones

25/10/2019 • 2 minutes to read • [Edit Online](#)

La expresión de `assert` es una característica de depuración que puede usar para probar una expresión. En caso de error en modo de depuración, una aserción genera un cuadro de diálogo de error del sistema.

## Sintaxis

```
assert condition
```

## Comentarios

La expresión `assert` tiene el tipo `bool -> unit`.

La función `assert` se resuelve como `Debug.Assert`. Esto significa que su comportamiento es idéntico a haber llamado a `Debug.Assert` directamente.

La comprobación de aserciones solo está habilitada cuando se compila en modo de depuración; es decir, si se define la constante `DEBUG`. En el sistema del proyecto, de forma predeterminada, la constante `DEBUG` se define en la configuración de depuración, pero no en la configuración de lanzamiento.

El error de aserción no se puede detectar mediante el F# control de excepciones.

## Ejemplo

En el ejemplo de código siguiente se muestra el uso de la expresión `assert`.

```
let subtractUnsigned (x : uint32) (y : uint32) =  
    assert (x > y)  
    let z = x - y  
    z  
  
// This code does not generate an assertion failure.  
let result1 = subtractUnsigned 2u 1u  
// This code generates an assertion failure.  
let result2 = subtractUnsigned 1u 2u
```

## Vea también

- [Referencia del lenguaje F#](#)

# Control de excepciones

04/11/2019 • 2 minutes to read • [Edit Online](#)

Esta sección contiene información sobre la compatibilidad con el control de excepciones del lenguaje F#.

## Fundamentos del control de excepciones

El control de excepciones es la manera estándar de controlar las condiciones de error en .NET Framework. Por tanto, cualquier lenguaje de .NET debe admitir este mecanismo, incluso F#. Una *excepción* es un objeto que encapsula la información sobre un error. Cuando se producen errores, se generan excepciones y se detiene la ejecución regular. En su lugar, el tiempo de ejecución busca un controlador adecuado para la excepción. La búsqueda se inicia en la función actual y continúa hasta la pila a través de los niveles de llamadores hasta que se encuentra un controlador coincidente. Después, se ejecuta el controlador.

Además, cuando se desenreda la pila, el tiempo de ejecución ejecuta cualquier código en bloques `finally`, para garantizar que los objetos se limpien correctamente durante el proceso de desenredo.

## Temas relacionados

TITLE	DESCRIPCIÓN
<a href="#">Tipos de excepción</a>	Describe cómo declarar un tipo de excepción.
<a href="#">Exceptions: The <code>try...with</code> Expression</a> (Excepciones: la expresión <code>try...with</code> )	Describe la construcción de lenguaje que admite el control de excepciones.
<a href="#">Exceptions: The <code>try...finally</code> Expression</a> (Excepciones: la expresión <code>try...finally</code> )	Describe la construcción de lenguaje que permite ejecutar código de limpieza cuando se desenreda la pila al producirse una excepción.
<a href="#">Exceptions: the <code>raise</code> Function</a> (Excepciones: la función <code>raise</code> )	Describe cómo iniciar un objeto de excepción.
<a href="#">Exceptions: the <code>failwith</code> Function</a> (Excepciones: la función <code>failwith</code> )	Describe cómo generar una excepción general de F#.
<a href="#">Exceptions: the <code>invalidArg</code> Function</a> (Excepciones: la función <code>invalidArg</code> )	Describe cómo generar una excepción de argumento no válido.



# Tipos de excepción

23/10/2019 • 2 minutes to read • [Edit Online](#)

Existen dos categorías de excepciones en F#: los tipos de excepción de F#.net y los tipos de excepción. En este tema se describe cómo definir y usar tipos de excepciones.

## Sintaxis

```
exception exception-type of argument-type
```

## Comentarios

En la sintaxis anterior, *Exception-Type* es el nombre de un nuevo F# tipo de excepción y *argument-Type* representa el tipo de un argumento que se puede proporcionar cuando se produce una excepción de este tipo. Puede especificar varios argumentos mediante un tipo de tupla para *el tipo de argumento*.

Una definición típica para una F# excepción es similar a la siguiente.

```
exception MyError of string
```

Puede generar una excepción de este tipo mediante la `raise` función, como se indica a continuación.

```
raise (MyError("Error message"))
```

Puede usar un F# tipo de excepción directamente en los filtros de una `try...with` expresión, como se muestra en el ejemplo siguiente.

```
exception Error1 of string
// Using a tuple type as the argument type.
exception Error2 of string * int

let function1 x y =
    try
        if x = y then raise (Error1("x"))
        else raise (Error2("x", 10))
    with
        | Error1(str) -> printfn "Error1 %s" str
        | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

El tipo de excepción que se define con `exception` la palabra F# clave en es un nuevo tipo que hereda `System.Exception` de.

## Vea también

- [Control de excepciones](#)
- [Exceptions: the `raise` Function](#) (Excepciones: la función `raise`)
- [Jerarquía de excepciones](#)



# Excepciones: Expresión try...with

23/10/2019 • 5 minutes to read • [Edit Online](#)

En este tema se `try...with` describe la expresión, la expresión que se utiliza para el control F# de excepciones en el lenguaje.

## Sintaxis

```
try
    expression1
with
| pattern1 -> expression2
| pattern2 -> expression3
...
```

## Comentarios

La `try...with` expresión se utiliza para controlar las excepciones F#en. Es similar a la `try...catch` instrucción de C#. En la sintaxis anterior, el código de *expression1* podría generar una excepción. La `try...with` expresión devuelve un valor. Si no se produce ninguna excepción, toda la expresión devuelve el valor de *expression1*. Si se produce una excepción, cada *patrón* se compara a su vez con la excepción, y para el primer patrón coincidente, se ejecuta la *expresión* correspondiente, conocida como *controlador de excepciones*, para esa bifurcación y la expresión general Devuelve el valor de la expresión en ese controlador de excepciones. Si ningún patrón coincide, la excepción propaga la pila de llamadas hasta que se encuentra un controlador coincidente. Los tipos de los valores devueltos por cada expresión en los controladores de excepciones deben coincidir con el tipo devuelto `try` de la expresión en el bloque.

Con frecuencia, el hecho de que se produzca un error también significa que no hay ningún valor válido que se pueda devolver desde las expresiones de cada controlador de excepciones. Un patrón frecuente es que el tipo de la expresión sea un tipo de opción. En el ejemplo de código siguiente se muestra este patrón.

```
let divide1 x y =
    try
        Some (x / y)
    with
        | :? System.DivideByZeroException -> printfn "Division by zero!"; None

let result1 = divide1 100 0
```

Las excepciones pueden ser excepciones de .NET o pueden ser F# excepciones. Puede definir F# excepciones mediante la `exception` palabra clave.

Puede usar diversos patrones para filtrar según el tipo de excepción y otras condiciones; las opciones se resumen en la tabla siguiente.

MODELO	DESCRIPCIÓN
<code>:? exception-type</code>	Coincide con el tipo de excepción de .NET especificado.

MODELO	DESCRIPCIÓN
<i>?: tipo de excepción como identificador</i>	Coincide con el tipo de excepción de .NET especificado, pero asigna a la excepción un valor con nombre.
<i>nombre de excepción (argumentos)</i>	Coincide con F# un tipo de excepción y enlaza los argumentos.
<i>identifier</i>	Coincide con cualquier excepción y enlaza el nombre al objeto de excepción. Equivalente a <b>?: System. Exception como identificador</b>
<i>identificador cuando la condición</i>	Coincide con cualquier excepción si la condición es true.

## Ejemplos

En los siguientes ejemplos de código se muestra el uso de los distintos patrones de controlador de excepciones.

```
// This example shows the use of the as keyword to assign a name to a
// .NET exception.
let divide2 x y =
    try
        Some( x / y )
    with
        | :? System.DivideByZeroException as ex -> printfn "Exception! %s " (ex.Message); None

// This version shows the use of a condition to branch to multiple paths
// with the same exception.
let divide3 x y flag =
    try
        x / y
    with
        | ex when flag -> printfn "TRUE: %s" (ex.ToString()); 0
        | ex when not flag -> printfn "FALSE: %s" (ex.ToString()); 1

let result2 = divide3 100 0 true

// This version shows the use of F# exceptions.
exception Error1 of string
exception Error2 of string * int

let function1 x y =
    try
        if x = y then raise (Error1("x"))
        else raise (Error2("x", 10))
    with
        | Error1(str) -> printfn "Error1 %s" str
        | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

### NOTE

La `try...with` construcción es una expresión independiente de la `try...finally` expresión. Por lo tanto, si el código requiere `with` un bloque y `finally` un bloque, tendrá que anidar las dos expresiones.

#### NOTE

Puede usar `try...with` en flujos de trabajo asincrónicos y otras expresiones de cálculo, en cuyo caso se utiliza una versión personalizada `try...with` de la expresión. Para obtener más información, vea [flujos de trabajo asincrónicos](#) y expresiones de [cálculo](#).

## Vea también

- [Control de excepciones](#)
- [Tipos de excepción](#)
- [Excepciones: La `try...finally` expresión](#)

# Excepciones: Expresión try...finally

23/10/2019 • 3 minutes to read • [Edit Online](#)

La `try...finally` expresión permite ejecutar código de limpieza incluso si un bloque de código produce una excepción.

## Sintaxis

```
try
    expression1
finally
    expression2
```

## Comentarios

La `try...finally` expresión se puede utilizar para ejecutar el código de *expresión2* en la sintaxis anterior, independientemente de si se genera una excepción durante la ejecución de *expresión1*.

El tipo de *expresión2* no contribuye al valor de la expresión completa; el tipo devuelto cuando una excepción no se produce es el último valor de *expresión1*. Cuando se produce una excepción, no se devuelve ningún valor y el flujo de control se transfiere al siguiente controlador de excepción coincidente en la pila de llamadas. Si no se encuentra ningún controlador de excepciones, el programa finaliza. Antes de que se ejecute el código de un controlador coincidente o finalice el programa, se ejecuta el `finally` código de la bifurcación.

En el código siguiente se muestra el uso `try...finally` de la expresión.

```
let divide x y =
    let stream : System.IO.FileStream = System.IO.File.Create("test.txt")
    let writer : System.IO.StreamWriter = new System.IO.StreamWriter(stream)
    try
        writer.WriteLine("test1");
        Some( x / y )
    finally
        writer.Flush()
        printfn "Closing stream"
        stream.Close()

let result =
    try
        divide 100 0
    with
        | :? System.DivideByZeroException -> printfn "Exception handled."; None
```

La salida a la consola es la siguiente.

```
Closing stream
Exception handled.
```

Como puede ver en la salida, la secuencia se cerró antes de que se administrara la excepción externa y el `test.txt` archivo contiene el `test1` texto, lo que indica que los búferes se vaciaron y escribieron en el disco aunque la excepción se haya transferido. control al controlador de excepciones externo.

Tenga en cuenta `try...with` que la construcción es una construcción independiente `try...finally` de la construcción. Por lo tanto, si el código requiere `with` un bloque y `finally` un bloque, debe anidar las dos construcciones, como en el ejemplo de código siguiente.

```
exception InnerError of string
exception OuterError of string

let function1 x y =
    try
        try
            if x = y then raise (InnerError("inner"))
            else raise (OuterError("outer"))
        with
        | InnerError(str) -> printfn "Error1 %s" str
    finally
        printfn "Always print this."

let function2 x y =
    try
        function1 x y
    with
    | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

En el contexto de las expresiones de cálculo, incluidas las expresiones de secuencia y los flujos de trabajo asíncronos, **pruebe... las expresiones Finally** pueden tener una implementación personalizada. Para obtener más información, vea [expresiones de cálculo](#).

## Vea también

- [Control de excepciones](#)
- Excepciones: La `try...with` expresión

# Excepciones: función raise

23/10/2019 • 2 minutes to read • [Edit Online](#)

La `raise` función se usa para indicar que se ha producido un error o una condición excepcional. La información sobre el error se captura en un objeto de excepción.

## Sintaxis

```
raise (expression)
```

## Comentarios

La `raise` función genera un objeto de excepción e inicia un proceso de desenredado de la pila. El proceso de desenredado de la pila se administra mediante el Common Language Runtime (CLR), por lo que el comportamiento de este proceso es el mismo que en cualquier otro lenguaje de .NET. El proceso de desenredado de la pila es una búsqueda de un controlador de excepciones que coincide con la excepción generada. La búsqueda se inicia en la `try...with` expresión actual, si hay alguna. Cada patrón del `with` bloque se comprueba en orden. Cuando se encuentra un controlador de excepciones coincidente, la excepción se considera controlada; de lo contrario, la pila se desenreda `with` y se bloquea la cadena de llamadas hasta que se encuentra un controlador coincidente. Los `finally` bloques que se encuentran en la cadena de llamadas también se ejecutan en secuencia cuando se desenreda la pila.

La `raise` función es el equivalente de `throw` en C# o C++. Use `reraise` en un controlador catch para propagar la misma excepción hacia arriba en la cadena de llamadas.

En los siguientes ejemplos de código se muestra el `raise` uso de la función para generar una excepción.

```
exception InnerError of string
exception OuterError of string

let function1 x y =
    try
        try
            if x = y then raise (InnerError("inner"))
            else raise (OuterError("outer"))
        with
            | InnerError(str) -> printfn "Error1 %s" str
    finally
        printfn "Always print this."

let function2 x y =
    try
        function1 x y
    with
        | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

La `raise` función también se puede utilizar para generar excepciones de .net, tal y como se muestra en el ejemplo siguiente.



```
let divide x y =  
  if (y = 0) then raise (System.ArgumentException("Divisor cannot be zero!"))  
  else  
    x / y
```

## Vea también

- [Control de excepciones](#)
- [Tipos de excepción](#)
- [Excepciones: La `try...with` expresión](#)
- [Excepciones: La `try...finally` expresión](#)
- [Excepciones: La `failwith` función](#)
- [Excepciones: La `invalidArg` función](#)

# Excepciones: Expresión failwith

23/10/2019 • 2 minutes to read • [Edit Online](#)

La `failwith` función genera una F# excepción.

## Sintaxis

```
failwith error-message-string
```

## Comentarios

La *cadena de mensaje de error* en la sintaxis anterior es una cadena literal o un valor de tipo `string`. Se convierte en `Message` la propiedad de la excepción.

La `failwith` excepción generada por es una `System.Exception` excepción, que es una referencia que tiene el nombre `Failure` en F# el código. En el código siguiente se muestra el uso `failwith` de para producir una excepción.

```
let divideFailwith x y =
    if (y = 0) then failwith "Divisor cannot be zero."
    else
        x / y

let testDivideFailwith x y =
    try
        divideFailwith x y
    with
        | Failure(msg) -> printfn "%s" msg; 0

let result1 = testDivideFailwith 100 0
```

## Vea también

- [Control de excepciones](#)
- [Tipos de excepción](#)
- [Excepciones: La `try...with` expresión](#)
- [Excepciones: La `try...finally` expresión](#)
- [Exceptions: the `raise` Function](#) (Excepciones: la función `raise`)

# Excepciones: Función invalidArg

23/10/2019 • 2 minutes to read • [Edit Online](#)

La `invalidArg` función genera una excepción de argumento.

## Sintaxis

```
invalidArg parameter-name error-message-string
```

## Comentarios

El parámetro-name de la sintaxis anterior es una cadena con el nombre del parámetro cuyo argumento no era válido. La *cadena de mensaje de error* es una cadena literal o un valor de tipo `string`. Se convierte en `Message` la propiedad del objeto de excepción.

La excepción generada `invalidArg` por es `System.ArgumentException` una excepción. En el código siguiente se muestra el uso `invalidArg` de para producir una excepción.

```
let months = [| "January"; "February"; "March"; "April";  
               "May"; "June"; "July"; "August"; "September";  
               "October"; "November"; "December" |]  
  
let lookupMonth month =  
    if (month > 12 || month < 1)  
        then invalidArg "month" (sprintf "Value passed in was %d." month)  
    months.[month - 1]  
  
printfn "%s" (lookupMonth 12)  
printfn "%s" (lookupMonth 1)  
printfn "%s" (lookupMonth 13)
```

El resultado es el siguiente, seguido de un seguimiento de la pila (no se muestra).

```
December  
January  
System.ArgumentException: Month parameter out of range.
```

## Vea también

- [Control de excepciones](#)
- [Tipos de excepción](#)
- [Excepciones: La `try...with` expresión](#)
- [Excepciones: La `try...finally` expresión](#)
- [Exceptions: the `raise` Function](#) (Excepciones: la función `raise`)
- [Excepciones: La `failwith` función](#)

# Atributos

21/02/2020 • 7 minutes to read • [Edit Online](#)

Los atributos permiten aplicar metadatos a una construcción de programación.

## Sintaxis

```
[<target:attribute-name(arguments)>]
```

## Observaciones

En la sintaxis anterior, el *destino* es opcional y, si está presente, especifica el tipo de entidad de programa a la que se aplica el atributo. Los valores válidos para el *destino* se muestran en la tabla que aparece más adelante en este documento.

El *atributo-name* hace referencia al nombre (posiblemente calificado con espacios de nombres) de un tipo de atributo válido, con o sin el sufijo `Attribute` que normalmente se usa en los nombres de tipo de atributo. Por ejemplo, el tipo `ObsoleteAttribute` se puede acortar a simplemente `Obsolete` en este contexto.

Los *argumentos* son los argumentos del constructor para el tipo de atributo. Si un atributo tiene un constructor sin parámetros, se pueden omitir la lista de argumentos y los paréntesis. Los atributos admiten argumentos posicionales y argumentos con nombre. Los *argumentos posicionales* son argumentos que se usan en el orden en que aparecen. Se pueden usar argumentos con nombre si el atributo tiene propiedades públicas. Puede establecerlos mediante la siguiente sintaxis en la lista de argumentos.

```
property-name = property-value
```

Dichas inicializaciones de propiedad pueden estar en cualquier orden, pero deben seguir cualquier argumento posicional. El siguiente es un ejemplo de un atributo que usa argumentos posicionales e inicializaciones de propiedad:

```
open System.Runtime.InteropServices

[<DllImport("kernel32", SetLastError=true)>]
extern bool CloseHandle(nativeint handle)
```

En este ejemplo, el atributo es `DllImportAttribute`, que se usa en forma abreviada. El primer argumento es un parámetro posicional y el segundo es una propiedad.

Los atributos son una construcción de programación de .NET que permite asociar un objeto conocido como *atributo* a un tipo u otro elemento de programa. El elemento de programa al que se aplica un atributo se conoce como *destino de atributo*. El atributo normalmente contiene metadatos sobre su destino. En este contexto, los metadatos pueden ser cualquier dato sobre el tipo que no sea sus campos y miembros.

Los atributos F# se pueden aplicar a las siguientes construcciones de programación: funciones, métodos, ensamblados, módulos, tipos (clases, registros, estructuras, interfaces, delegados, enumeraciones, uniones, etc.), constructores, propiedades, campos, parámetros, parámetros de tipo y valores devueltos. No se permiten atributos en `let` enlaces dentro de clases, expresiones o expresiones de flujo de trabajo.

Normalmente, la declaración de atributo aparece directamente antes de la declaración del destino de atributo. Se pueden usar varias declaraciones de atributos juntas, como se indica a continuación:

```
[<Owner("Jason Carlson")>]
[<Company("Microsoft")>]
type SomeType1 =
```

Puede consultar los atributos en tiempo de ejecución mediante la reflexión de .NET.

Puede declarar varios atributos individualmente, como en el ejemplo de código anterior, o puede declararlos en un conjunto de corchetes si usa un punto y coma para separar los atributos y constructores individuales, como se indica a continuación:

```
[<Owner("Darren Parker"); Company("Microsoft")>]
type SomeType2 =
```

Normalmente, los atributos encontrados incluyen el atributo `Obsolete`, atributos para las consideraciones de seguridad, atributos para la compatibilidad con COM, atributos relacionados con la propiedad del código y atributos que indican si un tipo se puede serializar. En el siguiente ejemplo se muestra el uso del atributo

`Obsolete`.

```
open System

[<Obsolete("Do not use. Use newFunction instead.")>]
let obsoleteFunction x y =
    x + y

let newFunction x y =
    x + 2 * y

// The use of the obsolete function produces a warning.
let result1 = obsoleteFunction 10 100
let result2 = newFunction 10 100
```

En el caso de los destinos de atributo `assembly` y `module`, aplique los atributos a un enlace de `do` de nivel superior en el ensamblado. Puede incluir la palabra `assembly` o `module` en la declaración de atributo, como se indica a continuación:

```
open System.Reflection
[<assembly:AssemblyVersionAttribute("1.0.0.0")>]
do
    printfn "Executing..."
```

Si se omite el destino de atributo de un atributo aplicado a un enlace de `do` F#, el compilador intenta determinar el destino de atributo que tiene sentido para ese atributo. Muchas clases de atributos tienen un atributo de tipo `System.AttributeUsageAttribute` que incluye información sobre los posibles destinos admitidos para ese atributo. Si el `System.AttributeUsageAttribute` indica que el atributo admite funciones como destinos, se toma el atributo para aplicar al punto de entrada principal del programa. Si el `System.AttributeUsageAttribute` indica que el atributo admite ensamblados como destinos, el compilador toma el atributo para aplicar al ensamblado. La mayoría de los atributos no se aplican a las funciones y los ensamblados, pero en los casos en que lo hacen, el atributo se toma para aplicarse a la función principal del programa. Si el destino del atributo se especifica explícitamente, el atributo se aplica al destino especificado.

Aunque normalmente no es necesario especificar explícitamente el destino de atributo, en la tabla siguiente se muestran los valores válidos para el *destino* en un atributo junto con ejemplos de uso:

DESTINO DE ATRIBUTO	EJEMPLO
ensamblado	<pre>[&lt;assembly: AssemblyVersion("1.0.0.0")&gt;]</pre>
return	<pre>let function1 x : [&lt;return: MyCustomAttributeThatWorksOnReturns&gt;] int = x + 1</pre>
campo	<pre>[&lt;DefaultValue&gt;] val mutable x: int</pre>
propiedad	<pre>[&lt;Obsolete&gt;] this.MyProperty = x</pre>
param	<pre>member this.MyMethod([&lt;Out&gt;] x : ref&lt;int&gt;) = x := 10</pre>
type	<pre>[&lt;type: StructLayout(LayoutKind.Sequential)&gt;] type MyStruct =     struct         val x : byte         val y : int     end</pre>

## Consulte también

- [Referencia del lenguaje F#](#)

# Administración de recursos: La palabra clave use

23/10/2019 • 6 minutes to read • [Edit Online](#)

En este tema se describe `use` la palabra `using` clave y la función, que puede controlar la inicialización y la liberación de los recursos.

## Recursos

El término *recurso* se utiliza en más de una manera. Sí, los recursos pueden ser datos que una aplicación utiliza, como cadenas, gráficos y similares, pero en este contexto, *los recursos* hacen referencia a recursos de software o del sistema operativo, como contextos de dispositivo gráficos, identificadores de archivo, red y base de datos. conexiones, objetos de simultaneidad, como identificadores de espera, etc. El uso de estos recursos por parte de las aplicaciones implica la adquisición del recurso desde el sistema operativo u otro proveedor de recursos, seguido de la versión posterior del recurso al grupo para que se pueda proporcionar a otra aplicación. Se producen problemas cuando las aplicaciones no liberan recursos de nuevo en el grupo común.

## Administrar recursos

Para administrar de forma eficaz y eficaz los recursos de una aplicación, debe liberar los recursos de forma rápida y predecible. El .NET Framework le ayuda a hacerlo proporcionando la `System.IDisposable` interfaz. Un tipo que implementa `System.IDisposable` tiene el `System.IDisposable.Dispose` método, que libera los recursos correctamente. La garantía de aplicaciones bien escritas garantiza que `System.IDisposable.Dispose` se llama al método rápidamente cuando ya no se necesita cualquier objeto que tenga un recurso limitado. Afortunadamente, la mayoría de los lenguajes .NET proporcionan compatibilidad para facilitar F# esta tarea y no es ninguna excepción. Hay dos construcciones de lenguaje útiles que admiten el patrón de Dispose `use` : el enlace `using` y la función.

## usar enlace

La `use` palabra clave tiene un formato similar al `let` del enlace:

*usar expresión de valor =*

Proporciona la misma funcionalidad que un `let` enlace pero agrega una llamada a `Dispose` en el valor cuando el valor sale del ámbito. Tenga en cuenta que el compilador inserta una comprobación nula en el valor, de modo que `null` si el valor es `Dispose` , no se intenta la llamada a.

En el ejemplo siguiente se muestra cómo cerrar un archivo automáticamente mediante la `use` palabra clave.

```
open System.IO

let writetofile filename obj =
    use file1 = File.CreateText(filename)
    file1.WriteLine("{0}", obj.ToString() )
    // file1.Dispose() is called implicitly here.

writetofile "abc.txt" "Humpty Dumpty sat on a wall."
```

## NOTE

Puede usar `use` en expresiones de cálculo, en cuyo caso se utiliza una versión personalizada de `use` la expresión. Para obtener más información, vea [secuencias](#), [flujos de trabajo asincrónicos](#) y expresiones de [cálculo](#).

## Using (función)

La `using` función tiene el formato siguiente:

```
using (expression1) función o lambda
```

En una `using` expresión, *expression1* crea el objeto que se debe desechar. El resultado de *expression1* (el objeto que se debe desechar) se convierte en un argumento, *valor*, en *función o expresión lambda*, que es una función que espera un único argumento restante de un tipo que coincida con el valor generado por *expression1*, o una expresión lambda que espera un argumento de ese tipo. Al final de la ejecución de la función, el tiempo de ejecución `Dispose` llama a y libera los recursos (a menos que `null` el valor sea, en cuyo caso no se intenta la llamada a `Dispose`).

En el ejemplo siguiente se `using` muestra la expresión con una expresión lambda.

```
open System.IO

let writetofile2 filename obj =
    using (System.IO.File.CreateText(filename)) ( fun file1 ->
        file1.WriteLine("{0}", obj.ToString() )
    )

writetofile2 "abc2.txt" "The quick sly fox jumps over the lazy brown dog."
```

En el ejemplo siguiente se `using` muestra la expresión con una función.

```
let printToFile (file1 : System.IO.StreamWriter) =
    file1.WriteLine("Test output");

using (System.IO.File.CreateText("test.txt")) printToFile
```

Tenga en cuenta que la función puede ser una función que ya tiene algunos argumentos aplicados. El siguiente ejemplo de código muestra esto. Crea un archivo que contiene la cadena `XYZ`.

```
let printToFile2 obj (file1 : System.IO.StreamWriter) =
    file1.WriteLine(obj.ToString())

using (System.IO.File.CreateText("test.txt")) (printToFile2 "XYZ")
```

La `using` función y el `use` enlace son formas casi equivalentes de lograr lo mismo. La `using` palabra clave proporciona más control sobre `Dispose` Cuando se llama a. Cuando se usa `using`, `Dispose` se llama al final de la función o expresión lambda; cuando se usa la `use` palabra clave, `Dispose` se llama al final del bloque de código contenedor. En general, es preferible usar `use` en lugar de la `using` función.

## Vea también

- [Referencia del lenguaje F#](#)



# Espacios de nombres

06/05/2020 • 8 minutes to read • [Edit Online](#)

Un espacio de nombres permite organizar el código en áreas de funcionalidad relacionada, ya que permite adjuntar un nombre a una agrupación de elementos de programa de F#. Los espacios de nombres suelen ser elementos de nivel superior en archivos de F#.

## Sintaxis

```
namespace [rec] [parent-namespaces.]identifier
```

## Comentarios

Si desea colocar el código en un espacio de nombres, la primera declaración del archivo debe declarar el espacio de nombres. El contenido de todo el archivo se convierte entonces en parte del espacio de nombres, siempre que no exista otra declaración de espacios de nombres en el archivo. En ese caso, todo el código hasta la siguiente declaración de espacio de nombres se considera que está dentro del primer espacio de nombres.

Los espacios de nombres no pueden contener directamente valores y funciones. En su lugar, los valores y las funciones deben estar incluidos en los módulos y los módulos se incluyen en los espacios de nombres. Los espacios de nombres pueden contener tipos, módulos.

Los comentarios de documento XML se pueden declarar sobre un espacio de nombres, pero se omiten. Las directivas de compilador también se pueden declarar sobre un espacio de nombres.

Los espacios de nombres se pueden declarar explícitamente con la palabra clave `namespace` o implícitamente al declarar un módulo. Para declarar explícitamente un espacio de nombres, use la palabra clave `namespace` seguida del nombre del espacio de nombres. En el ejemplo siguiente se muestra un archivo de código que declara

`Widgets` un espacio de nombres con un tipo y un módulo incluido en dicho espacio de nombres.

```
namespace Widgets

type MyWidget1 =
    member this.WidgetName = "Widget1"

module WidgetsModule =
    let widgetName = "Widget2"
```

Si todo el contenido del archivo está en un módulo, también puede declarar espacios de nombres implícitamente mediante la `module` palabra clave y proporcionando el nuevo nombre de espacio de nombres en el nombre completo del módulo. En el ejemplo siguiente se muestra un archivo de código que declara `Widgets` un espacio de `WidgetsModule` nombres y un módulo, que contiene una función.

```
module Widgets.WidgetModule

let widgetFunction x y =
    printfn "%A %A" x y
```

El código siguiente es equivalente al código anterior, pero el módulo es una declaración de módulo local. En ese caso, el espacio de nombres debe aparecer en su propia línea.

```
namespace Widgets

module WidgetModule =

    let widgetFunction x y =
        printfn "%A %A" x y
```

Si se necesita más de un módulo en el mismo archivo en uno o varios espacios de nombres, debe usar las declaraciones del módulo local. Al utilizar las declaraciones del módulo local, no se puede usar el espacio de nombres completo en las declaraciones del módulo. En el código siguiente se muestra un archivo que tiene una declaración de espacio de nombres y dos declaraciones de módulos locales. En este caso, los módulos se incluyen directamente en el espacio de nombres; no hay ningún módulo creado implícitamente que tenga el mismo nombre que el archivo. Cualquier otro código del archivo, como un `do` enlace, está en el espacio de nombres pero no en los módulos internos, por lo que debe calificar el miembro `widgetFunction` del módulo mediante el nombre del módulo.

```
namespace Widgets

module WidgetModule1 =
    let widgetFunction x y =
        printfn "Module1 %A %A" x y
module WidgetModule2 =
    let widgetFunction x y =
        printfn "Module2 %A %A" x y

module useWidgets =

    do
        WidgetModule1.widgetFunction 10 20
        WidgetModule2.widgetFunction 5 6
```

La salida de este ejemplo es la siguiente.

```
Module1 10 20
Module2 5 6
```

Para obtener más información, vea [módulos](#).

## Espacios de nombres anidados

Al crear un espacio de nombres anidado, debe calificarlo por completo. De lo contrario, se crea un nuevo espacio de nombres de nivel superior. La sangría se omite en las declaraciones de espacio de nombres.

En el ejemplo siguiente se muestra cómo declarar un espacio de nombres anidado.

```
namespace Outer

    // Full name: Outer.MyClass
    type MyClass() =
        member this.X(x) = x + 1

// Fully qualify any nested namespaces.
namespace Outer.Inner

    // Full name: Outer.Inner.MyClass
    type MyClass() =
        member this.Prop1 = "X"
```

# Espacios de nombres en archivos y ensamblados

Los espacios de nombres pueden abarcar varios archivos en un único proyecto o compilación. El término *espacio de nombres* de términos describe la parte de un espacio de nombres que se incluye en un archivo. Los espacios de nombres también pueden abarcar varios ensamblados. Por ejemplo, el `System` espacio de nombres incluye todo el .NET Framework, que abarca muchos ensamblados y contiene muchos espacios de nombres anidados.

## Espacio de nombres global

El espacio de nombres `global` predefinido se usa para colocar nombres en el espacio de nombres de nivel superior de .net.

```
namespace global

type SomeType() =
    member this.SomeMember = 0
```

También puede usar `global` para hacer referencia al espacio de nombres .NET de nivel superior, por ejemplo, para resolver conflictos de nombres con otros espacios de nombres.

```
global.System.Console.WriteLine("Hello World!")
```

## Espacios de nombres recursivos

Los espacios de nombres también se pueden declarar como recursivos para permitir que todo el código contenido sea mutuamente recursivo. Esto se hace a `namespace rec` través de. El uso `namespace rec` de puede aliviar algunos problemas al no poder escribir código referencial mutuamente entre tipos y módulos. El siguiente es un ejemplo de esto:

```

namespace rec MutualReferences

type Orientation = Up | Down
type PeelState = Peeled | Unpeeled

// This exception depends on the type below.
exception DontSqueezeTheBananaException of Banana

type Banana(orientation : Orientation) =
  member val IsPeeled = false with get, set
  member val Orientation = orientation with get, set
  member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

  member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
  member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on the
exception above.

module BananaHelpers =
  let peel (b: Banana) =
    let flip (banana: Banana) =
      match banana.Orientation with
      | Up ->
        banana.Orientation <- Down
        banana
      | Down -> banana

    let peelSides (banana: Banana) =
      banana.Sides
      |> List.map (function
        | Unpeeled -> Peeled
        | Peeled -> Peeled)

    match b.Orientation with
    | Up -> b |> flip |> peelSides
    | Down -> b |> peelSides

```

Tenga en cuenta que `DontSqueezeTheBananaException` la excepción y `Banana` la clase hacen referencia entre sí. Además, el módulo `BananaHelpers` y la clase `Banana` también hacen referencia entre sí. No sería posible expresar en F # si se quita la `rec` palabra clave del `MutualReferences` espacio de nombres.

Esta característica también está disponible para los [módulos](#) de nivel superior.

## Vea también

- [Referencia del lenguaje F #](#)
- [Módulos](#)
- [F # RFC FS-1009: permite tipos y módulos mutuamente referenciables en ámbitos mayores en archivos](#)

# Módulos

06/05/2020 • 11 minutes to read • [Edit Online](#)

En el contexto del lenguaje F #, un *módulo* es una agrupación de código de f #, como valores, tipos y valores de función, en un programa de f #. Agrupar el código en módulos ayuda a mantener junto el código relacionado y a evitar conflictos de nombres en los programas.

## Sintaxis

```
// Top-level module declaration.  
module [accessibility-modifier] [qualified-namespace.]module-name  
declarations  
// Local module declaration.  
module [accessibility-modifier] module-name =  
    declarations
```

## Comentarios

Un módulo de F # es una agrupación de construcciones de código de F #, como tipos, valores, valores de función y `do` código en enlaces. Se implementa como una clase de Common Language Runtime (CLR) que solo tiene miembros estáticos. Hay dos tipos de declaraciones de módulo, dependiendo de si el archivo completo está incluido en el módulo: una declaración de módulo de nivel superior y una declaración de módulo local. Una declaración de módulo de nivel superior incluye el archivo completo en el módulo. Una declaración de módulo de nivel superior solo puede aparecer como la primera declaración de un archivo.

En la sintaxis de la declaración de módulo de nivel superior, el *espacio* de nombres completo opcional es la secuencia de nombres de espacios de nombres anidados que contiene el módulo. El espacio de nombres completo no tiene que declararse previamente.

No es necesario aplicar sangría a las declaraciones en un módulo de nivel superior. Tiene que aplicar sangría a todas las declaraciones en los módulos locales. En una declaración de módulo local, solo las declaraciones a las que se aplica la sangría en la declaración de módulo forman parte del módulo.

Si un archivo de código no comienza con una declaración de módulo de nivel superior o una declaración de espacio de nombres, todo el contenido del archivo, incluidos los módulos locales, pasa a formar parte de un módulo de nivel superior creado implícitamente que tiene el mismo nombre que el archivo, sin la extensión, con la primera letra convertida en mayúsculas. Por ejemplo, considere el siguiente archivo.

```
// In the file program.fs.  
let x = 40
```

Este archivo se compilaría como si estuviera escrito de esta manera:

```
module Program  
let x = 40
```

Si tiene varios módulos en un archivo, debe usar una declaración de módulo local para cada módulo. Si se declara un espacio de nombres envolvente, estos módulos forman parte del espacio de nombres envolvente. Si no se declara un espacio de nombres envolvente, los módulos forman parte del módulo de nivel superior creado implícitamente. En el ejemplo de código siguiente se muestra un archivo de código que contiene varios

módulos. El compilador crea implícitamente un módulo de nivel `MultipleModules` superior denominado

`MyModule1` y `MyModule2` y se anidan en ese módulo de nivel superior.

```
// In the file multiplemodules.fs.
// MyModule1
module MyModule1 =
    // Indent all program elements within modules that are declared with an equal sign.
    let module1Value = 100

    let module1Function x =
        x + 10

// MyModule2
module MyModule2 =

    let module2Value = 121

    // Use a qualified name to access the function.
    // from MyModule1.
    let module2Function x =
        x * (MyModule1.module1Function module2Value)
```

Si tiene varios archivos en un proyecto o en una única compilación, o si está compilando una biblioteca, debe incluir una declaración de espacio de nombres o una declaración de módulo en la parte superior del archivo. El compilador de F # solo determina implícitamente un nombre de módulo cuando solo hay un archivo en un proyecto o una línea de comandos de compilación, y se crea una aplicación.

El *modificador Accessibility* puede ser uno de los siguientes: `public`, `private`, `internal`. Para obtener más información, consulta [Access Control](#). El valor predeterminado es `public`.

## Referencia al código en módulos

Al hacer referencia a funciones, tipos y valores de otro módulo, debe usar un nombre completo o abrir el módulo. Si usa un nombre completo, debe especificar los espacios de nombres, el módulo y el identificador del elemento de programa que desee. Cada parte de la ruta de acceso calificada se separa con un punto (.), como se indica a continuación.

```
Namespace1.Namespace2.ModuleName.Identifier
```

Puede abrir el módulo o uno o varios de los espacios de nombres para simplificar el código. Para obtener más información sobre cómo abrir espacios de nombres y módulos, vea [Import declarations: open](#) [The keyword](#).

En el ejemplo de código siguiente se muestra un módulo de nivel superior que contiene todo el código hasta el final del archivo.

```
module Arithmetic

let add x y =
    x + y

let sub x y =
    x - y
```

Para usar este código desde otro archivo en el mismo proyecto, use nombres completos o abra el módulo antes de usar las funciones, como se muestra en los ejemplos siguientes.

```
// Fully qualify the function name.
let result1 = Arithmetic.add 5 9
// Open the module.
open Arithmetic
let result2 = add 5 9
```

## Módulos anidados

Los módulos se pueden anidar. Se debe aplicar sangría a los módulos internos en lo que se refiere a las declaraciones de módulos externos para indicar que son módulos internos, no módulos nuevos. Por ejemplo, compare los dos ejemplos siguientes. Module `Z` es un módulo interno en el código siguiente.

```
module Y =
  let x = 1

  module Z =
    let z = 5
```

Pero el `Z` módulo es un elemento relacionado `Y` con el módulo en el código siguiente.

```
module Y =
  let x = 1

module Z =
  let z = 5
```

El `Z` módulo también es un módulo relacionado en el código siguiente, ya que no tiene ninguna sangría en lo que se refiere a otras `Y` declaraciones del módulo.

```
module Y =
  let x = 1

  module Z =
    let z = 5
```

Por último, si el módulo externo no tiene ninguna declaración y va seguido inmediatamente de otra declaración de módulo, se supone que la nueva declaración de módulo es un módulo interno, pero el compilador le advierte si la segunda definición de módulo no tiene una sangría más alejada que la primera.

```
// This code produces a warning, but treats Z as a inner module.
module Y =
module Z =
  let z = 5
```

Para eliminar la advertencia, aplique una sangría al módulo interno.

```
module Y =
  module Z =
    let z = 5
```

Si desea que todo el código de un archivo esté en un módulo externo único y desea que los módulos internos, el módulo externo no requiera el signo igual y no se tenga que aplicar sangría a las declaraciones, incluidas las declaraciones de módulo internas, que irán en el módulo externo. Es necesario aplicar sangría a las declaraciones dentro de las declaraciones del módulo interno. En el código siguiente se muestra este caso.

```
// The top-level module declaration can be omitted if the file is named
// TopLevel.fs or topLevel.fs, and the file is the only file in an
// application.
module TopLevel

let topLevelX = 5

module Inner1 =
    let inner1X = 1
module Inner2 =
    let inner2X = 5
```

## Módulos recursivos

F # 4,1 presenta el concepto de módulos que permiten que todo el código contenido sea recursivo. Esto se hace a `module rec` través de. El uso `module rec` de puede aliviar algunos problemas al no poder escribir código referencial mutuamente entre tipos y módulos. El siguiente es un ejemplo de esto:

```
module rec RecursiveModule =
    type Orientation = Up | Down
    type PeelState = Peeled | Unpeeled

    // This exception depends on the type below.
    exception DontSqueezeTheBananaException of Banana

    type Banana(orientation : Orientation) =
        member val IsPeeled = false with get, set
        member val Orientation = orientation with get, set
        member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled ] with get, set

        member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
        member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends
on the exception above.

    module BananaHelpers =
        let peel (b: Banana) =
            let flip (banana: Banana) =
                match banana.Orientation with
                | Up ->
                    banana.Orientation <- Down
                    banana
                | Down -> banana

            let peelSides (banana: Banana) =
                banana.Sides
                |> List.map (function
                    | Unpeeled -> Peeled
                    | Peeled -> Peeled)

            match b.Orientation with
            | Up -> b |> flip |> peelSides
            | Down -> b |> peelSides
```

Tenga en cuenta que `DontSqueezeTheBananaException` la excepción y `Banana` la clase hacen referencia entre sí. Además, el módulo `BananaHelpers` y la clase `Banana` también hacen referencia entre sí. Esto no sería posible si se hubiera quitado la `rec` palabra clave del `RecursiveModule` módulo en F #.

Esta capacidad también es posible en los [espacios de nombres](#) con F # 4,1.

Vea también



- [Referencia del lenguaje F #](#)
- [Espacios de nombres](#)
- [F # RFC FS-1009: permite tipos y módulos mutuamente referenciables en ámbitos mayores en archivos](#)

# Declaraciones de importación: la palabra clave `open`

22/04/2020 • 6 minutes to read • [Edit Online](#)

## NOTE

Los vínculos de la referencia de API de este artículo le llevarán a MSDN. La referencia de API de docs.microsoft.com no está completa.

Una declaración de *importación* especifica un módulo o espacio de nombres cuyos elementos se pueden hacer referencia sin usar un nombre completo.

## Sintaxis

```
open module-or-namespace-name
```

## Observaciones

Hacer referencia a código mediante el espacio de nombres completo o la ruta de acceso del módulo cada vez puede crear código que sea difícil de escribir, leer y mantener. En su lugar, `open` puede usar la palabra clave para módulos y espacios de nombres usados con frecuencia para que al hacer referencia a un miembro de ese módulo o espacio de nombres, pueda usar la forma abreviada del nombre en lugar del nombre completo. Esta palabra clave `using` es similar a `using namespace` la palabra clave `Imports` en C- , en Visual C+ y en Visual Basic.

El módulo o espacio de nombres proporcionado debe estar en el mismo proyecto o en un proyecto o ensamblado al que se hace referencia. Si no es así, puede agregar una referencia `-reference` al proyecto o utilizar la `-r` opción de línea de comandos (o su abreviatura). Para obtener más información, consulte [Opciones del compilador](#).

La declaración de importación hace que los nombres estén disponibles en el código que sigue a la declaración, hasta el final del espacio de nombres, módulo o archivo envolvente.

Cuando se utilizan varias declaraciones de importación, deben aparecer en líneas independientes.

El código siguiente muestra `open` el uso de la palabra clave para simplificar el código.

```
// Without the import declaration, you must include the full
// path to .NET Framework namespaces such as System.IO.
let writeFile1 filename (text: string) =
    let stream1 = new System.IO.FileStream(filename, System.IO.FileMode.Create)
    let writer = new System.IO.StreamWriter(stream1)
    writer.WriteLine(text)

// Open a .NET Framework namespace.
open System.IO

// Now you do not have to include the full paths.
let writeFile2 filename (text: string) =
    let stream1 = new FileStream(filename, FileMode.Create)
    let writer = new StreamWriter(stream1)
    writer.WriteLine(text)

writeFile2 "file1.txt" "Testing..."
```

El compilador de F no emite un error o una advertencia cuando se producen ambigüedades cuando se produce el mismo nombre en más de un módulo abierto o espacio de nombres. Cuando se producen ambigüedades, F- da preferencia al módulo o espacio de nombres abierto más recientemente. Por ejemplo, en el `empty` `Seq.empty` código siguiente, significa , aunque `empty` se encuentra en los `List` módulos y. `Seq`

```
open List
open Seq
printfn "%A" empty
```

Por lo tanto, tenga cuidado al `List` abrir `Seq` módulos o espacios de nombres como o que contienen miembros que tienen nombres idénticos; en su lugar, considere el uso de los nombres calificados. Debe evitar cualquier situación en la que el código dependa del orden de las declaraciones de importación.

## Espacios de nombres abiertos por defecto

Algunos espacios de nombres se usan con tanta frecuencia en el código de F que se abren implícitamente sin necesidad de una declaración de importación explícita. En la tabla siguiente se muestran los espacios de nombres que están abiertos de forma predeterminada.

ESPACIO DE NOMBRES	DESCRIPCIÓN
<code>Microsoft.FSharp.Core</code>	Contiene definiciones de tipos básicas de <code>int</code> <code>F</code> <code>float</code> para tipos integrados, como y .
<code>Microsoft.FSharp.Core.Operators</code>	Contiene operaciones aritméticas básicas como <code>+</code> y <code>*</code> .
<code>Microsoft.FSharp.Collections</code>	Contiene clases de colección inmutables como <code>List</code> y <code>Array</code> .
<code>Microsoft.FSharp.Control</code>	Contiene tipos para construcciones de control como la evaluación diferida y flujos de trabajo asincrónicos.
<code>Microsoft.FSharp.Text</code>	Contiene funciones para E/S <code>printf</code> con formato, como la función.

## Atributo AutoOpen

Puede aplicar `AutoOpen` el atributo a un ensamblado si desea abrir automáticamente un espacio de nombres o módulo cuando se hace referencia al ensamblado. También puede aplicar `AutoOpen` el atributo a un módulo para abrir automáticamente ese módulo cuando se abre el módulo primario o el espacio de nombres. Para obtener más información, vea [Core.AutoOpenAttribute \(Clase\)](#).

## Atributo RequireQualifiedAccess

Algunos módulos, registros o `RequireQualifiedAccess` tipos de unión pueden especificar el atributo. Al hacer referencia a elementos de esos módulos, registros o uniones, debe usar un nombre completo independientemente de si incluye una declaración de importación. Si utiliza este atributo estratégicamente en tipos que definen nombres de uso común, ayuda a evitar colisiones de nombres y, por lo tanto, hace que el código sea más resistente a los cambios en las bibliotecas. Para obtener más información, vea [Core.RequireQualifiedAccessAttribute \(Clase\)](#).

## Vea también

- [Referencia del lenguaje f](#)
- [Espacios de nombres](#)
- [Módulos](#)

# Prototipos

23/10/2019 • 9 minutes to read • [Edit Online](#)

Un archivo de signatura contiene información sobre las signaturas públicas de un conjunto de elementos de programa F# como, por ejemplo, tipos, espacios de nombres y módulos. Puede usarse para especificar la accesibilidad de estos elementos de programa.

## Comentarios

Para cada archivo de código F#, puede tener un *archivo de signatura*, que es un archivo que tiene el mismo nombre que el archivo de código, pero con la extensión .fsi en lugar de .fs. Los archivos de signatura también se pueden agregar a la compilación de línea de comandos si usa la línea de comandos directamente. Para distinguir entre los archivos de código y de signatura, a veces los archivos de código se denominan *archivos de implementación*. En un proyecto, el archivo de signatura debe preceder al archivo de código asociado.

Un archivo de signatura describe los espacios de nombres, módulos, tipos y miembros en el archivo de implementación correspondiente. Use la información de un archivo de signatura para especificar a qué partes del código de la implementación correspondiente se puede tener acceso desde el código de fuera del archivo de implementación, así como y qué elementos son internos para el archivo de implementación. Los espacios de nombres, módulos y tipos que se incluyen en el archivo de signatura deben ser un subconjunto de los espacios de nombres, módulos y tipos que se incluyen en el archivo de implementación. Con algunas excepciones que se describen más adelante en este tema, los elementos de lenguaje que no aparecen en el archivo de signatura se consideran privados en el archivo de implementación. Si no se encuentra ningún archivo de signatura en el proyecto o en la línea de comandos, se usará la accesibilidad predeterminada.

Para obtener más información sobre la accesibilidad predeterminada, vea [Access Control](#).

En un archivo de firma no se repite la definición de los tipos y las implementaciones de cada método o función. En su lugar, use la signatura de cada método y función, que actúa como una especificación completa de la funcionalidad que se implementa mediante un fragmento del módulo o espacio de nombres. La sintaxis de una signatura de tipo es la misma que la que se usa en las declaraciones de método abstractas de interfaces y clases abstractas. Esta sintaxis también se muestra en IntelliSense y en el archivo fsi.exe intérprete de F# cuando muestra la entrada compilada correctamente.

Si no hay suficiente información en la signatura de tipo para indicar si un tipo está sellado, o si se trata de un tipo de interfaz, debe agregar un atributo que indica la naturaleza del tipo para el compilador. En la tabla siguiente se describen los atributos que se usan para este propósito.

ATRIBUTO	DESCRIPCIÓN
[<Sealed>]	Para un tipo que no tiene ningún miembro abstracto o que no debe ampliarse.
[<Interface>]	Para un tipo que es una interfaz.

El compilador genera un error si los atributos no son coherentes entre la signatura y la declaración del archivo de implementación.

Use la palabra clave `val` para crear una firma para un valor o valor de función. La palabra clave `type` presenta una signatura de tipo.

Puede generar un archivo de signatura mediante la opción del compilador `--sig`. Por lo general, los archivos .fsi no se escriben manualmente. En su lugar, los archivos .fsi se generan mediante el compilador. A continuación, se agregan al proyecto, si tiene uno, y se editan quitando los métodos y las funciones que no desea que estén accesibles.

Existen varias reglas para las signaturas de tipo:

- Las abreviaturas de tipo de un archivo de implementación no deben coincidir con un tipo sin abreviatura en un archivo de signatura.
- Los registros y uniones discriminadas deben exponer todos sus campos o ninguno. Asimismo, los constructores y el orden de la signatura deben coincidir con el orden en el archivo de implementación. Las clases pueden revelar algunos, todos o ninguno de los campos y métodos de la signatura.
- Las clases y estructuras que tienen constructores deben exponer las declaraciones de sus clases base (declaración `inherits`). Además, las clases y estructuras que tienen constructores deben exponer todas sus declaraciones de interfaz y métodos abstractos.
- Los tipos de interfaz deben revelar todos sus métodos e interfaces.

Las reglas de las signaturas de valor son las siguientes:

- Los modificadores de accesibilidad (`public`, `internal`, etc.) y los modificadores de la signatura `inline` y `mutable` deben coincidir con los de la implementación.
- El número de parámetros de tipo genérico (inferidos de manera implícita o declarados de manera explícita) deben coincidir. Asimismo, los tipos y las restricciones de los parámetros de tipo genérico también deben coincidir.
- Si se usa el atributo `Literal`, este debe aparecer en la signatura y en la implementación, y debe usarse el mismo valor literal para ambos.
- El patrón de parámetros (también conocido como *aridad*) de las signaturas y las implementaciones debe ser coherente.
- Si los nombres de parámetro de un archivo de signatura difieren del archivo de implementación correspondiente, se usará el nombre del archivo de signatura, lo que puede causar problemas al depurar o generar perfiles. Si desea recibir notificaciones de no coincidencia, habilite la advertencia 3218 en el archivo del proyecto o al invocar al compilador (vea `--warnon` en [Opciones del compilador](#)).

En el ejemplo de código siguiente se muestra un ejemplo de archivo de signatura que tiene el espacio de nombres, el módulo, el valor de la función y las signaturas de tipo junto con los atributos adecuados. También muestra el archivo de implementación correspondiente.

```
// Module1.fsi

namespace Library1
module Module1 =
    val function1 : int -> int
    type Type1 =
        new : unit -> Type1
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Sealed>]
    type Type2 =
        new : unit -> Type2
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

En el código siguiente se muestra el archivo de implementación.

```
namespace Library1

module Module1 =

    let function1 x = x + 1

    type Type1() =
        member type1.method1() =
            printfn "type1.method1"
        member type1.method2() =
            printfn "type1.method2"

    [<Sealed>]
    type Type2() =
        member type2.method1() =
            printfn "type2.method1"
        member type2.method2() =
            printfn "type2.method2"

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

## Vea también

- [Referencia del lenguaje F#](#)
- [Control de acceso](#)
- [Opciones del compilador](#)

# Unidades de medida

23/10/2019 • 13 minutes to read • [Edit Online](#)

Los valores de punto flotante y de entero F# con signo en pueden tener asociadas unidades de medida, que se suelen usar para indicar la longitud, el volumen, la masa, etc. Mediante el uso de cantidades con unidades, se habilita el compilador para comprobar que las relaciones aritméticas tienen las unidades correctas, lo que ayuda a evitar errores de programación.

## Sintaxis

```
[<Measure>] type unit-name [ = measure ]
```

## Comentarios

La sintaxis anterior define el *nombre de unidad* como unidad de medida. El elemento opcional se utiliza para definir una nueva medida en términos de unidades definidas previamente. Por ejemplo, la línea siguiente define la medida `cm` (centímetro).

```
[<Measure>] type cm
```

En la línea siguiente se define `ml` la medida (milliliter) como un centímetro `cm^3` cúbico ().

```
[<Measure>] type ml = cm^3
```

En la sintaxis anterior, *Measure* es una fórmula que implica unidades. En las fórmulas que implican unidades, se admiten las potencias enteras (positivas y negativas), los espacios entre las `*` unidades indican un producto de las dos unidades `/`, también indica un producto de unidades e indica un cociente de unidades. En el caso de una unidad recíproca, puede usar una potencia entera negativa o una `/` que indique una separación entre el numerador y el denominador de una fórmula de unidad. Varias unidades del denominador deben ir entre paréntesis. Las unidades separadas por espacios después `/` de un se interpretan como parte del denominador, pero las unidades que siguen `*` a un se interpretan como parte del numerador.

Puede usar 1 en expresiones unitarias, ya sea solo para indicar una cantidad sin dimensiones, o junto con otras unidades, como en el numerador. Por ejemplo, las unidades de una tasa se escribirían como `1/s`, donde `s` indica segundos. No se usan paréntesis en las fórmulas unitarias. No se especifican constantes de conversión numéricas en las fórmulas unitarias; sin embargo, puede definir constantes de conversión con unidades por separado y usarlas en cálculos con comprobación unitaria.

Las fórmulas de unidad que significan lo mismo pueden escribirse de varias formas equivalentes. Por lo tanto, el compilador convierte las fórmulas unitarias en un formato coherente, que convierte las potencias negativas en recíprocos, agrupa las unidades en un solo numerador y un denominador, y alfabeticiza las unidades en el numerador y el denominador.

Por ejemplo, las fórmulas `kg m s^-2` de `m /s s * kg` unidad y se convierten en `kg m/s^2`.

En las expresiones de punto flotante se usan unidades de medida. El uso de números de punto flotante junto con unidades de medida asociadas agrega otro nivel de seguridad de tipos y ayuda a evitar los errores de no coincidencia de unidad que se pueden producir en las fórmulas cuando se utilizan números de punto flotante



débilmente tipados. Si escribe una expresión de punto flotante que utiliza unidades, las unidades de la expresión deben coincidir.

Puede anotar literales con una fórmula de unidad entre corchetes angulares, tal como se muestra en los ejemplos siguientes.

```
1.0<cm>
55.0<miles/hour>
```

No se coloca un espacio entre el número y el corchete angular; sin embargo, puede incluir un sufijo `f` literal como, como en el ejemplo siguiente.

```
// The f indicates single-precision floating point.
55.0f<miles/hour>
```

Esta anotación cambia el tipo del literal de su tipo primitivo (`float` como) a un tipo de dimensión, `float<cm>` como o, en este caso, `float<miles/hour>`. Una anotación de unidad de `<1>` indica una cantidad sin dimensiones y su tipo es equivalente al tipo primitivo sin un parámetro de unidad.

El tipo de una unidad de medida es un tipo de punto flotante o entero con signo junto con una anotación de unidad adicional, indicado entre corchetes. Por lo tanto, al escribir el tipo de una conversión `g` de (gramos) `kg` a (kilogramos), se describen los tipos de la siguiente manera.

```
let convertg2kg (x : float<g>) = x / 1000.0<g/kg>
```

Las unidades de medida se utilizan para la comprobación unitaria en tiempo de compilación pero no se conservan en el entorno en tiempo de ejecución. Por lo tanto, no afectan al rendimiento.

Las unidades de medida se pueden aplicar a cualquier tipo, no solo a los tipos de punto flotante; sin embargo, solo los tipos de punto flotante, los tipos enteros con signo y los tipos decimales admiten cantidades dimensionales. Por lo tanto, solo tiene sentido utilizar unidades de medida en los tipos primitivos y en los agregados que contienen estos tipos primitivos.

En el ejemplo siguiente se muestra el uso de unidades de medida.

```

// Mass, grams.
[<Measure>] type g
// Mass, kilograms.
[<Measure>] type kg
// Weight, pounds.
[<Measure>] type lb

// Distance, meters.
[<Measure>] type m
// Distance, cm
[<Measure>] type cm

// Distance, inches.
[<Measure>] type inch
// Distance, feet
[<Measure>] type ft

// Time, seconds.
[<Measure>] type s

// Force, Newtons.
[<Measure>] type N = kg m / s^2

// Pressure, bar.
[<Measure>] type bar
// Pressure, Pascals
[<Measure>] type Pa = N / m^2

// Volume, milliliters.
[<Measure>] type ml
// Volume, liters.
[<Measure>] type L

// Define conversion constants.
let gramsPerKilogram : float<g kg^-1> = 1000.0<g/kg>
let cmPerMeter : float<cm/m> = 100.0<cm/m>
let cmPerInch : float<cm/inch> = 2.54<cm/inch>

let mlPerCubicCentimeter : float<ml/cm^3> = 1.0<ml/cm^3>
let mlPerLiter : float<ml/L> = 1000.0<ml/L>

// Define conversion functions.
let convertGramsToKilograms (x : float<g>) = x / gramsPerKilogram
let convertCentimetersToInches (x : float<cm>) = x / cmPerInch

```

En el ejemplo de código siguiente se muestra cómo convertir un número de punto flotante no dimensional en un valor de punto flotante de dimensión. Solo tiene que multiplicar por 1,0 y aplicar las dimensiones a 1,0. Puede abstraerlo en una función como `degreesFahrenheit`.

Además, cuando pase valores con dimensiones a funciones que esperan números de punto flotante sin dimensiones, debe cancelar las unidades o convertirlos en `float` mediante el `float` operador. En este ejemplo, se divide por `1.0<degC>` para los argumentos en `printf` porque `printf` espera cantidades no dimensionales.

```
[<Measure>] type degC // temperature, Celsius/Centigrade
[<Measure>] type degF // temperature, Fahrenheit

let convertCtoF ( temp : float<degC> ) = 9.0<degF> / 5.0<degC> * temp + 32.0<degF>
let convertFtoC ( temp: float<degF> ) = 5.0<degC> / 9.0<degF> * ( temp - 32.0<degF>)

// Define conversion functions from dimensionless floating point values.
let degreesFahrenheit temp = temp * 1.0<degF>
let degreesCelsius temp = temp * 1.0<degC>

printfn "Enter a temperature in degrees Fahrenheit."
let input = System.Console.ReadLine()
let parsedOk, floatValue = System.Double.TryParse(input)
if parsedOk
    then
        printfn "That temperature in Celsius is %8.2f degrees C." ((convertFtoC (degreesFahrenheit
floatValue))/(1.0<degC>))
    else
        printfn "Error parsing input."
```

En la sesión de ejemplo siguiente se muestran los resultados de las entradas y en este código.

```
Enter a temperature in degrees Fahrenheit.
90
That temperature in degrees Celsius is    32.22.
```

## Uso de unidades genéricas

Puede escribir funciones genéricas que operan en datos que tienen una unidad de medida asociada. Para ello, especifique un tipo junto con una unidad genérica como parámetro de tipo, tal y como se muestra en el ejemplo de código siguiente.

```
// Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

let genericSumUnits ( x : float<'u>) (y: float<'u>) = x + y

let v1 = 3.1<m/s>
let v2 = 2.7<m/s>
let x1 = 1.2<m>
let t1 = 1.0<s>

// OK: a function that has unit consistency checking.
let result1 = genericSumUnits v1 v2
// Error reported: mismatched units.
// Uncomment to see error.
// let result2 = genericSumUnits v1 x1
```

## Crear tipos de agregado con unidades genéricas

En el código siguiente se muestra cómo crear un tipo de agregado que consta de valores de punto flotante individuales que tienen unidades que son genéricas. Esto permite crear un único tipo que funciona con una variedad de unidades. Además, las unidades genéricas conservan la seguridad de tipos asegurándose de que un tipo genérico que tiene un conjunto de unidades es un tipo diferente del mismo tipo genérico con un conjunto de unidades diferente. La base de esta técnica es que el `Measure` atributo se puede aplicar al parámetro de tipo.

```
// Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

// Define a vector together with a measure type parameter.
// Note the attribute applied to the type parameter.
type vector3D[<Measure>] 'u' = { x : float<'u>; y : float<'u>; z : float<'u>}

// Create instances that have two different measures.
// Create a position vector.
let xvec : vector3D<m> = { x = 0.0<m>; y = 0.0<m>; z = 0.0<m> }
// Create a velocity vector.
let v1vec : vector3D<m/s> = { x = 1.0<m/s>; y = -1.0<m/s>; z = 0.0<m/s> }
```

## Unidades en tiempo de ejecución

Las unidades de medida se utilizan para la comprobación de tipos estáticos. Cuando se compilan los valores de punto flotante, se eliminan las unidades de medida, por lo que las unidades se pierden en tiempo de ejecución. Por lo tanto, no es posible cualquier intento de implementar la funcionalidad que depende de la comprobación de las unidades en tiempo de ejecución. Por ejemplo, no es `ToString` posible implementar una función para imprimir las unidades.

## Conversiones

Para convertir un tipo que tiene unidades (por ejemplo, `float<'u>`) en un tipo que no tiene unidades, puede usar la función de conversión estándar. Por ejemplo, puede usar `float` para convertir en un `float` valor que no tiene unidades, como se muestra en el código siguiente.

```
[<Measure>]
type cm
let length = 12.0<cm>
let x = float length
```

Para convertir un valor sin unidades en un valor que tiene unidades, puede multiplicar por un valor 1 o 1,0 anotado con las unidades adecuadas. Sin embargo, para escribir capas de interoperabilidad, también hay algunas funciones explícitas que puede usar para convertir valores sin unidades en valores con unidades. Se encuentran en el módulo [Microsoft.FSharp.Core.LanguagePrimitives](#). Por ejemplo, para convertir de una unidad `float` a una `float<cm>`, use `floatwithmeasure` (, tal y como se muestra en el código siguiente.

```
open Microsoft.FSharp.Core
let height:float<cm> = LanguagePrimitives.FloatWithMeasure x
```

## Unidades de medida en la F# biblioteca principal

Una biblioteca de unidades está disponible en `FSharp.Data.UnitSystems.SI` el espacio de nombres. Incluye las unidades si en su forma de símbolo (como `m` para el medidor) en `UnitSymbols` el subespacio de nombres y su nombre completo ( `meter` como para el medidor) `UnitNames` en el subespacio de nombres.

## Vea también

- [Referencia del lenguaje F#](#)

# Documentación XML

08/01/2020 • 6 minutes to read • [Edit Online](#)

Puede generar documentación a partir de los comentarios de código de barra diagonal triple (`/ F#//`) en. Los comentarios XML pueden preceder a las declaraciones de los archivos de código (. FS) o de firma (. FSI).

## Generación de documentación a partir de comentarios

La compatibilidad con F# para generar documentación a partir de comentarios es la misma que en otros lenguajes .NET Framework. Como en otros lenguajes .NET Framework, la [opción del compilador-doc](#) le permite generar un archivo XML que contiene información que puede convertir en documentación mediante una herramienta como [DocFX](#) o [Sandcastle](#). La documentación que se genera mediante herramientas diseñadas para su uso con ensamblados escritos en otros lenguajes .NET Framework generalmente produce una vista de las API que se basa en la forma compilada de F# construcciones. A menos que las F#herramientas admitan específicamente, la documentación generada por F# estas herramientas no coincide con la vista de una API.

Para obtener más información sobre cómo generar documentación a partir de XML, vea [comentarios \(de# documentación XML\)guía de programación de C](#).

## Etiquetas recomendadas

Hay dos maneras de escribir comentarios de documentación XML. Uno consiste simplemente en escribir la documentación directamente en un Comentario de barra diagonal triple, sin usar etiquetas XML. Si lo hace, se toma todo el texto del comentario como la documentación de Resumen de la construcción de código que sigue inmediatamente. Utilice este método cuando desee escribir solo un breve resumen de cada construcción. El otro método es usar etiquetas XML para proporcionar documentación más estructurada. El segundo método permite especificar notas independientes para un breve resumen, comentarios adicionales, documentación para cada parámetro y parámetro de tipo y excepciones que se producen, y una descripción del valor devuelto. En la tabla siguiente se describen las etiquetas XML que F# se reconocen en los comentarios de código XML.

SINTAXIS DE ETIQUETAS	DESCRIPCIÓN
<code>&lt;c&gt; texto &lt;/c&gt;</code>	Especifica que el <i>texto</i> es código. Los generadores de documentación pueden usar esta etiqueta para mostrar texto en una fuente que sea adecuada para el código.
<code>&lt;summary&gt; texto &lt;/Summary&gt;</code>	Especifica que el <i>texto</i> es una breve descripción del elemento de programa. La descripción suele ser una o dos oraciones.
<code>&lt;remarks&gt; texto &lt;/remarks&gt;</code>	Especifica que el <i>texto</i> contiene información adicional sobre el elemento de programa.
<code>&lt;param name = " Name"&gt; Descripción &lt;/param Returns&gt;</code>	Especifica el nombre y la descripción de un parámetro de función o método.
<code>&lt;typeparam name = " Name"&gt; Descripción &lt;/typeparam&gt;</code>	Especifica el nombre y la descripción de un parámetro de tipo.
<code>&lt;returns&gt; texto &lt;/returns&gt;</code>	Especifica que el <i>texto</i> describe el valor devuelto de una función o un método.

SINTAXIS DE ETIQUETAS	DESCRIPCIÓN
<code>&lt;Exception CREF = " tipo"&gt; Descripción &lt;/Exception&gt;</code>	Especifica el tipo de excepción que se puede generar y las circunstancias en las que se produce.
<code>&lt;vea CREF = " referencia"&gt; texto &lt;/See&gt;</code>	Especifica un vínculo insertado a otro elemento de programa. La <i>referencia</i> es el nombre tal y como aparece en el archivo de documentación XML. El <i>texto</i> es el texto que se muestra en el vínculo.
<code>&lt;seeAlso CREF = " referencia"/&gt;</code>	Especifica un vínculo ver también a la documentación de otro tipo. La <i>referencia</i> es el nombre tal y como aparece en el archivo de documentación XML. Vea también los vínculos normalmente aparecen en la parte inferior de una página de documentación.
<code>&lt;para&gt; text &lt;/para&gt;</code>	Especifica un párrafo de texto. Se usa para separar texto dentro de la etiqueta <b>comentarios</b> .

## Ejemplo

### Descripción

A continuación se encuentra un Comentario de documentación XML típico en un archivo de signatura.

### Código

```
/// <summary>Builds a new string whose characters are the results of applying the function <c>mapping</c>
/// to each of the characters of the input string and concatenating the resulting
/// strings.</summary>
/// <param name="mapping">The function to produce a string from each character of the input string.</param>
///<param name="str">The input string.</param>
///<returns>The concatenated string.</returns>
///<exception cref="System.ArgumentNullException">Thrown when the input string is null.</exception>
val collect : (char -> string) -> string -> string
```

## Ejemplo

### Descripción

En el ejemplo siguiente se muestra el método alternativo, sin etiquetas XML. En este ejemplo, todo el texto del comentario se considera un resumen. Tenga en cuenta que si no especifica explícitamente una etiqueta de Resumen, no debe especificar otras etiquetas, como **param** o **Return Tags**.

### Código

```
/// Creates a new string whose characters are the result of applying
/// the function mapping to each of the characters of the input string
/// and concatenating the resulting strings.
val collect : (char -> string) -> string -> string
```

## Vea también

- [Referencia del lenguaje F#](#)
- [Opciones del compilador](#)

# Expresiones diferidas

23/10/2019 • 2 minutes to read • [Edit Online](#)

Las *expresiones diferidas* son expresiones que no se evalúan inmediatamente, pero se evalúan en su lugar cuando se necesita el resultado. Esto puede ayudar a mejorar el rendimiento del código.

## Sintaxis

```
let identifier = lazy ( expression )
```

## Comentarios

En la sintaxis anterior, *Expression* es un código que solo se evalúa cuando se requiere un resultado y *Identifier* es un valor que almacena el resultado. El valor es de tipo `Lazy<'T>`, donde el tipo real que se utiliza para `'T` se determina a partir del resultado de la expresión.

Las expresiones diferidas permiten mejorar el rendimiento mediante la restricción de la ejecución de una expresión a solo aquellas situaciones en las que se necesita un resultado.

Para forzar que se realicen las expresiones, llame al `Force` método. `Force` hace que la ejecución se realice solo una vez. Las siguientes llamadas `Force` a devuelven el mismo resultado, pero no ejecutan ningún código.

En el código siguiente se muestra el uso de expresiones diferidas y el `Force` uso de. En este código, el tipo de `result` es `Lazy<int>` y el `Force` método devuelve un `int`.

```
let x = 10
let result = lazy (x + 10)
printfn "%d" (result.Force())
```

La evaluación diferida, pero `Lazy` no el tipo, también se usa para las secuencias. Para obtener más información, vea [secuencias](#).

## Vea también

- [Referencia del lenguaje F#](#)
- [Módulo LazyExtensions](#)

# Expresiones de cálculo

19/03/2020 • 24 minutes to read • [Edit Online](#)

Las expresiones de cálculo en F- proporcionan una sintaxis conveniente para escribir cálculos que se pueden secuenciar y combinar mediante construcciones de flujo de control y enlaces. Dependiendo del tipo de expresión de cálculo, se pueden considerar como una forma de expresar monads, monoides, transformadores de monad y funtores aplicativos. Sin embargo, a diferencia de otros lenguajes (como la notación de *doente* en Haskell), no están vinculados a una sola abstracción y no se basan en macros u otras formas de metaprogramación para lograr una sintaxis conveniente y sensible al contexto.

## Información general

Los cálculos pueden tomar muchas formas. La forma más común de cálculo es la ejecución de un solo subproceso, que es fácil de entender y modificar. Sin embargo, no todas las formas de cálculo son tan sencillas como la ejecución de un solo subproceso. Estos son algunos ejemplos:

- Cálculos no deterministas
- Cálculos asincrónicos
- Cálculos eficaces
- Cálculos generativos

De forma más general, hay cálculos *contextuales* que debe realizar en determinadas partes de una aplicación. Escribir código sensible al contexto puede ser difícil, ya que es fácil "filtrar" cálculos fuera de un contexto determinado sin abstracciones para evitar que lo haga. Estas abstracciones a menudo son difíciles de escribir por sí mismo, razón por la cual F tiene una forma generalizada de hacerlo **llamadas expresiones de cálculo**.

Las expresiones de cálculo ofrecen un modelo uniforme de sintaxis y abstracción para codificar cálculos contextuales.

Cada expresión de cálculo está respaldada por un tipo *de generador*. El tipo de generador define las operaciones que están disponibles para la expresión de cálculo. Consulte Creación de [un nuevo tipo de expresión de cálculo](#), que muestra cómo crear una expresión de cálculo personalizada.

### Información general sobre la sintaxis

Todas las expresiones de cálculo tienen la siguiente forma:

```
builder-expr { cexper }
```

donde `builder-expr` es el nombre de un tipo de `cexper` generador que define la expresión de cálculo y es el cuerpo de expresión de la expresión de cálculo. Por ejemplo, `async` el código de expresión de cálculo puede tener este aspecto:

```
let fetchAndDownload url =
  async {
    let! data = downloadData url

    let processedData = processData data

    return processedData
  }
```



Hay una sintaxis especial y adicional disponible dentro de una expresión de cálculo, como se muestra en el ejemplo anterior. Los siguientes formularios de expresión son posibles con expresiones de cálculo:

```
expr { let! ... }
expr { do! ... }
expr { yield ... }
expr { yield! ... }
expr { return ... }
expr { return! ... }
expr { match! ... }
```

Cada una de estas palabras clave y otras palabras clave estándar de F- solo están disponibles en una expresión de cálculo si se han definido en el tipo de generador de respaldo. La única excepción `match!` a esto es, que es `let!` en sí mismo azúcar sintáctico para el uso de seguido de una coincidencia de patrón en el resultado.

El tipo de generador es un objeto que define métodos especiales que rigen la forma en que se combinan los fragmentos de la expresión de cálculo; es decir, sus métodos controlan cómo se comporta la expresión de cálculo. Otra forma de describir una clase de generador es decir que permite personalizar el funcionamiento de muchas construcciones de F, como bucles y enlaces.

`let!`

La `let!` palabra clave enlaza el resultado de una llamada a otra expresión de cálculo a un nombre:

```
let doThingsAsync url =
    async {
        let! data = getDataAsync url
        ...
    }
```

Si enlaza la llamada a `let` una expresión de cálculo con `,` no obtendrá el resultado de la expresión de cálculo. En su lugar, habrá enlazado el valor de la llamada *no realizada* a esa expresión de cálculo. Se `let!` usa para enlazar al resultado.

`let!` se define `Bind(x, f)` por el miembro en el tipo de generador.

`do!`

La `do!` palabra clave es para llamar `unit` a una expresión `zero` de cálculo que devuelve un tipo -like (definido por el miembro en el generador):

```
let doThingsAsync data url =
    async {
        do! submitData data url
        ...
    }
```

Para el flujo de `Async<unit>` trabajo [asíncronico](#), este tipo es `.` Para otras expresiones de cálculo, `CExpType<unit>` es probable que el tipo sea `.`

`do!` se define `Bind(x, f)` por el miembro en `f` el `unit` tipo de generador, donde produce un archivo `.`

`yield`

La `yield` palabra clave es para devolver un valor de la expresión `IEnumerable<T>` de cálculo para que se pueda consumir como:

```

let squares =
  seq {
    for i in 1..10 do
      yield i * i
  }

for sq in squares do
  printfn "%d" sq

```

En la mayoría de los casos, puede ser omitido por los llamadores. La forma más `yield` común de `->` omitir es con el operador:

```

let squares =
  seq {
    for i in 1..10 -> i * i
  }

for sq in squares do
  printfn "%d" sq

```

Para expresiones más complejas que pueden producir muchos valores diferentes, y tal vez condicionalmente, simplemente omitir la palabra clave puede hacer:

```

let weekdays includeWeekend =
  seq {
    "Monday"
    "Tuesday"
    "Wednesday"
    "Thursday"
    "Friday"
    if includeWeekend then
      "Saturday"
      "Sunday"
  }

```

Al igual que con la [palabra clave yield en C](#), cada elemento de la expresión de cálculo se devuelve a medida que se itera.

`yield` se define `yield(x)` por el miembro en `x` el tipo de generador, donde está el elemento para devolver.

`yield!`

La `yield!` palabra clave es para acoplar una colección de valores de una expresión de cálculo:

```

let squares =
    seq {
        for i in 1..3 -> i * i
    }

let cubes =
    seq {
        for i in 1..3 -> i * i * i
    }

let squaresAndCubes =
    seq {
        yield! squares
        yield! cubes
    }

printfn "%A" squaresAndCubes // Prints - 1; 4; 9; 1; 8; 27

```

Cuando se evalúa, la `yield!` expresión de cálculo llamada por hará que sus elementos se retraen uno por uno, aplanando el resultado.

`yield!` se define `YieldFrom(x)` por el miembro en `x` el tipo de generador, donde hay una colección de valores.

A `yield` `yield!` diferencia de , debe especificarse explícitamente. Su comportamiento no está implícito en las expresiones de cálculo.

`return`

La `return` palabra clave ajusta un valor en el tipo correspondiente a la expresión de cálculo. Aparte de `yield` las expresiones de cálculo que utilizan , se utiliza para "completar" una expresión de cálculo:

```

let req = // 'req' is of type is 'Async<data>'
    async {
        let! data = fetch url
        return data
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req

```

`return` se define `Return(x)` por el miembro en `x` el tipo de generador, donde está el elemento que se va a ajustar.

`return!`

La `return!` palabra clave realiza el valor de una expresión de cálculo y ajusta el tipo correspondiente a la expresión de cálculo:

```

let req = // 'req' is of type is 'Async<data>'
    async {
        return! fetch url
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req

```

`return!` se define `ReturnFrom(x)` por el miembro en `x` el tipo de generador, donde hay otra expresión de cálculo.

`match!`

La `match!` palabra clave le permite inlinear una llamada a otra expresión de cálculo y coincidencia de patrón en su resultado:

```
let doThingsAsync url =
  async {
    match! callService url with
    | Some data -> ...
    | None -> ...
  }
```

Al llamar a `match!` una expresión de cálculo con `let!`, se dará cuenta del resultado de la llamada como `.` Esto se utiliza a menudo cuando se llama a una expresión de cálculo donde el resultado es un [archivo](#).

## Expresiones de cálculo integradas

La biblioteca principal de F - define tres expresiones de cálculo integradas: [Expresiones](#) de secuencia, Flujos de [trabajo asincrónicos](#) y [Expresiones](#) de consulta.

## Creación de un nuevo tipo de expresión de cálculo

Puede definir las características de sus propias expresiones de cálculo creando una clase de generador y definiendo ciertos métodos especiales en la clase. La clase de generador puede definir opcionalmente los métodos enumerados en la tabla siguiente.

En la tabla siguiente se describen los métodos que se pueden usar en una clase de generador de flujo de trabajo.

MÉTODO	FIRMA(S) TÍPICA(S)	DESCRIPCIÓN
<code>Bind</code>	<code>M&lt;'T&gt; * ('T -&gt; M&lt;'U&gt;) -&gt; M&lt;'U&gt;</code>	Llamado <code>let!</code> para <code>do!</code> y en expresiones de cálculo.
<code>Delay</code>	<code>(unit -&gt; M&lt;'T&gt;) -&gt; M&lt;'T&gt;</code>	Ajusta una expresión de cálculo como una función.
<code>Return</code>	<code>'T -&gt; M&lt;'T&gt;</code>	Se <code>return</code> llama en expresiones de cálculo.
<code>ReturnFrom</code>	<code>M&lt;'T&gt; -&gt; M&lt;'T&gt;</code>	Se <code>return!</code> llama en expresiones de cálculo.
<code>Run</code>	<code>M&lt;'T&gt; -&gt; M&lt;'T&gt;</code> o <code>M&lt;'T&gt; -&gt; 'T</code>	Ejecuta una expresión de cálculo.
<code>Combine</code>	<code>M&lt;'T&gt; * M&lt;'T&gt; -&gt; M&lt;'T&gt;</code> o <code>M&lt;unit&gt; * M&lt;'T&gt; -&gt; M&lt;'T&gt;</code>	Se ha llamado para la secuenciación en expresiones de cálculo.

MÉTODO	FIRMA(S) TÍPICA(S)	DESCRIPCIÓN
For	<pre>seq&lt;'T&gt; * ('T -&gt; M&lt;'U&gt;) -&gt; M&lt;'U&gt;</pre> <p>O</p> <pre>seq&lt;'T&gt; * ('T -&gt; M&lt;'U&gt;) -&gt; seq&lt;M&lt;'U&gt;&gt;</pre>	Se <code>for...do</code> ha llamado a expresiones en expresiones de cálculo.
TryFinally	<pre>M&lt;'T&gt; * (unit -&gt; unit) -&gt; M&lt;'T&gt;</pre>	Se <code>try...finally</code> ha llamado a expresiones en expresiones de cálculo.
TryWith	<pre>M&lt;'T&gt; * (exn -&gt; M&lt;'T&gt;) -&gt; M&lt;'T&gt;</pre>	Se <code>try...with</code> ha llamado a expresiones en expresiones de cálculo.
Using	<pre>'T * ('T -&gt; M&lt;'U&gt;) -&gt; M&lt;'U&gt;</pre> <pre>when 'T :&gt; IDisposable</pre>	Se <code>use</code> llama para enlaces en expresiones de cálculo.
While	<pre>(unit -&gt; bool) * M&lt;'T&gt; -&gt; M&lt;'T&gt;</pre>	Se <code>while...do</code> ha llamado a expresiones en expresiones de cálculo.
Yield	<pre>'T -&gt; M&lt;'T&gt;</pre>	Se <code>yield</code> ha llamado a expresiones en expresiones de cálculo.
YieldFrom	<pre>M&lt;'T&gt; -&gt; M&lt;'T&gt;</pre>	Se <code>yield!</code> ha llamado a expresiones en expresiones de cálculo.
Zero	<pre>unit -&gt; M&lt;'T&gt;</pre>	Se llama <code>else</code> a <code>if...then</code> ramas vacías de expresiones en expresiones de cálculo.
Quote	<pre>Quotations.Expr&lt;'T&gt; -&gt; Quotations.Expr&lt;'T&gt;</pre>	Indica que la expresión de <code>Run</code> cálculo se pasa al miembro como una cita. Traduce todas las instancias de un cálculo en una cita.

Muchos de los métodos de una `M<'T>` clase de generador usan y devuelven una construcción, que suele ser `Async<'T>` un tipo `Seq<'T>` definido por separado que caracteriza el tipo de cálculos que se combinan, por ejemplo, para flujos de trabajo asincrónicos y para flujos de trabajo de secuencia. Las firmas de estos métodos permiten combinarlas y anidarlas entre sí, de modo que el objeto de flujo de trabajo devuelto por una construcción se puede pasar a la siguiente. El compilador, cuando analiza una expresión de cálculo, convierte la expresión en una serie de llamadas a funciones anidadas mediante los métodos de la tabla anterior y el código de la expresión de cálculo.

La expresión anidada tiene la siguiente forma:

```
builder.Run(builder.Delay(fun () -> { | cexpr | })))
```

En el código anterior, `Run` `Delay` las llamadas y se omiten si no están definidas en la clase del generador de expresiones de cálculo. El cuerpo de la expresión de `{ | cexpr | }` cálculo, aquí denotado como , se traduce en llamadas que implican los métodos de la clase de generador por las traducciones descritas en la tabla siguiente. La expresión `{ | cexpr | }` de cálculo se define de `expr` forma recursiva según estas traducciones, donde es una expresión de F y `cexpr` es una expresión de cálculo.

EXPRESSION	TRADUCCIÓN
<code>{ let binding in cexpr }</code>	<code>let binding in {   cexpr   }</code>
<code>{ let! pattern = expr in cexpr }</code>	<code>builder.Bind(expr, (fun pattern -&gt; {   cexpr   }))</code>
<code>{ do! expr in cexpr }</code>	<code>builder.Bind(expr, (fun () -&gt; {   cexpr   }))</code>
<code>{ yield expr }</code>	<code>builder.Yield(expr)</code>
<code>{ yield! expr }</code>	<code>builder.YieldFrom(expr)</code>
<code>{ return expr }</code>	<code>builder.Return(expr)</code>
<code>{ return! expr }</code>	<code>builder.ReturnFrom(expr)</code>
<code>{ use pattern = expr in cexpr }</code>	<code>builder.Using(expr, (fun pattern -&gt; {   cexpr   }))</code>
<code>{ use! value = expr in cexpr }</code>	<code>builder.Bind(expr, (fun value -&gt; builder.Using(value, (fun value -&gt; { cexpr }))))</code>
<code>{ if expr then cexpr0 }</code>	<code>if expr then { cexpr0 } else builder.Zero()</code>
<code>{ if expr then cexpr0 else cexpr1 }</code>	<code>if expr then { cexpr0 } else { cexpr1 }</code>
<code>{ match expr with   pattern_i -&gt; cexpr_i }</code>	<code>match expr with   pattern_i -&gt; { cexpr_i }</code>
<code>{ for pattern in expr do cexpr }</code>	<code>builder.For(enumeration, (fun pattern -&gt; { cexpr }))</code>
<code>{ for identifier = expr1 to expr2 do cexpr }</code>	<code>builder.For(enumeration, (fun identifier -&gt; { cexpr })))</code>
<code>{ while expr do cexpr }</code>	<code>builder.While(fun () -&gt; expr, builder.Delay({ cexpr })))</code>
<code>{ try cexpr with   pattern_i -&gt; expr_i }</code>	<code>builder.TryWith(builder.Delay({ cexpr }), (fun value -&gt; match value with   pattern_i -&gt; expr_i   exn -&gt; reraise exn)))</code>
<code>{ try cexpr finally expr }</code>	<code>builder.TryFinally(builder.Delay( { cexpr } ), (fun ( ) -&gt; expr))</code>
<code>{ cexpr1; cexpr2 }</code>	<code>builder.Combine({ cexpr1 }, { cexpr2 })</code>
<code>{ other-expr; cexpr }</code>	<code>expr; { cexpr }</code>
<code>{ other-expr }</code>	<code>expr; builder.Zero()</code>

En la tabla `other-expr` anterior, se describe una expresión que no aparece de otro modo en la tabla. Una clase de generador no necesita implementar todos los métodos y admitir todas las traducciones enumeradas en la tabla anterior. Las construcciones que no se implementan no están disponibles en expresiones de cálculo de ese tipo. Por ejemplo, si no desea `use` admitir la palabra clave en las `use` expresiones de cálculo, puede omitir la definición de en la clase de generador.

En el ejemplo de código siguiente se muestra una expresión de cálculo que encapsula un cálculo como una serie de pasos que se pueden evaluar paso a paso. Un tipo de `OkOrException` unión discriminada, , codifica el estado de error de la expresión tal como se ha evaluado hasta ahora. Este código muestra varios patrones típicos que puede usar en las expresiones de cálculo, como las implementaciones reutilizables de algunos de los métodos del generador.

```
// Computations that can be run step by step
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)

module Eventually =
    // The bind for the computations. Append 'func' to the
    // computation.
    let rec bind func expr =
        match expr with
        | Done value -> func value
        | NotYetDone work -> NotYetDone (fun () -> bind func (work()))

    // Return the final value wrapped in the Eventually type.
    let result value = Done value

    type OkOrException<'T> =
        | Ok of 'T
        | Exception of System.Exception

    // The catch for the computations. Stitch try/with throughout
    // the computation, and return the overall result as an OkOrException.
    let rec catch expr =
        match expr with
        | Done value -> result (Ok value)
        | NotYetDone work ->
            NotYetDone (fun () ->
                let res = try Ok(work()) with | exn -> Exception exn
                match res with
                | Ok cont -> catch cont // note, a tailcall
                | Exception exn -> result (Exception exn))

    // The delay operator.
    let delay func = NotYetDone (fun () -> func())

    // The stepping action for the computations.
    let step expr =
        match expr with
        | Done _ -> expr
        | NotYetDone func -> func ()

    // The rest of the operations are boilerplate.
    // The tryFinally operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryFinally expr compensation =
        catch (expr)
        |> bind (fun res ->
            compensation();
            match res with
            | Ok value -> result value
            | Exception exn -> raise exn)

    // The tryWith operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryWith exn handler =
        catch exn
        |> bind (function Ok value -> result value | Exception exn -> handler exn)

    // The whileLoop operator.
    // This is boilerplate in terms of "result" and "bind".
    let whileLoop expr condition =
        let rec loop () =
            match expr with
            | Done _ -> result (Ok ())
            | NotYetDone work ->
                NotYetDone (fun () ->
                    loop ()
                    |> bind (fun () -> condition))
        loop ()
        result (Ok ())
```

```

let rec whileLoop pred body =
    if pred() then body |> bind (fun _ -> whileLoop pred body)
    else result ()

// The sequential composition operator.
// This is boilerplate in terms of "result" and "bind".
let combine expr1 expr2 =
    expr1 |> bind (fun () -> expr2)

// The using operator.
let using (resource: #System.IDisposable) func =
    tryFinally (func resource) (fun () -> resource.Dispose())

// The forLoop operator.
// This is boilerplate in terms of "catch", "result", and "bind".
let forLoop (collection:seq<_>) func =
    let ie = collection.GetEnumerator()
    tryFinally
        (whileLoop
            (fun () -> ie.MoveNext())
            (delay (fun () -> let value = ie.Current in func value)))
    (fun () -> ie.Dispose())

// The builder class.
type EventuallyBuilder() =
    member x.Bind(comp, func) = Eventually.bind func comp
    member x.Return(value) = Eventually.result value
    member x.ReturnFrom(value) = value
    member x.Combine(expr1, expr2) = Eventually.combine expr1 expr2
    member x.Delay(func) = Eventually.delay func
    member x.Zero() = Eventually.result ()
    member x.TryWith(expr, handler) = Eventually.tryWith expr handler
    member x.TryFinally(expr, compensation) = Eventually.tryFinally expr compensation
    member x.For(coll:seq<_>, func) = Eventually.forLoop coll func
    member x.Using(resource, expr) = Eventually.using resource expr

let eventually = new EventuallyBuilder()

let comp = eventually {
    for x in 1..2 do
        printfn " x = %d" x
    return 3 + 4 }

// Try the remaining lines in F# interactive to see how this
// computation expression works in practice.
let step x = Eventually.step x

// returns "NotYetDone <closure>"
comp |> step

// prints "x = 1"
// returns "NotYetDone <closure>"
comp |> step |> step

// prints "x = 1"
// prints "x = 2"
// returns "Done 7"
comp |> step |> step |> step |> step

```

Una expresión de cálculo tiene un tipo subyacente, que devuelve la expresión. El tipo subyacente puede representar un resultado calculado o un cálculo retrasado que se puede realizar, o puede proporcionar una manera de recorrer en iteración algún tipo de colección. En el ejemplo anterior, el tipo subyacente era **Eventually**. Para una expresión de secuencia, [System.Collections.Generic.IEnumerable<T>](#) el tipo subyacente es [T](#). Para una expresión de consulta, [System.Linq.IQueryable](#) el tipo subyacente es [T](#). Para un flujo de trabajo [Async](#) asíncronico, el tipo subyacente es [Task](#). El [Async](#) objeto representa el trabajo que se va a realizar para calcular el resultado. Por ejemplo, [Async.RunSynchronously](#) se llama para ejecutar un cálculo y devolver el



resultado.

## Operaciones personalizadas

Puede definir una operación personalizada en una expresión de cálculo y utilizar una operación personalizada como operador en una expresión de cálculo. Por ejemplo, puede incluir un operador de consulta en una expresión de consulta. Al definir una operación personalizada, debe definir los métodos `Yield` y `For` en la expresión de cálculo. Para definir una operación personalizada, colóquela en una `CustomOperationAttribute` clase de generador para la expresión de cálculo y, a continuación, aplique el archivo . Este atributo toma una cadena como argumento, que es el nombre que se usará en una operación personalizada. Este nombre entra en el ámbito al principio de la llave de apertura de la expresión de cálculo. Por lo tanto, no debe usar identificadores que tengan el mismo nombre que una operación personalizada en este bloque. Por ejemplo, evite el uso `all` de `last` identificadores como `o` en expresiones de consulta.

### Ampliación de los constructores existentes con nuevas operaciones personalizadas

Si ya tiene una clase de generador, sus operaciones personalizadas se pueden extender desde fuera de esta clase de generador. Las extensiones deben declararse en módulos. Los espacios de nombres no pueden contener miembros de extensión excepto en el mismo archivo y el mismo grupo de declaración de espacio de nombres donde se define el tipo.

En el ejemplo siguiente se `Microsoft.FSharp.Linq.QueryBuilder` muestra la extensión de la clase existente.

```
type Microsoft.FSharp.Linq.QueryBuilder with

    [
```

## Consulte también

- [Referencia del lenguaje f](#)
- [Flujos de trabajo asincrónicos](#)
- [Secuencias](#)
- [Expresiones de consulta](#)

# Flujos de trabajo asincrónicos

23/10/2019 • 9 minutes to read • [Edit Online](#)

## NOTE

El vínculo de la referencia de API le llevará a MSDN. La referencia de API de docs.microsoft.com no está completa.

En este tema se describe F# la compatibilidad de para realizar cálculos de forma asincrónica, es decir, sin bloquear la ejecución de otro trabajo. Por ejemplo, los cálculos asincrónicos se pueden usar para escribir aplicaciones que tengan interfaces de usuario que sigan respondiendo a los usuarios a medida que la aplicación realice otro trabajo.

## Sintaxis

```
async { expression }
```

## Comentarios

En la sintaxis anterior, el cálculo representado por `expression` está configurado para ejecutarse de forma asincrónica, es decir, sin bloquear el subproceso de cálculo actual cuando se realizan operaciones asincrónicas de suspensión, e/s y otras operaciones asincrónicas. A menudo, los cálculos asincrónicos se inician en un subproceso en segundo plano mientras la ejecución continúa en el subproceso actual. El tipo de la expresión es `Async<'T>`, donde `'T` es el tipo devuelto por la expresión cuando `return` se usa la palabra clave. El código en una expresión de este tipo se conoce como un *bloque asincrónico* o un bloque *asincrónico*.

Hay varias maneras de programar de forma asincrónica y la `Async` clase proporciona métodos que admiten varios escenarios. El enfoque general es crear `Async` objetos que representan el cálculo o los cálculos que desea ejecutar de forma asincrónica y, a continuación, iniciar estos cálculos mediante una de las funciones de desencadenamiento. Las distintas funciones de desencadenador proporcionan diferentes maneras de ejecutar cálculos asincrónicos y la que se usa depende de si se desea usar el subproceso actual, un subproceso en segundo plano o un objeto de tarea .NET Framework, y si hay continuación. funciones que se deben ejecutar cuando finaliza el cálculo. Por ejemplo, para iniciar un cálculo asincrónico en el subproceso actual, puede usar `Async.StartImmediate`. Cuando se inicia un cálculo asincrónico desde el subproceso de la interfaz de usuario, no se bloquea el bucle de eventos principal que procesa las acciones del usuario, como pulsaciones de teclas y actividad del mouse, para que la aplicación siga respondiendo.

## Enlace asincrónico mediante Let!

En un flujo de trabajo asincrónico, algunas expresiones y operaciones son sincrónicas y otras son cálculos más largos que están diseñados para devolver un resultado de forma asincrónica. Cuando se llama a un método de forma asincrónica, en lugar `let` de un enlace normal `let!`, se usa. El efecto de `let!` es permitir que la ejecución continúe en otros cálculos o subprocesos mientras se realiza el cálculo. Una vez que el lado derecho `let!` del enlace vuelve, el resto del flujo de trabajo asincrónico reanuda la ejecución.

En el código siguiente se muestra la `let` diferencia `let!` entre y. La línea de código que usa `let` simplemente crea un cálculo asincrónico como un objeto que se puede ejecutar más adelante mediante, por ejemplo, `Async.StartImmediate` o `Async.RunSynchronously`. La línea de código que usa `let!` inicia el cálculo y, a continuación, el subproceso se suspende hasta que el resultado esté disponible, momento en el que

continúa la ejecución.

```
// let just stores the result as an asynchronous operation.  
let (result1 : Async<byte[]>) = stream.AsyncRead(bufferSize)  
// let! completes the asynchronous operation and returns the data.  
let! (result2 : byte[]) = stream.AsyncRead(bufferSize)
```

Además `let!` de, puede usar `use!` para realizar enlaces asíncronos. La diferencia entre `let!` y `use!` es la misma que la diferencia entre `let` y `use`. En `use!` el caso de, el objeto se desecha al cerrar el ámbito actual. Tenga en cuenta que en la versión actual F# del lenguaje `use!`, no permite que un valor se inicialice en `use` null, aunque sí lo hace.

## Primitivas asíncronas

Un método que realiza una única tarea asíncrona y devuelve el resultado se denomina *primitiva asíncrona*, y se ha diseñado específicamente para su uso con `let!`. Se definen varias primitivas asíncronas en la F# biblioteca principal. En el módulo `Microsoft.FSharp.Control.WebExtensions` se definen dos métodos de este tipo para las aplicaciones Web: `WebClient.AsyncDownloadString` y `WebRequest.AsyncGetResponse`. Ambos primitivos descargan datos de una página web, dada una dirección URL. `AsyncGetResponse` genera un `System.Net.WebResponse` objeto y `AsyncDownloadString` genera una cadena que representa el código HTML de una página web.

En el `Microsoft.FSharp.Control.CommonExtensions` módulo se incluyen varios primitivos para las operaciones de e/s asíncronas. Estos métodos de extensión de `System.IO.Stream` la clase `Stream.AsyncRead` son `Stream.AsyncWrite` y.

También puede escribir sus propias primitivas asíncronas definiendo una función cuyo cuerpo completo esté incluido en un bloque asíncrono.

Para usar métodos asíncronos en el .NET Framework que están diseñados para otros modelos asíncronos con el F# modelo de programación asíncrona, cree una función que devuelva un F# `Async` objeto. La F# biblioteca tiene funciones que facilitan esta tarea.

Aquí se incluye un ejemplo del uso de flujos de trabajo asíncronos. Hay muchas otras en la documentación de los métodos de la [clase asíncrona](#).

En este ejemplo se muestra cómo usar flujos de trabajo asíncronos para realizar cálculos en paralelo.

En el ejemplo de código siguiente, una `fetchAsync` función obtiene el texto HTML devuelto de una solicitud Web. La `fetchAsync` función contiene un bloque de código asíncrono. Cuando se realiza un enlace al resultado de una primitiva asíncrona, en este caso `AsyncDownloadString`, `Let` se usa en lugar de `Let`.

Utilice la función `Async.RunSynchronously` para ejecutar una operación asíncrona y esperar su resultado. Por ejemplo, puede ejecutar varias operaciones asíncronas en paralelo mediante la `Async.Parallel` función junto con la `Async.RunSynchronously` función. La `Async.Parallel` función toma una lista de los `Async` objetos, configura el código para que cada `Async` objeto de tarea se ejecute en paralelo y devuelve un `Async` objeto que representa el cálculo en paralelo. Al igual que en el caso de una sola `Async.RunSynchronously` operación, se llama a para iniciar la ejecución.

La `runAll` función inicia tres flujos de trabajo asíncronos en paralelo y espera hasta que se hayan completado.

```

open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]

let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }

let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()

```

## Vea también

- [Referencia del lenguaje F#](#)
- [Expresiones de cálculo](#)
- [Control.Async \(clase\)](#)

# Expresiones de consulta

20/05/2020 • 49 minutes to read • [Edit Online](#)

## NOTE

Los vínculos de la referencia de API de este artículo le llevarán a MSDN. La referencia de API de docs.microsoft.com no está completa.

Las expresiones de consulta permiten consultar un origen de datos y colocar los datos en un formato deseado. Las expresiones de consulta proporcionan compatibilidad con LINQ en F #.

## Sintaxis

```
query { expression }
```

## Observaciones

Las expresiones de consulta son un tipo de expresión de cálculo similar a las expresiones de secuencia. Del mismo modo que se especifica una secuencia proporcionando código en una expresión de secuencia, se especifica un conjunto de datos proporcionando código en una expresión de consulta. En una expresión de secuencia, la `yield` palabra clave identifica los datos que se van a devolver como parte de la secuencia resultante. En las expresiones de consulta, la `select` palabra clave realiza la misma función. Además de la `select` palabra clave, F # también admite un número de operadores de consulta que son muy similares a las partes de una instrucción SELECT de SQL. Este es un ejemplo de una expresión de consulta simple, junto con el código que se conecta al origen de OData de Northwind.

```
// Use the OData type provider to create types that can be used to access the Northwind database.
// Add References to FSharp.Data.TypeProviders and System.Data.Services.Client
open Microsoft.FSharp.Data.TypeProviders

type Northwind = ODataService<"http://services.odata.org/Northwind/Northwind.svc">
let db = Northwind.GetDataContext()

// A query expression.
let query1 =
    query {
        for customer in db.Customers do
            select customer
    }

// Print results
query1
|> Seq.iter (fun customer -> printfn "Company: %s Contact: %s" customer.CompanyName customer.ContactName)
```

En el ejemplo de código anterior, la expresión de consulta está entre llaves. El significado del código en la expresión es, devuelva todos los clientes de la tabla Customers de la base de datos en los resultados de la consulta. Las expresiones de consulta devuelven un tipo que implementa `IQueryable<T>` y `IEnumerable<T>`, y, por tanto, se pueden iterar mediante el [módulo SEQ](#) como se muestra en el ejemplo.

Cada tipo de expresión de cálculo se genera a partir de una clase de generador. La clase de generador para la expresión de cálculo de consulta es `QueryBuilder`. Para obtener más información, vea [expresiones de cálculo](#) y

## Operadores de consulta

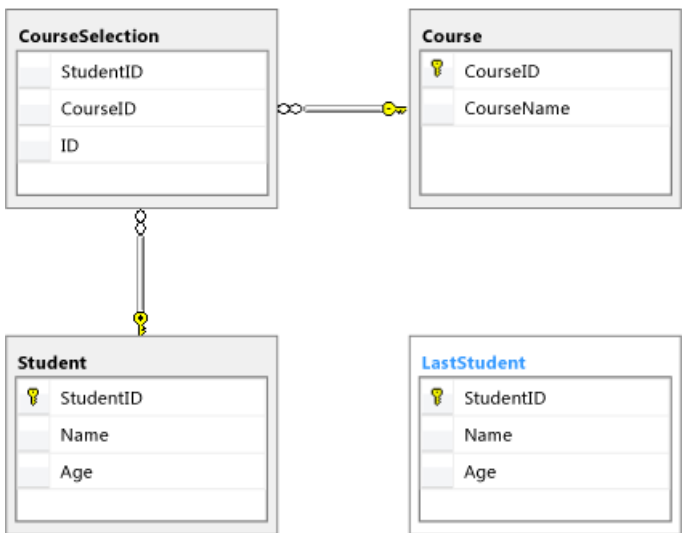
Los operadores de consulta permiten especificar los detalles de la consulta, como incluir criterios en los registros que se van a devolver o especificar el criterio de ordenación de los resultados. El origen de la consulta debe admitir el operador de consulta. Si intenta utilizar un operador de consulta no admitido, se

`System.NotSupportedException` producirá una excepción.

En las expresiones de consulta solo se permiten las expresiones que se pueden traducir a SQL. Por ejemplo, no se permiten llamadas de función en las expresiones cuando se usa el `where` operador de consulta.

En la tabla 1 se muestran los operadores de consulta disponibles. Además, vea Tabla2, que compara las consultas SQL y las expresiones de consulta de F # equivalentes más adelante en este tema. Algunos operadores de consulta no son compatibles con algunos proveedores de tipos. En concreto, el proveedor de tipo de OData está limitado en los operadores de consulta que admite debido a las limitaciones de OData. Para obtener más información, vea [ODataService Type Provider \(F #\)](#).

En esta tabla se presupone una base de datos de la siguiente forma:



El código de las tablas siguientes también asume el siguiente código de conexión de base de datos. Los proyectos deben agregar referencias a los ensamblados System. Data, System. Data. Linq y FSharp. Data. TypeProviders. Al final de este tema se incluye el código que crea esta base de datos.

```
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq
open Microsoft.FSharp.Linq

type schema = SqlConnection< @"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated Security=SSPI;" >

let db = schema.GetDataContext()

// Needed for some query operator examples:
let data = [ 1; 5; 7; 11; 18; 21]
```

Tabla 1. Operadores de consulta

OPERADOR	DESCRIPCIÓN
----------	-------------

<div>contains</div>	<p>Determina si los elementos seleccionados incluyen un elemento especificado.</p> <pre> query {   for student in db.Student do     select student.Age.Value     contains 11 } </pre>
<div>count</div>	<p>Devuelve el número de elementos seleccionados.</p> <pre> query {   for student in db.Student do     select student     count } </pre>
<div>last</div>	<p>Selecciona el último elemento de los seleccionados hasta el momento.</p> <pre> query {   for number in data do     last } </pre>
<div>lastOrDefault</div>	<p>Selecciona el último elemento de los seleccionados hasta el momento, o un valor predeterminado si no se encuentra ningún elemento.</p> <pre> query {   for number in data do     where (number &lt; 0)     lastOrDefault } </pre>
<div>exactlyOne</div>	<p>Selecciona el elemento específico único seleccionado hasta el momento. Si hay varios elementos presentes, se produce una excepción.</p> <pre> query {   for student in db.Student do     where (student.StudentID = 1)     select student     exactlyOne } </pre>

<div>exactlyOneOrDefault</div>	<p>Selecciona el elemento único específico de los seleccionados hasta el momento, o un valor predeterminado si no se encuentra dicho elemento.</p> <pre> query {   for student in db.Student do     where (student.StudentID = 1)     select student     exactlyOneOrDefault }</pre>
<div>headOrDefault</div>	<p>Selecciona el primer elemento de los seleccionados hasta el momento, o un valor predeterminado si la secuencia no contiene elementos.</p> <pre> query {   for student in db.Student do     select student     headOrDefault }</pre>
<div>select</div>	<p>Proyecta cada uno de los elementos seleccionados hasta el momento.</p> <pre> query {   for student in db.Student do     select student }</pre>
<div>where</div>	<p>Selecciona elementos basándose en un predicado especificado.</p> <pre> query {   for student in db.Student do     where (student.StudentID &gt; 4)     select student }</pre>
<div>minBy</div>	<p>Selecciona un valor para cada elemento seleccionado hasta el momento y devuelve el valor mínimo resultante.</p> <pre> query {   for student in db.Student do     minBy student.StudentID }</pre>



<div>maxBy</div>	<p>Selecciona un valor para cada elemento seleccionado hasta el momento y devuelve el valor máximo resultante.</p> <pre>query {   for student in db.Student do     maxBy student.StudentID }</pre>
<div>groupBy</div>	<p>Agrupar los elementos seleccionados hasta el momento según un selector de claves especificado.</p> <pre>query {   for student in db.Student do     groupBy student.Age into g     select (g.Key, g.Count()) }</pre>
<div>sortBy</div>	<p>Ordena los elementos seleccionados hasta el momento en orden ascendente por la clave de ordenación especificada.</p> <pre>query {   for student in db.Student do     sortBy student.Name     select student }</pre>
<div>sortByDescending</div>	<p>Ordena los elementos seleccionados hasta el momento en orden descendente por la clave de ordenación especificada.</p> <pre>query {   for student in db.Student do     sortByDescending student.Name     select student }</pre>
<div>thenBy</div>	<p>Realiza una clasificación posterior de los elementos seleccionados hasta el momento en orden ascendente por la clave de ordenación especificada. Este operador solo se puede usar después de <code>sortBy</code>, <code>sortByDescending</code>, <code>thenBy</code> o <code>thenByDescending</code>.</p> <pre>query {   for student in db.Student do     where student.Age.HasValue     sortBy student.Age.Value     thenBy student.Name     select student }</pre>

thenByDescending

Realiza una clasificación posterior de los elementos seleccionados hasta el momento en orden descendente por la clave de ordenación especificada. Este operador solo se puede usar después de `sortBy` , `sortByDescending` , `thenBy` o `thenByDescending` .

```
query {
  for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenByDescending student.Name
    select student
}
```

groupValBy

Selecciona un valor para cada elemento seleccionado hasta el momento y agrupa los elementos por la clave especificada.

```
query {
  for student in db.Student do
    groupValBy student.Name student.Age into g
    select (g, g.Key, g.Count())
}
```

join

Correlaciona dos conjuntos de valores seleccionados en función de las claves coincidentes. Tenga en cuenta que el orden de las claves alrededor del signo = en una expresión de combinación es significativo. En todas las combinaciones, si la línea se divide después del `->` símbolo, se debe aplicar sangría a la sangría al menos tan lejos como la palabra clave `for` .

```
query {
  for student in db.Student do
    join selection in db.CourseSelection
      on (student.StudentID =
selection.StudentID)
    select (student, selection)
}
```

<div>groupJoin</div>	<p>Correlaciona dos conjuntos de valores seleccionados en función de las claves coincidentes y agrupa los resultados. Tenga en cuenta que el orden de las claves alrededor del signo = en una expresión de combinación es significativo.</p> <pre> query {   for student in db.Student do     groupJoin courseSelection in       db.CourseSelection       on (student.StudentID =         courseSelection.StudentID) into g     for courseSelection in g do       join course in db.Course         on (courseSelection.CourseID =           course.CourseID)       select (student.Name, course.CourseName)     } </pre>
<div>leftOuterJoin</div>	<p>Correlaciona dos conjuntos de valores seleccionados en función de las claves coincidentes y agrupa los resultados. Si algún grupo está vacío, se utiliza en su lugar un grupo con un único valor predeterminado. Tenga en cuenta que el orden de las claves alrededor del signo = en una expresión de combinación es significativo.</p> <pre> query {   for student in db.Student do     leftOuterJoin selection in       db.CourseSelection       on (student.StudentID =         selection.StudentID) into result     for selection in result.DefaultIfEmpty() do       select (student, selection)     } </pre>
<div>sumByNullable</div>	<p>Selecciona un valor que acepta valores NULL para cada elemento seleccionado hasta el momento y devuelve la suma de estos valores. Si alguno de los valores NULL no tiene un valor, se omite.</p> <pre> query {   for student in db.Student do     sumByNullable student.Age   } </pre>
<div>minByNullable</div>	<p>Selecciona un valor que acepta valores NULL para cada elemento seleccionado hasta el momento y devuelve el mínimo de estos valores. Si alguno de los valores NULL no tiene un valor, se omite.</p> <pre> query {   for student in db.Student do     minByNullable student.Age   } </pre>

<code>maxByNullable</code>	<p>Selecciona un valor que acepta valores NULL para cada elemento seleccionado hasta el momento y devuelve el máximo de estos valores. Si alguno de los valores NULL no tiene un valor, se omite.</p> <pre>query {   for student in db.Student do     maxByNullable student.Age }</pre>
<code>averageByNullable</code>	<p>Selecciona un valor que acepta valores NULL para cada elemento seleccionado hasta el momento y devuelve el promedio de estos valores. Si alguno de los valores NULL no tiene un valor, se omite.</p> <pre>query {   for student in db.Student do     averageByNullable (Nullable.float student.Age) }</pre>
<code>averageBy</code>	<p>Selecciona un valor para cada elemento seleccionado hasta el momento y devuelve el promedio de estos valores.</p> <pre>query {   for student in db.Student do     averageBy (float student.StudentID) }</pre>
<code>distinct</code>	<p>Selecciona distintos elementos de los elementos seleccionados hasta el momento.</p> <pre>query {   for student in db.Student do     join selection in db.CourseSelection       on (student.StudentID = selection.StudentID)     distinct }</pre>

exists	<p>Determina si cualquier elemento seleccionado hasta el momento satisface una condición.</p> <pre> query {   for student in db.Student do     where       (query {         for courseSelection in           db.CourseSelection do             exists (courseSelection.StudentID =               student.StudentID) })       select student     }   } </pre>
find	<p>Selecciona el primer elemento seleccionado hasta el momento que satisface una condición especificada.</p> <pre> query {   for student in db.Student do     find (student.Name = "Abercrombie, Kim")   } </pre>
all	<p>Determina si todos los elementos seleccionados hasta el momento satisfacen una condición.</p> <pre> query {   for student in db.Student do     all (SqlMethods.Like(student.Name, "%,%"))   } </pre>
head	<p>Selecciona el primer elemento de los seleccionados hasta el momento.</p> <pre> query {   for student in db.Student do     head   } </pre>
nth	<p>Selecciona el elemento en un índice especificado entre los seleccionados hasta el momento.</p> <pre> query {   for numbers in data do     nth 3   } </pre>

<div>skip</div>	<p>Omite un número especificado de los elementos seleccionados hasta el momento y, a continuación, selecciona los elementos restantes.</p> <pre> query {   for student in db.Student do     skip 1 } </pre>
<div>skipWhile</div>	<p>Omite los elementos de una secuencia siempre que el valor de una condición especificada sea true y, a continuación, seleccione los elementos restantes.</p> <pre> query {   for number in data do     skipWhile (number &lt; 3)     select student } </pre>
<div>sumBy</div>	<p>Selecciona un valor para cada elemento seleccionado hasta el momento y devuelve la suma de estos valores.</p> <pre> query {   for student in db.Student do     sumBy student.StudentID } </pre>
<div>take</div>	<p>Selecciona un número especificado de elementos contiguos de los seleccionados hasta el momento.</p> <pre> query {   for student in db.Student do     select student     take 2 } </pre>
<div>takeWhile</div>	<p>Selecciona elementos de una secuencia siempre que una condición especificada sea true y, a continuación, omite los elementos restantes.</p> <pre> query {   for number in data do     takeWhile (number &lt; 10) } </pre>

<div>sortByNullable</div>	<p>Ordena los elementos seleccionados hasta el momento en orden ascendente por la clave de ordenación especificada que acepta valores NULL.</p> <pre> query {   for student in db.Student do     sortByNullable student.Age   select student }</pre>
<div>sortByNullableDescending</div>	<p>Ordena los elementos seleccionados hasta el momento en orden descendente por la clave de ordenación especificada que acepta valores NULL.</p> <pre> query {   for student in db.Student do     sortByNullableDescending student.Age   select student }</pre>
<div>thenByNullable</div>	<p>Realiza una clasificación posterior de los elementos seleccionados hasta el momento en orden ascendente por la clave de ordenación especificada que acepta valores NULL. Este operador solo se puede usar inmediatamente después de <code>sortBy</code>, <code>sortByDescending</code>, <code>thenBy</code> o <code>thenByDescending</code>, o sus variantes que aceptan valores NULL.</p> <pre> query {   for student in db.Student do     sortBy student.Name     thenByNullable student.Age   select student }</pre>
<div>thenByNullableDescending</div>	<p>Realiza una clasificación posterior de los elementos seleccionados hasta el momento en orden descendente por la clave de ordenación especificada que acepta valores NULL. Este operador solo se puede usar inmediatamente después de <code>sortBy</code>, <code>sortByDescending</code>, <code>thenBy</code> o <code>thenByDescending</code>, o sus variantes que aceptan valores NULL.</p> <pre> query {   for student in db.Student do     sortBy student.Name     thenByNullableDescending student.Age   select student }</pre>

## Comparación de Transact-SQL y las expresiones de consulta de F#

En la tabla siguiente se muestran algunas consultas de Transact-SQL comunes y sus equivalentes en F#. El código

de esta tabla también asume la misma base de datos que la tabla anterior y el mismo código inicial para configurar el proveedor de tipo.

**Tabla 2. Transact-SQL y expresiones de consulta de F#**

TRANSACT-SQL (NO DISTINGUE MAYÚSCULAS DE MINÚSCULAS)	EXPRESIÓN DE CONSULTA DE F# (DISTINGUE MAYÚSCULAS DE MINÚSCULAS)
<p>Seleccionar todos los campos de la tabla.</p> <pre>SELECT * FROM Student</pre>	<pre>// All students. query {     for student in db.Student do         select student }</pre>
<p>Recuento de registros en una tabla.</p> <pre>SELECT COUNT( * ) FROM Student</pre>	<pre>// Count of students. query {     for student in db.Student do         count }</pre>
<p><b>EXISTS</b></p> <pre>SELECT * FROM Student WHERE EXISTS     (SELECT * FROM CourseSelection      WHERE CourseSelection.StudentID =        Student.StudentID)</pre>	<pre>// Find students who have signed up at least one course. query {     for student in db.Student do         where             (query {                 for courseSelection in db.CourseSelection do                     exists (courseSelection.StudentID = student.StudentID) })             select student }</pre>
<p>Agrupar</p> <pre>SELECT Student.Age, COUNT( * ) FROM Student GROUP BY Student.Age</pre>	<pre>// Group by age and count. query {     for n in db.Student do         groupBy n.Age into g         select (g.Key, g.Count()) } // OR query {     for n in db.Student do         groupValBy n.Age n.Age into g         select (g.Key, g.Count()) }</pre>
<p>Agrupación con condición.</p> <pre>SELECT Student.Age, COUNT( * ) FROM Student GROUP BY Student.Age HAVING student.Age &gt; 10</pre>	<pre>// Group students by age where age &gt; 10. query {     for student in db.Student do         groupBy student.Age into g         where (g.Key.HasValue &amp;&amp; g.Key.Value &gt; 10)         select (g.Key, g.Count()) }</pre>



### Agrupación con condición de recuento.

```
SELECT Student.Age, COUNT( * )
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
```

```
// Group students by age and count number of
students
// at each age with more than 1 student.
query {
  for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
```

### Agrupación, recuento y suma.

```
SELECT Student.Age, COUNT( * ),
SUM(Student.Age) as total
FROM Student
GROUP BY Student.Age
```

```
// Group students by age and sum ages.
query {
  for student in db.Student do
    groupBy student.Age into g
    let total =
      query {
        for student in g do
          sumByNullable student.Age
      }
    select (g.Key, g.Count(), total)
}
```

### Agrupar, contar y ordenar por recuento.

```
SELECT Student.Age, COUNT( * ) as myCount
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
ORDER BY COUNT( * ) DESC
```

```
// Group students by age, count number of
students
// at each age, and display all with count > 1
// in descending order of count.
query {
  for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
    select (g.Key, g.Count())
}
```

### IN un conjunto de valores especificados.

```
SELECT *
FROM Student
WHERE Student.StudentID IN (1, 2, 5, 10)
```

```
// Select students where studentID is one of a
given list.
let idQuery =
  query {
    for id in [1; 2; 5; 10] do
      select id
  }
query {
  for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
}
```

LIKE y TOP .

```
-- '_e%' matches strings where the second
character is 'e'
SELECT TOP 2 * FROM Student
WHERE Student.Name LIKE '_e%'
```

```
// Look for students with Name match _e% pattern
and take first two.
query {
  for student in db.Student do
    where (SqlMethods.Like( student.Name, "_e%"))
  )
  select student
  take 2
}
```

LIKE con el conjunto de coincidencia de patrones.

```
-- '[abc]%' matches strings where the first
character is
-- 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[abc]%'
```

```
query {
  for student in db.Student do
    where (SqlMethods.Like( student.Name, "[abc]%" )
  select student
}
```

LIKE con el patrón de exclusión set.

```
-- '[^abc]%' matches strings where the first
character is
-- not 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
// Look for students with name matching [^abc]%%
pattern.
query {
  for student in db.Student do
    where (SqlMethods.Like( student.Name, "[^abc]%" )
  select student
}
```

LIKE en un campo, pero seleccione otro campo.

```
SELECT StudentID AS ID FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
query {
  for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%" )
  select n.StudentID
}
```

LIKE , con búsqueda de subcadenas.

```
SELECT * FROM Student
WHERE Student.Name like '%A%'
```

```
// Using Contains as a query filter.
query {
  for student in db.Student do
    where (student.Name.Contains("a"))
  select student
}
```

Simple JOIN con dos tablas.

```
SELECT * FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join Student and CourseSelection tables.
query {
  for student in db.Student do
    join selection in db.CourseSelection
    on (student.StudentID =
selection.StudentID)
  select (student, selection)
}
```

LEFT JOIN con dos tablas.

```
SELECT * FROM Student
LEFT JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
//Left Join Student and CourseSelection tables.
query {
  for student in db.Student do
    leftOuterJoin selection in
      db.CourseSelection
        on (student.StudentID =
            selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
      select (student, selection)
}
```

JOIN con COUNT

```
SELECT COUNT( * ) FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join with count.
query {
  for n in db.Student do
    join e in db.CourseSelection
      on (n.StudentID = e.StudentID)
    count
}
```

DISTINCT

```
SELECT DISTINCT StudentID FROM CourseSelection
```

```
// Join with distinct.
query {
  for student in db.Student do
    join selection in db.CourseSelection
      on (student.StudentID =
          selection.StudentID)
    distinct
}
```

Recuento distintivo.

```
SELECT DISTINCT COUNT(StudentID) FROM
CourseSelection
```

```
// Join with distinct and count.
query {
  for n in db.Student do
    join e in db.CourseSelection
      on (n.StudentID = e.StudentID)
    distinct
    count
}
```

BETWEEN

```
SELECT * FROM Student
WHERE Student.Age BETWEEN 10 AND 15
```

```
// Selecting students with ages between 10 and
15.
query {
  for student in db.Student do
    where (student.Age ?>= 10 && student.Age ?<
15)
    select student
}
```

OR

```
SELECT * FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
```

```
// Selecting students with age that's either 11
or 12.
query {
  for student in db.Student do
    where (student.Age.Value = 11 ||
student.Age.Value = 12)
    select student
}
```

OR con ordenación

```
SELECT * FROM Student
WHERE Student.Age = 12 OR Student.Age = 13
ORDER BY Student.Age DESC
```

```
// Selecting students in a certain age range and
sorting.
query {
  for n in db.Student do
    where (n.Age.Value = 12 || n.Age.Value = 13)
    sortByNullableDescending n.Age
    select n
}
```

TOP, OR y ordenación.

```
SELECT TOP 2 student.Name FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
ORDER BY Student.Name DESC
```

```
// Selecting students with certain ages,
// taking account of the possibility of nulls.
query {
  for student in db.Student do
    where
      ((student.Age.HasValue &&
student.Age.Value = 11) ||
      (student.Age.HasValue &&
student.Age.Value = 12))
    sortByDescending student.Name
    select student.Name
    take 2
}
```

UNION de dos consultas.

```
SELECT * FROM Student
UNION
SELECT * FROM lastStudent
```

```
let query1 =
  query {
    for n in db.Student do
      select (n.Name, n.Age)
  }

let query2 =
  query {
    for n in db.LastStudent do
      select (n.Name, n.Age)
  }

query2.Union (query1)
```

Intersección de dos consultas.

```
SELECT * FROM Student
INTERSECT
SELECT * FROM LastStudent
```

```
let query1 =
    query {
        for n in db.Student do
            select (n.Name, n.Age)
        }

let query2 =
    query {
        for n in db.LastStudent do
            select (n.Name, n.Age)
        }

query1.Intersect(query2)
```

CASE cumple.

```
SELECT student.StudentID,
CASE Student.Age
    WHEN -1 THEN 100
    ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special value.
query {
    for student in db.Student do
        select
            (if student.Age.HasValue &&
student.Age.Value = -1 then
                (student.StudentID,
System.Nullable<int>(100), student.Age)
            else (student.StudentID, student.Age,
student.Age))
        }
}
```

Varios casos.

```
SELECT Student.StudentID,
CASE Student.Age
    WHEN -1 THEN 100
    WHEN 0 THEN 1000
    ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special values.
query {
    for student in db.Student do
        select
            (if student.Age.HasValue &&
student.Age.Value = -1 then
                (student.StudentID,
System.Nullable<int>(100), student.Age)
            elif student.Age.HasValue &&
student.Age.Value = 0 then
                (student.StudentID,
System.Nullable<int>(1000), student.Age)
            else (student.StudentID, student.Age,
student.Age))
        }
}
```

Varias tablas.

```
SELECT * FROM Student, Course
```

```
// Multiple table select.
query {
    for student in db.Student do
        for course in db.Course do
            select (student, course)
        }
}
```

Varias combinaciones.

```
SELECT Student.Name, Course.CourseName
FROM Student
JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Multiple joins.
query {
  for student in db.Student do
    join courseSelection in db.CourseSelection
      on (student.StudentID =
courseSelection.StudentID)
    join course in db.Course
      on (courseSelection.CourseID =
course.CourseID)
    select (student.Name, course.CourseName)
}
```

Múltiples combinaciones externas izquierdas.

```
SELECT Student.Name, Course.CourseName
FROM Student
LEFT OUTER JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
LEFT OUTER JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Using leftOuterJoin with multiple joins.
query {
  for student in db.Student do
    leftOuterJoin courseSelection in
db.CourseSelection
      on (student.StudentID =
courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty()
do
      leftOuterJoin course in db.Course
        on (courseSelection.CourseID =
course.CourseID) into g2
      for course in g2.DefaultIfEmpty() do
        select (student.Name, course.CourseName)
}
```

El siguiente código se puede utilizar para crear la base de datos de ejemplo para estos ejemplos.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

USE [master];
GO

IF EXISTS (SELECT * FROM sys.databases WHERE name = 'MyDatabase')
DROP DATABASE MyDatabase;
GO

-- Create the MyDatabase database.
CREATE DATABASE MyDatabase COLLATE SQL_Latin1_General_CP1_CI_AS;
GO

-- Specify a simple recovery model
-- to keep the log growth to a minimum.
ALTER DATABASE MyDatabase
SET RECOVERY SIMPLE;
GO

USE MyDatabase;
GO

CREATE TABLE [dbo].[Course] (
  [CourseID] INT NOT NULL,
  [CourseName] NVARCHAR (50) NOT NULL,
  PRIMARY KEY CLUSTERED ([CourseID] ASC)
);

CREATE TABLE [dbo].[Student] (
  [StudentID] INT NOT NULL,
```

```

[Name]          NVARCHAR (50) NOT NULL,
[Age]           INT             NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

```

```

CREATE TABLE [dbo].[CourseSelection] (
[ID]            INT NOT NULL,
[StudentID] INT NOT NULL,
[CourseID] INT NOT NULL,
PRIMARY KEY CLUSTERED ([ID] ASC),
CONSTRAINT [FK_CourseSelection_ToTable] FOREIGN KEY ([StudentID]) REFERENCES [dbo].[Student] ([StudentID]) ON
DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT [FK_CourseSelection_Course_1] FOREIGN KEY ([CourseID]) REFERENCES [dbo].[Course] ([CourseID]) ON
DELETE NO ACTION ON UPDATE NO ACTION
);

```

```

CREATE TABLE [dbo].[LastStudent] (
[StudentID] INT             NOT NULL,
[Name]       NVARCHAR (50) NOT NULL,
[Age]        INT             NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

```

```
-- Insert data into the tables.
```

```

USE MyDatabase
INSERT INTO Course (CourseID, CourseName)
VALUES(1, 'Algebra I');
INSERT INTO Course (CourseID, CourseName)
VALUES(2, 'Trigonometry');
INSERT INTO Course (CourseID, CourseName)
VALUES(3, 'Algebra II');
INSERT INTO Course (CourseID, CourseName)
VALUES(4, 'History');
INSERT INTO Course (CourseID, CourseName)
VALUES(5, 'English');
INSERT INTO Course (CourseID, CourseName)
VALUES(6, 'French');
INSERT INTO Course (CourseID, CourseName)
VALUES(7, 'Chinese');

```

```

INSERT INTO Student (StudentID, Name, Age)
VALUES(1, 'Abercrombie, Kim', 10);
INSERT INTO Student (StudentID, Name, Age)
VALUES(2, 'Abolrous, Hazen', 14);
INSERT INTO Student (StudentID, Name, Age)
VALUES(3, 'Hance, Jim', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(4, 'Adams, Terry', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(5, 'Hansen, Claus', 11);
INSERT INTO Student (StudentID, Name, Age)
VALUES(6, 'Penor, Lori', 13);
INSERT INTO Student (StudentID, Name, Age)
VALUES(7, 'Perham, Tom', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(8, 'Peng, Yun-Feng', NULL);

```

```

INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(1, 1, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(2, 1, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(3, 1, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(4, 2, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(5, 2, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(6, 2, 6);

```

```
VALUES(0, 2, 0);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(7, 2, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(8, 3, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(9, 3, 1);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(10, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(11, 4, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(12, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(13, 5, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(14, 5, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(15, 7, 3);
```

El código siguiente contiene el código de ejemplo que aparece en este tema.

```
#if INTERACTIVE
#r "FSharp.Data.TypeProviders.dll"
#r "System.Data.dll"
#r "System.Data.Linq.dll"
#endif
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq

type schema = SqlConnection<"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = schema.GetDataContext()

let data = [1; 5; 7; 11; 18; 21]

type Nullable<'T when 'T : ( new : unit -> 'T) and 'T : struct and 'T :> ValueType > with
    member this.Print() =
        if this.HasValue then this.Value.ToString()
        else "NULL"

printfn "\ncontains query operator"
query {
    for student in db.Student do
        select student.Age.Value
        contains 11
}
|> printfn "Is at least one student age 11? %b"

printfn "\ncount query operator"
query {
    for student in db.Student do
        select student
        count
}
|> printfn "Number of students: %d"

printfn "\nlast query operator."
let num =
    query {
        for number in data do
            sortBy number
            last
    }
printfn "last number: %d" num
```



```
printfn "Last number: %d" num
```

```
open Microsoft.FSharp.Linq
```

```
printfn "\nlastOrDefault query operator."
```

```
query {  
    for number in data do  
        sortBy number  
        lastOrDefault  
}  
|> printfn "lastOrDefault: %d"
```

```
printfn "\nexactlyOne query operator."
```

```
let student2 =  
    query {  
        for student in db.Student do  
            where (student.StudentID = 1)  
            select student  
            exactlyOne  
    }  
printfn "Student with StudentID = 1 is %s" student2.Name
```

```
printfn "\nexactlyOneOrDefault query operator."
```

```
let student3 =  
    query {  
        for student in db.Student do  
            where (student.StudentID = 1)  
            select student  
            exactlyOneOrDefault  
    }  
printfn "Student with StudentID = 1 is %s" student3.Name
```

```
printfn "\nheadOrDefault query operator."
```

```
let student4 =  
    query {  
        for student in db.Student do  
            select student  
            headOrDefault  
    }  
printfn "head student is %s" student4.Name
```

```
printfn "\nselect query operator."
```

```
query {  
    for student in db.Student do  
        select student  
}  
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)
```

```
printfn "\nwhere query operator."
```

```
query {  
    for student in db.Student do  
        where (student.StudentID > 4)  
        select student  
}  
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)
```

```
printfn "\nminBy query operator."
```

```
let student5 =  
    query {  
        for student in db.Student do  
            minBy student.StudentID  
    }  
printfn "minBy student is %s" student5.Name
```

```
printfn "\nmaxBy query operator."
```

```
let student6 =  
    query {  
        for student in db.Student do  
            maxBy student.StudentID  
    }  
printfn "maxBy student is %s" student6.Name
```

```

printfn "\ngroupBy query operator."
query {
    for student in db.Student do
        groupBy student.Age into g
        select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "Age: %s Count at that age: %d" (age.Print()) count)

printfn "\nsortBy query operator."
query {
    for student in db.Student do
        sortBy student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nsortByDescending query operator."
query {
    for student in db.Student do
        sortByDescending student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nthenBy query operator."
query {
    for student in db.Student do
        where student.Age.HasValue
        sortBy student.Age.Value
        thenBy student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\nthenByDescending query operator."
query {
    for student in db.Student do
        where student.Age.HasValue
        sortBy student.Age.Value
        thenByDescending student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\ngroupValBy query operator."
query {
    for student in db.Student do
        groupValBy student.Name student.Age into g
        select (g, g.Key, g.Count())
}
|> Seq.iter (fun (group, age, count) ->
    printfn "Age: %s Count at that age: %d" (age.Print()) count
    group |> Seq.iter (fun name -> printfn "Name: %s" name))

printfn "\n sumByNullable query operator"
query {
    for student in db.Student do
        sumByNullable student.Age
}
|> (fun sum -> printfn "Sum of ages: %s" (sum.Print()))

printfn "\n minByNullable"
query {
    for student in db.Student do
        minByNullable student.Age
}
|> (fun age -> printfn "Minimum age: %s" (age.Print()))

```

```

printfn "\n maxByNullable"
query {
    for student in db.Student do
        maxByNullable student.Age
}
|> (fun age -> printfn "Maximum age: %s" (age.Print()))

printfn "\n averageBy"
query {
    for student in db.Student do
        averageBy (float student.StudentID)
}
|> printfn "Average student ID: %f"

printfn "\n averageByNullable"
query {
    for student in db.Student do
        averageByNullable (Nullable.float student.Age)
}
|> (fun avg -> printfn "Average age: %s" (avg.Print()))

printfn "\n find query operator"
query {
    for student in db.Student do
        find (student.Name = "Abercrombie, Kim")
}
|> (fun student -> printfn "Found a match with StudentID = %d" student.StudentID)

printfn "\n all query operator"
query {
    for student in db.Student do
        all (SqlMethods.Like(student.Name, "%,%"))
}
|> printfn "Do all students have a comma in the name? %b"

printfn "\n head query operator"
query {
    for student in db.Student do
        head
}
|> (fun student -> printfn "Found the head student with StudentID = %d" student.StudentID)

printfn "\n nth query operator"
query {
    for numbers in data do
        nth 3
}
|> printfn "Third number is %d"

printfn "\n skip query operator"
query {
    for student in db.Student do
        skip 1
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n skipWhile query operator"
query {
    for number in data do
        skipWhile (number < 3)
        select number
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sumBy query operator"
query {
    for student in db.Student do
        sumBy student.StudentID
}

```

```

|> printfn "Sum of student IDs: %d"

printfn "\n take query operator"
query {
    for student in db.Student do
        select student
        take 2
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n takeWhile query operator"
query {
    for number in data do
        takeWhile (number < 10)
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sortByNullable query operator"
query {
    for student in db.Student do
        sortByNullable student.Age
        select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n sortByNullableDescending query operator"
query {
    for student in db.Student do
        sortByNullableDescending student.Age
        select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullable query operator"
query {
    for student in db.Student do
        sortBy student.Name
        thenByNullable student.Age
        select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullableDescending query operator"
query {
    for student in db.Student do
        sortBy student.Name
        thenByNullableDescending student.Age
        select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "All students: "
query {
    for student in db.Student do
        select student
}
|> Seq.iter (fun student -> printfn "%s %d %s" student.Name student.StudentID (student.Age.Print()))

printfn "\nCount of students: "
query {
    for student in db.Student do
        count
}
|> (fun count -> printfn "Student count: %d" count)

```

```

println "\nExists."
query {
    for student in db.Student do
        where
            (query {
                for courseSelection in db.CourseSelection do
                    exists (courseSelection.StudentID = student.StudentID) })
            select student
    }
    |> Seq.iter (fun student -> println "%A" student.Name)

println "\n Group by age and count"
query {
    for n in db.Student do
        groupBy n.Age into g
        select (g.Key, g.Count())
    }
    |> Seq.iter (fun (age, count) -> println "%s %d" (age.Print()) count)

println "\n Group value by age."
query {
    for n in db.Student do
        groupValBy n.Age n.Age into g
        select (g.Key, g.Count())
    }
    |> Seq.iter (fun (age, count) -> println "%s %d" (age.Print()) count)

println "\nGroup students by age where age > 10."
query {
    for student in db.Student do
        groupBy student.Age into g
        where (g.Key.HasValue && g.Key.Value > 10)
        select (g, g.Key)
    }
    |> Seq.iter (fun (students, age) ->
        println "Age: %s" (age.Value.ToString())
        students
        |> Seq.iter (fun student -> println "%s" student.Name))

println "\nGroup students by age and print counts of number of students at each age with more than 1 student."
query {
    for student in db.Student do
        groupBy student.Age into group
        where (group.Count() > 1)
        select (group.Key, group.Count())
    }
    |> Seq.iter (fun (age, ageCount) ->
        println "Age: %s Count: %d" (age.Print()) ageCount)

println "\nGroup students by age and sum ages."
query {
    for student in db.Student do
        groupBy student.Age into g
        let total = query { for student in g do sumByNullable student.Age }
        select (g.Key, g.Count(), total)
    }
    |> Seq.iter (fun (age, count, total) ->
        println "Age: %d" (age.GetValueOrDefault())
        println "Count: %d" count
        println "Total years: %s" (total.ToString()))

println "\nGroup students by age and count number of students at each age, and display all with count > 1 in descending order of count."
query {
    for student in db.Student do
        groupBy student.Age into g
        where (g.Count() > 1)
        sortByDescending (g.Count())

```

```

        select (g.Key, g.Count())
    }
    |> Seq.iter (fun (age, myCount) ->
        printfn "Age: %s" (age.Print())
        printfn "Count: %d" myCount)

printfn "\n Select students from a set of IDs"
let idList = [1; 2; 5; 10]
let idQuery =
    query { for id in idList do select id }
query {
    for student in db.Student do
        where (idQuery.Contains(student.StudentID))
        select student
}
|> Seq.iter (fun student ->
    printfn "Name: %s" student.Name)

printfn "\nLook for students with Name match _e%% pattern and take first two."
query {
    for student in db.Student do
        where (SqlMethods.Like( student.Name, "_e%") )
        select student
        take 2
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with Name matching [abc]%% pattern."
query {
    for student in db.Student do
        where (SqlMethods.Like( student.Name, "[abc]%" )
        select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern."
query {
    for student in db.Student do
        where (SqlMethods.Like( student.Name, "[^abc]%" )
        select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern and select ID."
query {
    for n in db.Student do
        where (SqlMethods.Like( n.Name, "[^abc]%" )
        select n.StudentID
}
|> Seq.iter (fun id -> printfn "%d" id)

printfn "\n Using Contains as a query filter."
query {
    for student in db.Student do
        where (student.Name.Contains("a"))
        select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nSearching for names from a list."
let names = [|"a";"b";"c"|]
query {
    for student in db.Student do
        if names.Contains (student.Name) then select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nJoin Student and CourseSelection tables."
query {

```

```

    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    select (student, selection)
}
|> Seq.iter (fun (student, selection) -> printfn "%d %s %d" student.StudentID student.Name
selection.CourseID)

printfn "\nLeft Join Student and CourseSelection tables."
query {
    for student in db.Student do
    leftOuterJoin selection in db.CourseSelection
        on (student.StudentID = selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
    select (student, selection)
}
|> Seq.iter (fun (student, selection) ->
    let selectionID, studentID, courseID =
        match selection with
        | null -> "NULL", "NULL", "NULL"
        | sel -> (sel.ID.ToString(), sel.StudentID.ToString(), sel.CourseID.ToString())
    printfn "%d %s %d %s %s %s" student.StudentID student.Name (student.Age.GetValueOrDefault()) selectionID
studentID courseID)

printfn "\nJoin with count"
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    count
}
|> printfn "%d"

printfn "\n Join with distinct."
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    distinct
}
|> Seq.iter (fun (student, selection) -> printfn "%s %d" student.Name selection.CourseID)

printfn "\n Join with distinct and count."
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    distinct
    count
}
|> printfn "%d"

printfn "\n Selecting students with age between 10 and 15."
query {
    for student in db.Student do
    where (student.Age.Value >= 10 && student.Age.Value < 15)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students with age either 11 or 12."
query {
    for student in db.Student do
    where (student.Age.Value = 11 || student.Age.Value = 12)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students in a certain age range and counting "

```

```

println "\n Selecting students in a certain age range and sorting."
query {
    for n in db.Student do
        where (n.Age.Value = 12 || n.Age.Value = 13)
        sortByNullableDescending n.Age
        select n
    }
|> Seq.iter (fun student -> printfn "%s %s" student.Name (student.Age.Print()))

println "\n Selecting students with certain ages, taking account of possibility of nulls."
query {
    for student in db.Student do
        where
            ((student.Age.HasValue && student.Age.Value = 11) ||
             (student.Age.HasValue && student.Age.Value = 12))
        sortByDescending student.Name
        select student.Name
        take 2
    }
|> Seq.iter (fun name -> printfn "%s" name)

println "\n Union of two queries."
module Queries =
    let query1 = query {
        for n in db.Student do
            select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
            select (n.Name, n.Age)
    }

    query2.Union (query1)
|> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

println "\n Intersect of two queries."
module Queries2 =
    let query1 = query {
        for n in db.Student do
            select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
            select (n.Name, n.Age)
    }

    query1.Intersect(query2)
|> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

println "\n Using if statement to alter results for special value."
query {
    for student in db.Student do
        select
            (if student.Age.HasValue && student.Age.Value = -1 then
                (student.StudentID, System.Nullable<int>(100), student.Age)
            else (student.StudentID, student.Age, student.Age))
    }
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

println "\n Using if statement to alter results special values."
query {
    for student in db.Student do
        select
            (if student.Age.HasValue && student.Age.Value = -1 then
                (student.StudentID, System.Nullable<int>(100), student.Age)
            elif student.Age.HasValue && student.Age.Value = 0 then
                (student.StudentID, System.Nullable<int>(100), student.Age)

```



```

        else (student.StudentID, student.Age, student.Age))
    }
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Multiple table select."
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
|> Seq.iteri (fun index (student, course) ->
    if index = 0 then
        printfn "StudentID Name Age CourseID CourseName"
        printfn "%d %s %s %d %s" student.StudentID student.Name (student.Age.Print()) course.CourseID
        course.CourseName)

printfn "\nMultiple Joins"
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID = course.CourseID)
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)

printfn "\nMultiple Left Outer Joins"
query {
    for student in db.Student do
    leftOuterJoin courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty() do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID = course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)

```

Y este es el resultado completo cuando este código se ejecuta en F# interactivo.

```

--> Referenced 'C:\Program Files (x86)\Reference Assemblies\Microsoft\FSharp\3.0\Runtime\v4.0\Type
Providers\FSharp.Data.TypeProviders.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.Linq.dll'

contains query operator
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp5E3C.dll'...
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp611A.dll'...
Is at least one student age 11? true

count query operator
Number of students: 8

last query operator.
Last number: 21

lastOrDefault query operator.
lastOrDefault: 21

exactlyOne query operator.
Student with StudentID = 1 is Abercrombie, Kim

exactlyOneOrDefault query operator.
Student with StudentID = 1 is Abercrombie, Kim

```

Student with StudentID = 1 is Abercrombie, Kim

headOrDefault query operator.

head student is Abercrombie, Kim

select query operator.

StudentID, Name: 1 Abercrombie, Kim

StudentID, Name: 2 Abolrous, Hazen

StudentID, Name: 3 Hance, Jim

StudentID, Name: 4 Adams, Terry

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 6 Penor, Lori

StudentID, Name: 7 Perham, Tom

StudentID, Name: 8 Peng, Yun-Feng

where query operator.

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 6 Penor, Lori

StudentID, Name: 7 Perham, Tom

StudentID, Name: 8 Peng, Yun-Feng

minBy query operator.

maxBy query operator.

groupBy query operator.

Age: NULL Count at that age: 1

Age: 10 Count at that age: 1

Age: 11 Count at that age: 1

Age: 12 Count at that age: 3

Age: 13 Count at that age: 1

Age: 14 Count at that age: 1

sortBy query operator.

StudentID, Name: 1 Abercrombie, Kim

StudentID, Name: 2 Abolrous, Hazen

StudentID, Name: 4 Adams, Terry

StudentID, Name: 3 Hance, Jim

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 8 Peng, Yun-Feng

StudentID, Name: 6 Penor, Lori

StudentID, Name: 7 Perham, Tom

sortByDescending query operator.

StudentID, Name: 7 Perham, Tom

StudentID, Name: 6 Penor, Lori

StudentID, Name: 8 Peng, Yun-Feng

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 3 Hance, Jim

StudentID, Name: 4 Adams, Terry

StudentID, Name: 2 Abolrous, Hazen

StudentID, Name: 1 Abercrombie, Kim

thenBy query operator.

StudentID, Name: 10 Abercrombie, Kim

StudentID, Name: 11 Hansen, Claus

StudentID, Name: 12 Adams, Terry

StudentID, Name: 12 Hance, Jim

StudentID, Name: 12 Perham, Tom

StudentID, Name: 13 Penor, Lori

StudentID, Name: 14 Abolrous, Hazen

thenByDescending query operator.

StudentID, Name: 10 Abercrombie, Kim

StudentID, Name: 11 Hansen, Claus

StudentID, Name: 12 Perham, Tom

StudentID, Name: 12 Hance, Jim

StudentID, Name: 12 Adams, Terry

StudentID, Name: 13 Penor, Lori

StudentID, Name: 14 Abolrous, Hazen

groupBy query operator.

Age: NULL Count at that age: 1

Name: Peng, Yun-Feng

Age: 10 Count at that age: 1

Name: Abercrombie, Kim

Age: 11 Count at that age: 1

Name: Hansen, Claus

Age: 12 Count at that age: 3

Name: Hance, Jim

Name: Adams, Terry

Name: Perham, Tom

Age: 13 Count at that age: 1

Name: Penor, Lori

Age: 14 Count at that age: 1

Name: Abolrous, Hazen

sumByNullable query operator

Sum of ages: 84

minByNullable

Minimum age: 10

maxByNullable

Maximum age: 14

averageBy

Average student ID: 4.500000

averageByNullable

Average age: 12

find query operator

Found a match with StudentID = 1

all query operator

Do all students have a comma in the name? true

head query operator

Found the head student with StudentID = 1

nth query operator

Third number is 11

skip query operator

StudentID = 2

StudentID = 3

StudentID = 4

StudentID = 5

StudentID = 6

StudentID = 7

StudentID = 8

skipWhile query operator

Number = 5

Number = 7

Number = 11

Number = 18

Number = 21

sumBy query operator

Sum of student IDs: 36

take query operator

StudentID = 1

StudentID = 2

takeWhile query operator

Number = 1  
Number = 5  
Number = 7

sortByNullable query operator

StudentID, Name, Age: 8 Peng, Yun-Feng NULL  
StudentID, Name, Age: 1 Abercrombie, Kim 10  
StudentID, Name, Age: 5 Hansen, Claus 11  
StudentID, Name, Age: 7 Perham, Tom 12  
StudentID, Name, Age: 3 Hance, Jim 12  
StudentID, Name, Age: 4 Adams, Terry 12  
StudentID, Name, Age: 6 Penor, Lori 13  
StudentID, Name, Age: 2 Abolrous, Hazen 14

sortByNullableDescending query operator

StudentID, Name, Age: 2 Abolrous, Hazen 14  
StudentID, Name, Age: 6 Penor, Lori 13  
StudentID, Name, Age: 7 Perham, Tom 12  
StudentID, Name, Age: 3 Hance, Jim 12  
StudentID, Name, Age: 4 Adams, Terry 12  
StudentID, Name, Age: 5 Hansen, Claus 11  
StudentID, Name, Age: 1 Abercrombie, Kim 10  
StudentID, Name, Age: 8 Peng, Yun-Feng NULL

thenByNullable query operator

StudentID, Name, Age: 1 Abercrombie, Kim 10  
StudentID, Name, Age: 2 Abolrous, Hazen 14  
StudentID, Name, Age: 4 Adams, Terry 12  
StudentID, Name, Age: 3 Hance, Jim 12  
StudentID, Name, Age: 5 Hansen, Claus 11  
StudentID, Name, Age: 8 Peng, Yun-Feng NULL  
StudentID, Name, Age: 6 Penor, Lori 13  
StudentID, Name, Age: 7 Perham, Tom 12

thenByNullableDescending query operator

StudentID, Name, Age: 1 Abercrombie, Kim 10  
StudentID, Name, Age: 2 Abolrous, Hazen 14  
StudentID, Name, Age: 4 Adams, Terry 12  
StudentID, Name, Age: 3 Hance, Jim 12  
StudentID, Name, Age: 5 Hansen, Claus 11  
StudentID, Name, Age: 8 Peng, Yun-Feng NULL  
StudentID, Name, Age: 6 Penor, Lori 13  
StudentID, Name, Age: 7 Perham, Tom 12

All students:

Abercrombie, Kim 1 10  
Abolrous, Hazen 2 14  
Hance, Jim 3 12  
Adams, Terry 4 12  
Hansen, Claus 5 11  
Penor, Lori 6 13  
Perham, Tom 7 12  
Peng, Yun-Feng 8 NULL

Count of students:

Student count: 8

Exists.

"Abercrombie, Kim"

"Abolrous, Hazen"

"Hance, Jim"

"Adams, Terry"

"Hansen, Claus"

"Perham, Tom"

Group by age and count

NULL 1

10 1

11 1

12 3

```
13 1
14 1
```

Group value by age.

```
NULL 1
10 1
11 1
12 3
13 1
14 1
```

Group students by age where age > 10.

```
Age: 11
Hansen, Claus
Age: 12
Hance, Jim
Adams, Terry
Perham, Tom
Age: 13
Penor, Lori
Age: 14
Abolrous, Hazen
```

Group students by age and print counts of number of students at each age with more than 1 student.

```
Age: 12 Count: 3
```

Group students by age and sum ages.

```
Age: 0
Count: 1
Total years:
Age: 10
Count: 1
Total years: 10
Age: 11
Count: 1
Total years: 11
Age: 12
Count: 3
Total years: 36
Age: 13
Count: 1
Total years: 13
Age: 14
Count: 1
Total years: 14
```

Group students by age and count number of students at each age, and display all with count > 1 in descending order of count.

```
Age: 12
Count: 3
```

Select students from a set of IDs

```
Name: Abercrombie, Kim
Name: Abolrous, Hazen
Name: Hansen, Claus
```

Look for students with Name match _e% pattern and take first two.

```
Penor, Lori
Perham, Tom
```

Look for students with Name matching [abc]% pattern.

```
Abercrombie, Kim
Abolrous, Hazen
Adams, Terry
```

Look for students with name matching [^abc]% pattern.

```
Hance, Jim
Hansen, Claus
Penor, Lori
```

Perham, Tom  
Peng, Yun-Feng

Look for students with name matching [^abc]% pattern and select ID.

3  
5  
6  
7  
8

Using Contains as a query filter.

Abercrombie, Kim  
Abolrous, Hazen  
Hance, Jim  
Adams, Terry  
Hansen, Claus  
Perham, Tom

Searching for names from a list.

Join Student and CourseSelection tables.

2 Abolrous, Hazen 2  
3 Hance, Jim 3  
5 Hansen, Claus 5  
2 Abolrous, Hazen 2  
5 Hansen, Claus 5  
6 Penor, Lori 6  
3 Hance, Jim 3  
2 Abolrous, Hazen 2  
1 Abercrombie, Kim 1  
2 Abolrous, Hazen 2  
5 Hansen, Claus 5  
2 Abolrous, Hazen 2  
3 Hance, Jim 3  
2 Abolrous, Hazen 2  
3 Hance, Jim 3

Left Join Student and CourseSelection tables.

1 Abercrombie, Kim 10 9 3 1  
2 Abolrous, Hazen 14 1 1 2  
2 Abolrous, Hazen 14 4 2 2  
2 Abolrous, Hazen 14 8 3 2  
2 Abolrous, Hazen 14 10 4 2  
2 Abolrous, Hazen 14 12 4 2  
2 Abolrous, Hazen 14 14 5 2  
3 Hance, Jim 12 2 1 3  
3 Hance, Jim 12 7 2 3  
3 Hance, Jim 12 13 5 3  
3 Hance, Jim 12 15 7 3  
4 Adams, Terry 12 NULL NULL NULL  
5 Hansen, Claus 11 3 1 5  
5 Hansen, Claus 11 5 2 5  
5 Hansen, Claus 11 11 4 5  
6 Penor, Lori 13 6 2 6  
7 Perham, Tom 12 NULL NULL NULL  
8 Peng, Yun-Feng 0 NULL NULL NULL

Join with count

15

Join with distinct.

Abercrombie, Kim 2  
Abercrombie, Kim 3  
Abercrombie, Kim 5  
Abolrous, Hazen 2  
Abolrous, Hazen 5  
Abolrous, Hazen 6  
Abolrous, Hazen 3  
Hance, Jim 2

```
Hance, Jim 2
Hance, Jim 1
Adams, Terry 2
Adams, Terry 5
Adams, Terry 2
Hansen, Claus 3
Hansen, Claus 2
Perham, Tom 3
```

Join with distinct and count.  
15

Selecting students with age between 10 and 15.  
Abercrombie, Kim  
Abolrous, Hazen  
Hance, Jim  
Adams, Terry  
Hansen, Claus  
Penor, Lori  
Perham, Tom

Selecting students with age either 11 or 12.  
Hance, Jim  
Adams, Terry  
Hansen, Claus  
Perham, Tom

Selecting students in a certain age range and sorting.  
Penor, Lori 13  
Perham, Tom 12  
Hance, Jim 12  
Adams, Terry 12

Selecting students with certain ages, taking account of possibility of nulls.  
Hance, Jim  
Adams, Terry

Union of two queries.  
Abercrombie, Kim 10  
Abolrous, Hazen 14  
Hance, Jim 12  
Adams, Terry 12  
Hansen, Claus 11  
Penor, Lori 13  
Perham, Tom 12  
Peng, Yun-Feng NULL

Intersect of two queries.

Using if statement to alter results for special value.  
1 10 10  
2 14 14  
3 12 12  
4 12 12  
5 11 11  
6 13 13  
7 12 12  
8 NULL NULL

Using if statement to alter results special values.  
1 10 10  
2 14 14  
3 12 12  
4 12 12  
5 11 11  
6 13 13  
7 12 12  
8 NULL NULL

Multiple table select

Multiple table select.

StudentID	Name	Age	CourseID	CourseName
-----------	------	-----	----------	------------

1	Abercrombie, Kim	10	1	Algebra I
2	Abolrous, Hazen	14	1	Algebra I
3	Hance, Jim	12	1	Algebra I
4	Adams, Terry	12	1	Algebra I
5	Hansen, Claus	11	1	Algebra I
6	Penor, Lori	13	1	Algebra I
7	Perham, Tom	12	1	Algebra I
8	Peng, Yun-Feng	NULL	1	Algebra I
1	Abercrombie, Kim	10	2	Trigonometry
2	Abolrous, Hazen	14	2	Trigonometry
3	Hance, Jim	12	2	Trigonometry
4	Adams, Terry	12	2	Trigonometry
5	Hansen, Claus	11	2	Trigonometry
6	Penor, Lori	13	2	Trigonometry
7	Perham, Tom	12	2	Trigonometry
8	Peng, Yun-Feng	NULL	2	Trigonometry
1	Abercrombie, Kim	10	3	Algebra II
2	Abolrous, Hazen	14	3	Algebra II
3	Hance, Jim	12	3	Algebra II
4	Adams, Terry	12	3	Algebra II
5	Hansen, Claus	11	3	Algebra II
6	Penor, Lori	13	3	Algebra II
7	Perham, Tom	12	3	Algebra II
8	Peng, Yun-Feng	NULL	3	Algebra II
1	Abercrombie, Kim	10	4	History
2	Abolrous, Hazen	14	4	History
3	Hance, Jim	12	4	History
4	Adams, Terry	12	4	History
5	Hansen, Claus	11	4	History
6	Penor, Lori	13	4	History
7	Perham, Tom	12	4	History
8	Peng, Yun-Feng	NULL	4	History
1	Abercrombie, Kim	10	5	English
2	Abolrous, Hazen	14	5	English
3	Hance, Jim	12	5	English
4	Adams, Terry	12	5	English
5	Hansen, Claus	11	5	English
6	Penor, Lori	13	5	English
7	Perham, Tom	12	5	English
8	Peng, Yun-Feng	NULL	5	English
1	Abercrombie, Kim	10	6	French
2	Abolrous, Hazen	14	6	French
3	Hance, Jim	12	6	French
4	Adams, Terry	12	6	French
5	Hansen, Claus	11	6	French
6	Penor, Lori	13	6	French
7	Perham, Tom	12	6	French
8	Peng, Yun-Feng	NULL	6	French
1	Abercrombie, Kim	10	7	Chinese
2	Abolrous, Hazen	14	7	Chinese
3	Hance, Jim	12	7	Chinese
4	Adams, Terry	12	7	Chinese
5	Hansen, Claus	11	7	Chinese
6	Penor, Lori	13	7	Chinese
7	Perham, Tom	12	7	Chinese
8	Peng, Yun-Feng	NULL	7	Chinese

Multiple Joins

Abercrombie, Kim	Trigonometry
Abercrombie, Kim	Algebra II
Abercrombie, Kim	English
Abolrous, Hazen	Trigonometry
Abolrous, Hazen	English
Abolrous, Hazen	French
Abolrous, Hazen	Algebra II
Hance, Jim	Trigonometry
Hance, Jim	Algebra I



Adams, Terry Trigonometry  
Adams, Terry English  
Adams, Terry Trigonometry  
Hansen, Claus Algebra II  
Hansen, Claus Trigonometry  
Perham, Tom Algebra II

Multiple Left Outer Joins  
Abercrombie, Kim Trigonometry  
Abercrombie, Kim Algebra II  
Abercrombie, Kim English  
Abolrous, Hazen Trigonometry  
Abolrous, Hazen English  
Abolrous, Hazen French  
Abolrous, Hazen Algebra II  
Hance, Jim Trigonometry  
Hance, Jim Algebra I  
Adams, Terry Trigonometry  
Adams, Terry English  
Adams, Terry Trigonometry  
Hansen, Claus Algebra II  
Hansen, Claus Trigonometry  
Penor, Lori  
Perham, Tom Algebra II  
Peng, Yun-Feng

```
type schema
val db : schema.ServiceTypes.SimpleDataContextTypes.MyDatabase1
val student : System.Data.Linq.Table<schema.ServiceTypes.Student>
val data : int list = [1; 5; 7; 11; 18; 21]
type Nullable<'T
    when 'T : (new : unit -> 'T) and 'T : struct and
        'T :> System.ValueType> with
    member Print : unit -> string
val num : int = 21
val student2 : schema.ServiceTypes.Student
val student3 : schema.ServiceTypes.Student
val student4 : schema.ServiceTypes.Student
val student5 : int = 1
val student6 : int = 8
val idList : int list = [1; 2; 5; 10]
val idQuery : seq<int>
val names : string [] = [|"a"; "b"; "c"|]
module Queries = begin
    val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
    val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
module Queries2 = begin
    val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
    val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
```

## Consulta también

- [Referencia del lenguaje F #](#)
- [Linq. QueryBuilder \(clase\)](#)
- [Expresiones de cálculo](#)

# Expresiones de código delimitadas

23/10/2019 • 12 minutes to read • [Edit Online](#)

## NOTE

El vínculo de la referencia de API le llevará a MSDN. La referencia de API de docs.microsoft.com no está completa.

En este tema se describen las expresiones de *código* delimitadas, una característica del lenguaje que F# permite generar y trabajar con expresiones de código mediante programación. Esta característica permite generar un árbol de sintaxis abstracta que representa F# el código. A continuación, el árbol de sintaxis abstracta puede atravesarse y procesarse según las necesidades de la aplicación. Por ejemplo, puede usar el árbol para generar F# código o generar código en otro lenguaje.

## Expresiones entre comillas

Una *expresión entre comillas* F# es una expresión del código que está delimitada de manera que no se compila como parte del programa, sino que se compila en un objeto que representa una F# expresión. Puede marcar una expresión entre comillas de una de estas dos maneras: con información de tipo o sin información de tipo. Si desea incluir información de tipo, use los símbolos `<@` y `@>` para delimitar la expresión entre comillas. Si no necesita información de tipo, use los símbolos `<@@` y `@@>`. En el código siguiente se muestran las comillas con tipo y sin tipo.

```
open Microsoft.FSharp.Quotations
// A typed code quotation.
let expr : Expr<int> = <@ 1 + 1 @>
// An untyped code quotation.
let expr2 : Expr = <@@ 1 + 1 @@>
```

Atravesar un árbol de expresión grande es más rápido si no incluye información de tipo. El tipo resultante de una expresión entre comillas con los símbolos `<Expr<'T>` tipo es, donde el parámetro de tipo tiene el tipo de la expresión determinado F# por el algoritmo de inferencia de tipos del compilador. Cuando se usan expresiones de código delimitadas sin información de tipos, el tipo de la expresión entrecomillada es el tipo no genérico `Expr`. Puede llamar a la propiedad `raw` en la `Expr` clase con tipo para obtener `Expr` el objeto sin tipo.

Hay una variedad de métodos estáticos que le permiten generar F# objetos de expresión mediante programación en la `Expr` clase sin usar expresiones entre comillas.

Tenga en cuenta que una expresión de código delimitada debe incluir una expresión completa. Para un `let` enlace, por ejemplo, necesitará la definición del nombre enlazado y una expresión adicional que use el enlace. En la sintaxis detallada, se trata de una expresión que sigue `in` a la palabra clave. En el nivel superior de un módulo, se trata simplemente de la siguiente expresión del módulo, pero en una expresión de código delimitada, se requiere explícitamente.

Por lo tanto, la siguiente expresión no es válida.

```
// Not valid:
// <@ let f x = x + 1 @>
```

Pero las siguientes expresiones son válidas.

```
// Valid:
<@ let f x = x + 10 in f 20 @>
// Valid:
<@
    let f x = x + 10
    f 20
@>
```

Para evaluar F# las comillas, debe utilizar el [F# evaluador](#) de Comillas. Proporciona compatibilidad para evaluar y ejecutar objetos de F# expresión.

## Tipo de expresión

Una instancia del `Expr` tipo representa una F# expresión. Los tipos genéricos y no genéricos `Expr` están documentados en la F# documentación de la biblioteca. Para obtener más información, vea [Microsoft. FSharp. Quotations \(espacio de nombres\)](#) y [Quotations . expr \(clase\)](#).

## Operadores de inserción

La inserción permite combinar las expresiones de código delimitadas de literales con las expresiones que ha creado mediante programación o desde otro código de Comillas. Los `%` operadores `%%` y permiten agregar un F# objeto Expression en una expresión de código. El `%` operador se usa para insertar un objeto de expresión con tipo en una comilla con tipo; el `%%` operador se usa para insertar un objeto de expresión sin tipo en una comilla sin tipo. Ambos operadores son operadores de prefijo unario. Por lo `expr` tanto, si es una expresión sin tipo `Expr` de tipo, el código siguiente es válido.

```
<@@ 1 + %%expr @@>
```

Y si `expr` es una comilla con tipo de `Expr<int>` tipo, el código siguiente es válido.

```
<@ 1 + %expr @>
```

## Ejemplo

### DESCRIPCIÓN

En el ejemplo siguiente se muestra el uso de expresiones de código delimitadas para colocar F# el código en un objeto F# de expresión y, a continuación, imprimir el código que representa la expresión. Se define `println` una función que contiene una función `print` recursiva que muestra un F# objeto de expresión (de tipo `Expr`) en un formato descriptivo. Hay varios patrones activos en los módulos [Microsoft. FSharp. Quotations. Patterns](#) y [Microsoft. FSharp. Quotations. DerivedPatterns](#) que se pueden usar para analizar objetos de expresión. En este ejemplo no se incluyen todos los patrones posibles que pueden aparecer en F# una expresión. Cualquier patrón no reconocido desencadena una coincidencia con el patrón de caracteres comodín `_` () y se representa mediante el `ToString` método, que, en el `Expr` tipo, permite conocer el modelo activo que se va a agregar a la expresión de coincidencia.

### Código

```
module Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

let println expr =
```

```

let rec print expr =
  match expr with
  | Application(expr1, expr2) ->
    // Function application.
    print expr1
    printf " "
    print expr2
  | SpecificCall <@@ (+) @@> (_, _, exprList) ->
    // Matches a call to (+). Must appear before Call pattern.
    print exprList.Head
    printf " + "
    print exprList.Tail.Head
  | Call(exprOpt, methodInfo, exprList) ->
    // Method or module function call.
    match exprOpt with
    | Some expr -> print expr
    | None -> printf "%s" methodInfo.DeclaringType.Name
    printf ".%s(" methodInfo.Name
    if (exprList.IsEmpty) then printf ")" else
    print exprList.Head
    for expr in exprList.Tail do
      printf ","
      print expr
    printf ")"
  | Int32(n) ->
    printf "%d" n
  | Lambda(param, body) ->
    // Lambda expression.
    printf "fun (%s:%s) -> " param.Name (param.Type.ToString())
    print body
  | Let(var, expr1, expr2) ->
    // Let binding.
    if (var.IsMutable) then
      printf "let mutable %s = " var.Name
    else
      printf "let %s = " var.Name
    print expr1
    printf " in "
    print expr2
  | PropertyGet(_, propOrValInfo, _) ->
    printf "%s" propOrValInfo.Name
  | String(str) ->
    printf "%s" str
  | Value(value, typ) ->
    printf "%s" (value.ToString())
  | Var(var) ->
    printf "%s" var.Name
  | _ -> printf "%s" (expr.ToString())
print expr
printfn ""

```

```

let a = 2

```

```

// exprLambda has type "(int -> int)".
let exprLambda = <@ fun x -> x + 1 @>
// exprCall has type unit.
let exprCall = <@ a + 1 @>

println exprLambda
println exprCall
println <@@ let f x = x + 10 in f 10 @@>

```

**Salida**

```
fun (x:System.Int32) -> x + 1
a + 1
let f = fun (x:System.Int32) -> x + 10 in f 10
```

## Ejemplo

### DESCRIPCIÓN

También puede usar los tres patrones activos en el [módulo ExprShape](#) para recorrer los árboles de expresión con menos patrones activos. Estos modelos activos pueden ser útiles si desea atravesar un árbol, pero no necesita toda la información en la mayoría de los nodos. Cuando se usan estos patrones, cualquier F# expresión coincide con uno de los tres patrones siguientes `ShapeVar` : Si la expresión es una variable `ShapeLambda` , si la expresión es una expresión lambda, `ShapeCombination` o si la expresión es otra cosa. Si atraviesa un árbol de expresión usando los modelos activos como en el ejemplo de código anterior, tendrá que usar muchos más patrones para controlar todos los tipos de F# expresión posibles y el código será más complejo. Para obtener más información, consulte [ExprShape.ShapeVarShapeLambda ShapeCombination \(Active Pattern\)](#).

El siguiente ejemplo de código se puede utilizar como base para los recorridos más complejos. En este código, se crea un árbol de expresión para una expresión que implica una llamada a `add` función,. El modelo activo `SpecificCall` ( se usa para detectar cualquier llamada a `add` en el árbol de expresión. Este modelo activo asigna los argumentos de la llamada al `exprList` valor. En este caso, solo hay dos, por lo que se extraen y se llama a la función de forma recursiva en los argumentos. Los resultados se insertan en una expresión de código que representa una `mul` llamada a mediante el operador de inserción `%%` (). La `println` función del ejemplo anterior se usa para mostrar los resultados.

El código de las otras ramas del modelo activo simplemente vuelve a generar el mismo árbol de expresión, por lo que el único cambio en la expresión resultante `add` es `mul` el cambio de a.

### Código

```
module Module1
open Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.DerivedPatterns
open Microsoft.FSharp.Quotations.ExprShape

let add x y = x + y
let mul x y = x * y

let rec substituteExpr expression =
    match expression with
    | SpecificCall <@@ add @@> (_, _, exprList) ->
        let lhs = substituteExpr exprList.Head
        let rhs = substituteExpr exprList.Tail.Head
        <@@ mul %%lhs %%rhs @@>
    | ShapeVar var -> Expr.Var var
    | ShapeLambda (var, expr) -> Expr.Lambda (var, substituteExpr expr)
    | ShapeCombination(shapeComboObject, exprList) ->
        RebuildShapeCombination(shapeComboObject, List.map substituteExpr exprList)

let expr1 = <@@ 1 + (add 2 (add 3 4)) @@>
println expr1
let expr2 = substituteExpr expr1
println expr2
```

### Salida

```
1 + Module1.add(2,Module1.add(3,4))  
1 + Module1.mul(2,Module1.mul(3,4))
```

## Vea también

- [Referencia del lenguaje F#](#)

# La palabra clave fixed

23/10/2019 • 2 minutes to read • [Edit Online](#)

F#4.1 presenta el `fixed` palabra clave, lo que permite que una variable local en la pila para evitar que se recopilan o movido durante la recolección de elementos no utilizados "anclar". Se usa para escenarios de programación de bajo nivel.

## Sintaxis

```
use ptr = fixed expression
```

## Comentarios

Esto amplía la sintaxis de expresiones para permitir que un puntero de extracción y se enlaza a un nombre que ha impedido que se recopilan o movido durante la recolección de elementos no utilizados.

Un puntero de una expresión que se ha corregido a través de la `fixed` palabra clave está enlazado a un identificador a través de la `use` palabra clave. La semántica de esto es similar a la administración de recursos a través de la `use` palabra clave. El puntero se fija mientras esté dentro del ámbito, y una vez que esté fuera del ámbito, ya no es fijo. `fixed` no se puede usar fuera del contexto de un `use` enlace. Debe enlazar el puntero a un nombre con `use`.

El uso de `fixed` debe producirse dentro de una expresión en una función o un método. No se puede usar en un ámbito de nivel de módulo o script.

Como todo el código de puntero, esto es una característica no segura y emitirá una advertencia cuando se usa.

## Ejemplo

```

open Microsoft.FSharp.NativeInterop

type Point = { mutable X: int; mutable Y: int}

let squareWithPointer (p: nativeptr<int>) =
    // Dereference the pointer at the 0th address.
    let mutable value = NativePtr.get p 0

    // Perform some work
    value <- value * value

    // Set the value in the pointer at the 0th address.
    NativePtr.set p 0 value

let pnt = { X = 1; Y = 2 }
printfn "pnt before - X: %d Y: %d" pnt.X pnt.Y // prints 1 and 2

// Note that the use of 'fixed' is inside a function.
// You cannot fix a pointer at a script-level or module-level scope.
let doPointerWork() =
    use ptr = fixed &pnt.Y

    // Square the Y value
    squareWithPointer ptr
    printfn "pnt after - X: %d Y: %d" pnt.X pnt.Y // prints 1 and 4

doPointerWork()

```

## Vea también

- [NativePtr \(módulo\)](#)



# Byrefs

19/03/2020 • 11 minutes to read • [Edit Online](#)

F tiene dos áreas de características principales que se ocupa en el espacio de la programación de bajo nivel:

- `byref` / Los `inref` / tipos, que son punteros administrados. `outref` Tienen restricciones de uso para que no pueda compilar un programa que no sea válido en tiempo de ejecución.
- Una `byref` estructura -like, que es una **estructura** que tiene una semántica `byref<'T>` similar y las mismas restricciones en tiempo de compilación que . Un ejemplo `Span<T>` es .

## Sintaxis

```
// Byref types as parameters
let f (x: byref<'T>) = ()
let g (x: inref<'T>) = ()
let h (x: outref<'T>) = ()

// Calling a function with a byref parameter
let mutable x = 3
f &x

// Declaring a byref-like struct
open System.Runtime.CompilerServices

[<Struct; IsByRefLike>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

## Byref, inref y outref

Hay tres formas `byref` de:

- `inref<'T>` , un puntero administrado para leer el valor subyacente.
- `outref<'T>` , un puntero administrado para escribir en el valor subyacente.
- `byref<'T>` , un puntero administrado para leer y escribir el valor subyacente.

A `byref<'T>` se puede `inref<'T>` pasar donde se espera un. Del mismo `byref<'T>` modo, se `outref<'T>` puede pasar a donde se espera un.

## Uso de byrefs

Para utilizar `inref<'T>` un , es necesario `&` obtener un valor de puntero con:

```
open System

let f (dt: inref<DateTime>) =
    printfn "Now: %s" (dt.ToString())

let usage =
    let dt = DateTime.Now
    f &dt // Pass a pointer to 'dt'
```

Para escribir en el `outref<'T>` puntero `byref<'T>` mediante un `o`, también debe `mutable` hacer que el valor que se captura un puntero a .

```
open System

let f (dt: byref<DateTime>) =
    printfn "Now: %s" (dt.ToString())
    dt <- DateTime.Now

// Make 'dt' mutable
let mutable dt = DateTime.Now

// Now you can pass the pointer to 'dt'
f &dt
```

Si solo está escribiendo el puntero en `outref<'T>` lugar `byref<'T>` de leerlo, considere la posibilidad de usar archivos en lugar de .

### Semántica inref

Observe el código siguiente:

```
let f (x: inref<SomeStruct>) = x.SomeField
```

Semánticamente, esto significa lo siguiente:

- El titular `x` del puntero solo puede usarlo para leer el valor.
- Cualquier puntero `struct` adquirido a `SomeStruct` los `inref<_>` campos anidados dentro se les da el tipo .

También se cumple lo siguiente:

- No hay ninguna implicación de que otros subprocesos o alias no tengan acceso de escritura a `x` .
- No hay implicación que `SomeStruct` sea inmutable en virtud de `x` ser un `inref` archivo .

Sin embargo, para los tipos de `this` valor de F que `inref` **son** inmutables, se deduce que el puntero es un archivo .

Todas estas reglas juntas significan que `inref` el titular de un puntero no puede modificar el contenido inmediato de la memoria que se apunta.

### Semántica de Outref

El propósito `outref<'T>` de es indicar que el puntero sólo debe escribirse en. Inesperadamente, `outref<'T>` permite leer el valor subyacente a pesar de su nombre. Esto es para fines de compatibilidad. Semánticamente, `outref<'T>` no es `byref<'T>` diferente de .

### Interoperabilidad con C#

Además `in ref` de `out ref` las devoluciones, `ref` c. admite las palabras clave y, además de las devoluciones. En la tabla siguiente se muestra cómo f- interpreta lo que emite:

CONSTRUCCIÓN DE C-	INFERIDOS DE F
<code>ref</code> valor de retorno	<code>outref&lt;'T&gt;</code>
<code>ref readonly</code> valor de retorno	<code>inref&lt;'T&gt;</code>
Parámetro <code>in ref</code>	<code>inref&lt;'T&gt;</code>

CONSTRUCCIÓN DE C -	INFERIDOS DE F
Parámetro <code>out ref</code>	<code>outref&lt;'T&gt;</code>

En la tabla siguiente se muestra lo que emite F:

CONSTRUCCIÓN DE F -	CONSTRUCCIÓN EMITIDA
Argumento <code>inref&lt;'T&gt;</code>	<code>[In]</code> atributo en el argumento
<code>inref&lt;'T&gt;</code> devolución	<code>modreq</code> atributo sobre el valor
<code>inref&lt;'T&gt;</code> en ranura abstracta o implementación	<code>modreq</code> sobre discusión o devolución
Argumento <code>outref&lt;'T&gt;</code>	<code>[Out]</code> atributo en el argumento

## Reglas de inferencia y sobrecarga de tipos

En `inref<'T>` los siguientes casos, el compilador de F - deduce un tipo:

1. Un parámetro o tipo de `ReadOnly` valor devuelto de .NET que tiene un atributo.
2. El `this` puntero en un tipo struct que no tiene campos mutables.
3. La dirección de una ubicación `inref<_>` de memoria derivada de otro puntero.

Cuando se toma `inref` una dirección implícita de un, `SomeType` se prefiere una sobrecarga con `inref<SomeType>` un argumento de tipo a una sobrecarga con un argumento de tipo . Por ejemplo:

```
type C() =
    static member M(x: System.DateTime) = x.AddDays(1.0)
    static member M(x: inref<System.DateTime>) = x.AddDays(2.0)
    static member M2(x: System.DateTime, y: int) = x.AddDays(1.0)
    static member M2(x: inref<System.DateTime>, y: int) = x.AddDays(2.0)

let res = System.DateTime.Now
let v = C.M(res)
let v2 = C.M2(res, 4)
```

En ambos casos, las `System.DateTime` sobrecargas que toman `inref<System.DateTime>` se resuelven en lugar de las sobrecargas que toman .

## Estructuras similares a referencias externas

Además `byref` / `inref` / `outref` del trío, puede definir sus propias estructuras que se adhieran a `byref` la semántica -como. Esto se hace [IsByRefLikeAttribute](#) con el atributo:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` no implica `Struct` . Ambos deben estar presentes en el tipo.

Una `byref` estructura " -like" en F es un tipo de valor enlazado a la pila. Nunca se asigna en el montón

administrado. Una `byref` estructura -like es útil para la programación de alto rendimiento, ya que se aplica con un conjunto de comprobaciones sólidas sobre la duración y la no captura. Las reglas son:

- Se pueden utilizar como parámetros de función, parámetros de método, variables locales, retornos de método.
- No pueden ser miembros estáticos o de instancia de una clase o estructura normal.
- No se pueden capturar mediante `async` ninguna construcción de cierre (métodos o expresiones lambda).
- No se pueden utilizar como parámetro genérico.

Este último punto es crucial para la `|>` programación de estilo de canalización de F, al igual que una función genérica que parametriza sus tipos de entrada. Esta restricción puede `|>` ser relajada para el futuro, ya que está en línea y no hace ninguna llamada a funciones genéricas no insertadas en su cuerpo.

Aunque estas reglas restringen fuertemente el uso, lo hacen para cumplir la promesa de la informática de alto rendimiento de una manera segura.

## Devoluciones `byref`

Las devoluciones `Byref` de las funciones o miembros de F. se pueden producir y consumir. Al consumir `byref` un método -returning, el valor se desreferencia implícitamente. Por ejemplo:

```
let squareAndPrint (data : byref<int>) =  
    let squared = data*data    // data is implicitly dereferenced  
    printfn "%d" squared
```

Para devolver un valor `byref`, la variable que contiene el valor debe vivir más tiempo que el ámbito actual. Además, para devolver `&value` `byref`, use (donde `value` es una variable que dura más que el ámbito actual).

```
let mutable sum = 0  
let safeSum (bytes: Span<byte>) =  
    for i in 0 .. bytes.Length - 1 do  
        sum <- sum + int bytes.[i]  
    &sum // sum lives longer than the scope of this function.
```

Para evitar la desreferencia implícita, como pasar una `&x` referencia `x` a través de varias llamadas encadenadas, use (donde está el valor).

También puede asignar directamente `byref` a una devolución. Considere el siguiente programa (altamente imperativo):

```

type C() =
    let mutable nums = [| 1; 3; 7; 15; 31; 63; 127; 255; 511; 1023 |]

    override _.ToString() = String.Join(' ', nums)

    member _.FindLargestSmallerThan(target: int) =
        let mutable ctr = nums.Length - 1

        while ctr > 0 && nums.[ctr] >= target do ctr <- ctr - 1

        if ctr > 0 then &nums.[ctr] else &nums.[0]

[<EntryPoint>]
let main argv =
    let c = C()
    printfn "Original sequence: %s" (c.ToString())

    let v = &c.FindLargestSmallerThan 16

    v <- v*2 // Directly assign to the byref return

    printfn "New sequence:      %s" (c.ToString())

    0 // return an integer exit code

```

Éste es el resultado:

```

Original sequence: 1 3 7 15 31 63 127 255 511 1023
New sequence:      1 3 7 30 31 63 127 255 511 1023

```

## Scoping para byrefs

Un `let` valor enlazado no puede hacer que su referencia supere el ámbito en el que se definió. Por ejemplo, no se permite lo siguiente:

```

let test2 () =
    let x = 12
    &x // Error: 'x' exceeds its defined scope!

let test () =
    let x =
        let y = 1
        &y // Error: `y` exceeds its defined scope!
    ()

```

Esto le impide obtener resultados diferentes dependiendo de si compila con optimizaciones o no.

# Celdas de referencia

23/10/2019 • 4 minutes to read • [Edit Online](#)

*Las celdas de referencia* son ubicaciones de almacenamiento que permiten crear valores mutables con semántica de referencia.

## Sintaxis

```
ref expression
```

## Comentarios

Se utiliza el operador `ref` antes de un valor para crear una nueva celda de referencia que encapsula el valor. A continuación, se puede cambiar el valor subyacente, porque es mutable.

Una celda de referencia contiene un valor real; no una mera dirección. Al crear una celda de referencia mediante el operador `ref`, se crea una copia del valor subyacente como valor mutable encapsulado.

Se puede desreferenciar una celda de referencia mediante el operador `!` (bang).

En el ejemplo de código siguiente se muestran la declaración y el uso de celdas de referencia.

```
// Declare a reference.
let refVar = ref 6

// Change the value referred to by the reference.
refVar := 50

// Dereference by using the ! operator.
printfn "%d" !refVar
```

El resultado es `50`

Las celdas de referencia son instancias del tipo de registro genérico `Ref`, que se declara como sigue.

```
type Ref<'a> =
{ mutable contents: 'a }
```

El tipo `'a ref` es un sinónimo de `Ref<'a>`. Tanto el compilador como IntelliSense en el IDE, muestran el primero para este tipo, pero la definición subyacente es el segundo.

El operador `ref` crea una nueva celda de referencia. El código siguiente es la declaración del operador `ref`.

```
let ref x = { contents = x }
```

En la tabla siguiente se muestran las características que están disponibles en la celda de referencia.

OPERADOR, MIEMBRO O CAMPO	DESCRIPCIÓN	TIPO	DEFINICIÓN
<code>!</code> (operador de desreferencia)	Devuelve el valor subyacente.	<code>'a ref -&gt; 'a</code>	<code>let (!) r = r.contents</code>
<code>:=</code> (operador de asignación)	Cambia el valor subyacente.	<code>'a ref -&gt; 'a -&gt; unit</code>	<code>let (:=) r x = r.contents &lt;- x</code>
<code>ref</code> (operador)	Encapsula un valor en una nueva celda de referencia.	<code>'a -&gt; 'a ref</code>	<code>let ref x = { contents = x }</code>
<code>Value</code> propiedad	Obtiene o establece el valor subyacente.	<code>unit -&gt; 'a</code>	<code>member x.Value = x.contents</code>
<code>contents</code> (campo de registro)	Obtiene o establece el valor subyacente.	<code>'a</code>	<code>let ref x = { contents = x }</code>

Hay varias maneras de tener acceso al valor subyacente. El valor devuelto por el operador de desreferencia (`!`) no es un valor asignable. Por consiguiente, si se va a modificar el valor subyacente, se debe utilizar el operador de asignación (`:=`) en su lugar.

Tanto la propiedad `Value` como el campo `contents` son valores asignables. Así pues, se pueden utilizar para obtener acceso al valor subyacente o cambiarlo, como se muestra en el código siguiente.

```
let xRef : int ref = ref 10

printfn "%d" (xRef.Value)
printfn "%d" (xRef.contents)

xRef.Value <- 11
printfn "%d" (xRef.Value)
xRef.contents <- 12
printfn "%d" (xRef.contents)
```

La salida es la siguiente.

```
10
10
11
12
```

El campo `contents` se proporciona por motivos de compatibilidad con otras versiones de ML y generará una advertencia durante la compilación. Para deshabilitar la advertencia, utilice la opción `--mlcompatibility` del compilador. Para obtener más información, consulte [Opciones del compilador](#).

C#los programadores deben saber `ref` que C# en no es lo mismo que `ref` en F#. Las construcciones equivalentes en F# son [byrefs](#), que son un concepto diferente de las celdas de referencia.

Los valores marcados como `mutable` se pueden promover automáticamente a `'a ref` si los captura un cierre; vea [valores](#).

## Vea también

- [Referencia del lenguaje F#](#)
- [Parámetros y argumentos](#)

- [Referencia de símbolos y operadores](#)
- [Valores](#)



# Directivas de compilador

23/10/2019 • 8 minutes to read • [Edit Online](#)

En este tema se describen las directivas de procesador y las directivas de compilador.

## Directivas de preprocesador

Una directiva de preprocesador tiene como prefijo el símbolo # y aparece en una línea independiente. La interpreta el preprocesador, que se ejecuta antes que el compilador.

En la siguiente tabla se recoge una lista de las directivas de preprocesador disponibles en F#.

DIRECTIVA	DESCRIPCIÓN
<code>#if</code> <i>símbolo de</i>	Admite la compilación condicional. El código de la sección después <code>#if</code> de se incluye si se define el <i>símbolo</i> . El símbolo también se puede negar con <code>!</code> .
<code>#else</code>	Admite la compilación condicional. Marca una sección de código que incluir si el símbolo usado con la directiva <code>#if</code> anterior no se ha definido.
<code>#endif</code>	Admite la compilación condicional. Marca el final de una sección condicional de código.
<code>#</code> Ondula <i>int</i> , <code>#</code> Ondula valor <i>int cadena</i> , <code>#</code> Ondula valor <i>int cadena textual</i>	Indica el nombre de archivo y la línea de código fuente original para la depuración. Esta característica se proporciona para las herramientas que generan código fuente de F#.
<code>#nowarn</code> <i>warningcode</i>	Deshabilita una o varias advertencias del compilador. Para deshabilitar una advertencia, encuentre su número correspondiente en los resultados del compilador e inclúyalo entre comillas. Omita el prefijo "FS". Para deshabilitar varios números de advertencia en la misma línea, incluya cada número entre comillas y separe cada cadena con un espacio. Por ejemplo:

```
#nowarn "9" "40"
```

El efecto de deshabilitar una advertencia se aplica a todo el archivo, incluidas las partes del archivo que preceden a la Directiva.

## Directivas de compilación condicional

El código que está desactivado por una de estas directivas aparece atenuado en el editor de Visual Studio Code.

### NOTE

El comportamiento de las directivas de compilación condicional no es el mismo que en otros idiomas. Así, no se pueden usar expresiones booleanas con símbolos, mientras que `true` y `false` no tienen ningún significado especial. Los símbolos que se usan en la directiva `if` se tienen que definir con la línea de comandos o en la configuración del proyecto; no hay ninguna directiva de preprocesador `define`.

El siguiente código muestra el uso de las directivas `#if`, `#else` y `#endif`. En este ejemplo, el código contiene dos versiones de la definición de `function1`. Cuando `VERSION1` se define mediante la [opción-define](#) del compilador, se activa `#if` el código entre `#else` la Directiva y la Directiva. De lo contrario, se activará el código entre `#else` y `#endif`.

```
#if VERSION1
let function1 x y =
    printfn "x: %d y: %d" x y
    x + 2 * y
#else
let function1 x y =
    printfn "x: %d y: %d" x y
    x - 2*y
#endif

let result = function1 10 20
```

No hay ninguna directiva de preprocesador `#define` en F#. Hay que usar la configuración del proyecto o la opción del compilador pertinente para definir los símbolos que la directiva `#if` usa.

Las directivas de compilación condicional se pueden anidar. La sangría no es significativa en las directivas de preprocesador.

También puede negar un símbolo con `!`. En este ejemplo, el valor de una cadena es algo que solo se produce cuando *no* se depura:

```
#if !DEBUG
let str = "Not debugging!"
#else
let str = "Debugging!"
#endif
```

## Directivas de línea

Al compilar, el compilador informa de los posibles errores en el código de F# haciendo referencia a los números de línea en los que cada error se produce. Estos números de línea empiezan por 1 en la primera línea en un archivo. Pero si genera código fuente de F# con otra herramienta, los números de línea en el código generado no suelen ser de interés, ya que lo más probable es que los errores en el código de F# generado provengan de otro código fuente. La directiva `#line` constituye una forma de que los creadores de herramientas que generan código fuente de F# pasen información sobre los archivos de origen y los números de línea originales al código de F# generado.

Cuando se usa la directiva `#line`, es necesario que los nombres de archivo estén entre comillas. A menos que el token textual (`@`) aparezca delante de la cadena, hay que anteponer caracteres de barra diagonal inversa usando dos caracteres de barra diagonal inversa en lugar de uno, ya que así se podrán usar en la ruta de acceso. Los siguientes tokens de línea son válidos. En estos ejemplos, se da por hecho que el archivo original `Script1` da como resultado un archivo de código de F# generado automáticamente cuando se ejecuta con una herramienta y, asimismo, que el código en la ubicación de estas directivas se genera a partir de determinados tokens en la línea 25 del archivo `Script1`.

```
# 25
#line 25
#line 25 "C:\\Projects\\MyProject\\MyProject\\Script1"
#line 25 @"C:\Projects\MyProject\MyProject\Script1"
# 25 @"C:\Projects\MyProject\MyProject\Script1"
```

Estos tokens indican que el código de F# generado en esta ubicación se deriva de algunas construcciones en la línea 25 en Script1, o cerca de ella.

## Directivas de compilador

Las directivas de compilador se parecen a las directivas de preprocesador porque llevan como prefijo un signo #, pero el preprocesador no las interpreta, sino que se dejan para que el compilador las interprete y actúe en consecuencia.

En la siguiente tabla se recoge una lista de las directivas de compilador disponibles en F#.

DIRECTIVA	DESCRIPCIÓN
<code>#light ["ON" "OFF"]</code>	Habilita o deshabilita la sintaxis ligera de cara a la compatibilidad con otras versiones de ML. La sintaxis ligera está habilitada de forma predeterminada. La sintaxis detallada siempre está habilitada. Por lo tanto, puede usar ambas sintaxis, la ligera y la detallada. La directiva <code>#light</code> es equivalente en sí misma a <code>#light "on"</code> . Si especifica <code>#light "off"</code> , tendrá que usar la sintaxis detallada en todas las construcciones del lenguaje. La sintaxis en la documentación de F# se muestra bajo la asunción de que se está usando la sintaxis ligera. Para obtener más información, vea <a href="#">Sintaxis detallada</a> .

Para las directivas de intérprete (FSI. exe), vea [programación F#interactiva con](#) .

## Vea también

- [Referencia del lenguaje F#](#)
- [Opciones del compilador](#)

# Opciones del compilador

23/07/2020 • 18 minutes to read • [Edit Online](#)

En este tema se describen las opciones de línea de comandos del compilador de F # fsc.exe.

El entorno de compilación también se puede controlar estableciendo las propiedades del proyecto. En el caso de los proyectos que tienen como destino .NET Core, la propiedad "otras marcas",

`<OtherFlags>...</OtherFlags>` en `.fsproj`, se usa para especificar opciones de línea de comandos adicionales.

## Opciones del compilador por orden alfabético

En la tabla siguiente se muestran las opciones del compilador ordenadas alfabéticamente. Algunas de las opciones del compilador de F # son similares a las opciones del compilador de C#. En ese caso, se proporciona un vínculo al tema Opciones del compilador de C#.

OPCIÓN DEL COMPILADOR	DESCRIPCIÓN
<code>-a filename.fs</code>	Genera una biblioteca a partir del archivo especificado. Esta opción es una forma abreviada de <code>--target:library filename.fs</code> .
<code>--baseaddress:address</code>	Especifica la dirección base preferida para cargar una DLL.  Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/baseaddress (C# opciones del Compilador)</a> .
<code>--codepage:id</code>	Especifica la página de códigos que se va a usar durante la compilación si la página requerida no es la página de códigos predeterminada actual del sistema.  Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/páginas de códigos (C# opciones del Compilador)</a> .
<code>--consolecolors</code>	Especifica que los errores y las advertencias usan texto codificado por colores en la consola.
<code>--crossoptimize[+ -]</code>	Habilita o deshabilita las optimizaciones entre módulos.
<code>--delaysign[+ -]</code>	Retrasa la firma del ensamblado usando solo la parte pública de la clave de nombre seguro.  Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/delaysign (C# opciones del Compilador)</a> .

OPCIÓN DEL COMPILADOR	DESCRIPCIÓN
<pre>--checked[+ -]</pre>	<p>Habilita o deshabilita la generación de comprobaciones de desbordamiento.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/checked (C# opciones del Compilador)</a>.</p>
<pre>--debug[+ -]</pre> <pre>-g[+ -]</pre> <pre>--debug:[full pdbonly]</pre> <pre>-g: [full pdbonly]</pre>	<p>Habilita o deshabilita la generación de información de depuración, o especifica el tipo de información de depuración que se va a generar. El valor predeterminado es <code>full</code>, que permite asociar a un programa en ejecución. Elija esta opción <code>pdbonly</code> para obtener información de depuración limitada almacenada en un archivo PDB (base de datos de programa).</p> <p>Equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/debug (C# opciones del Compilador)</a>.</p>
<pre>--define:symbol</pre> <pre>-d:symbol</pre>	<p>Define un símbolo para su uso en la compilación condicional.</p>
<pre>--deterministic[+ -]</pre>	<p>Genera un ensamblado determinista (incluido el GUID y la marca de tiempo de la versión del módulo). Esta opción no se puede usar con números de versión de caracteres comodín y solo admite tipos de depuración integrados y portátiles</p>
<pre>--doc:xml doc-filename</pre>	<p>Indica al compilador que genere comentarios de documentación XML en el archivo especificado. Para obtener más información, consulta <a href="#">XML Documentation</a>.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/document (C# opciones del Compilador)</a>.</p>
<pre>--fullpaths</pre>	<p>Indica al compilador que genere rutas de acceso completas.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/fullpaths (C# opciones del Compilador)</a>.</p>
<pre>--help</pre> <pre>-?</pre>	<p>Muestra información de uso, incluida una breve descripción de todas las opciones del compilador.</p>

OPCIÓN DEL COMPILADOR	DESCRIPCIÓN
<code>--highentropyva[+ -]</code>	Habilitar o deshabilitar la selección aleatoria del diseño del espacio de direcciones (ASLR) de alta entropía, una característica de seguridad mejorada. El sistema operativo aleatoriza las ubicaciones de la memoria donde se carga la infraestructura de las aplicaciones (como la pila y el montón). Si habilita esta opción, los sistemas operativos pueden usar esta aleatorización para usar el espacio de direcciones de 64 bits completo en un equipo de 64 bits.
<code>--keycontainer:key-container-name</code>	Especifica un contenedor de claves de nombre seguro.
<code>--keyfile:filename</code>	Especifica el nombre de un archivo de clave pública para firmar el ensamblado generado.
<code>--lib:folder-name</code> <code>-I:folder-name</code>	<p>Especifica el directorio en el que se van a buscar los ensamblados a los que se hace referencia.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/lib (C# opciones del Compilador)</a>.</p>
<code>--linkresource:resource-info</code>	<p>Vincula un recurso especificado al ensamblado. El formato de Resource-info es <code>filename[name[public private]]</code></p> <p>La vinculación de un recurso único con esta opción es una alternativa a la incrustación de un archivo de recursos completo con la <code>--resource</code> opción.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/linkresource ((C# opciones del Compilador)</a>.</p>
<code>--mlcompatibility</code>	Omite las advertencias que aparecen cuando se usan características diseñadas para la compatibilidad con otras versiones de ML.
<code>--noframework</code>	Deshabilita la referencia predeterminada al ensamblado .NET Framework.
<code>--nointerfacedata</code>	Indica al compilador que omita el recurso que normalmente agrega a un ensamblado que incluye metadatos específicos de F #.
<code>--nologo</code>	No muestra el texto del banner al iniciar el compilador.
<code>--nooptimizationdata</code>	Indica al compilador que solo incluya la optimización esencial para implementar construcciones insertadas. Impide la inclusión entre módulos, pero mejora la compatibilidad binaria.
<code>--nowin32manifest</code>	Indica al compilador que omita el manifiesto de Win32 predeterminado.

OPCIÓN DEL COMPILADOR	DESCRIPCIÓN
<pre>--nowarn:warning-number-list</pre>	<p>Deshabilita las advertencias específicas enumeradas por número. Separe cada número de advertencia con una coma. Puede detectar el número de advertencia de cualquier advertencia en la salida de compilación.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/nowarn (C# opciones del Compilador)</a>.</p>
<pre>--optimize[+ -][optimization-option-list]</pre> <pre>-O[+ -] [optimization-option-list]</pre>	<p>Habilita o deshabilita las optimizaciones. Algunas opciones de optimización se pueden deshabilitar o habilitar de forma selectiva si se enumeran. Estos son:</p> <pre>nojitoptimize , nojittracking , noloaloptimize , nocrossoptimize , notailcalls .</pre>
<pre>--out:output-filename</pre> <pre>-o:output-filename</pre>	<p>Especifica el nombre del ensamblado o módulo compilado.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/out (C# opciones del Compilador)</a>.</p>
<pre>--pathmap:path=sourcePath,...</pre>	<p>Especifica cómo asignar rutas físicas a nombres de ruta de origen generados por el compilador.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/pathmap (C# opciones del Compilador)</a>.</p>
<pre>--pdb:pdb-filename</pre>	<p>Asigna un nombre al archivo PDB de depuración de salida (base de datos de programa). Esta opción solo se aplica cuando <code>--debug</code> también está habilitada.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/pdb (C# opciones del Compilador)</a>.</p>
<pre>--platform:platform-name</pre>	<p>Especifica que el código generado solo se ejecutará en la plataforma especificada ( <code>x86</code> , <code>Itanium</code> o <code>x64</code> ), o bien, si se elige el nombre de la plataforma <code>anycpu</code> , especifica que el código generado puede ejecutarse en cualquier plataforma.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/platform (C# opciones del Compilador)</a>.</p>
<pre>--preferreduilang:lang</pre>	<p>Especifica el nombre de la referencia cultural del lenguaje de salida preferido (por ejemplo, <code>es-ES</code> , <code>ja-JP</code> ).</p>

OPCIÓN DEL COMPILADOR	DESCRIPCIÓN
<code>--quotations-debug</code>	Especifica que debe emitirse información de depuración adicional para las expresiones que se derivan de los literales de Comillas de F # y las definiciones reflejadas. La información de depuración se agrega a los atributos personalizados de un nodo de árbol de expresión de F #. Vea <a href="#">expresiones de código delimitadas</a> y <a href="#">expr.CustomAttributes</a> .
<code>--reference:assembly-filename</code> <code>-r:assembly-filename</code>	<p>Hace que el código de un ensamblado de F # o .NET Framework esté disponible para el código que se está compilando.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">referencia de/(opciones del compilador de C#)</a>.</p>
<code>--resource:resource-filename</code>	<p>Incrusta un archivo de recursos administrado en el ensamblado generado.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/resource (C# opciones del Compilador)</a>.</p>
<code>--sig:signature-filename</code>	Genera un archivo de signature basado en el ensamblado generado. Para obtener más información acerca de los archivos de firma, consulte <a href="#">firmas</a> .
<code>--simpleresolution</code>	Especifica que las referencias de ensamblado deben resolverse mediante reglas mono basadas en directorios en lugar de la resolución de MSBuild. El valor predeterminado es usar la resolución de MSBuild excepto cuando se ejecuta en mono.
<code>--standalone</code>	Especifica que se genere un ensamblado que contenga todas sus dependencias para que se ejecute por sí solo sin necesidad de ensamblados adicionales, como la biblioteca de F #.
<code>--staticlink:assembly-name</code>	Vincula estáticamente el ensamblado especificado y todos los archivos DLL a los que se hace referencia que dependen de este ensamblado. Use el nombre de ensamblado, no el nombre del archivo DLL.
<code>--subsystemversion</code>	Especifica la versión del subsistema del sistema operativo que va a usar el ejecutable generado. Use 6,02 para Windows 8.1, 6,01 para Windows 7, 6,00 para Windows Vista. Esta opción solo se aplica a los ejecutables, no a las dll y solo se debe usar si la aplicación depende de características de seguridad específicas disponibles solo en determinadas versiones del sistema operativo. Si se usa esta opción y un usuario intenta ejecutar la aplicación en una versión anterior del sistema operativo, se producirá un mensaje de error.



OPCIÓN DEL COMPILADOR	DESCRIPCIÓN
<code>--tailcalls[+ -]</code>	<p>Habilita o deshabilita el uso de la instrucción IL de cola, que hace que se reutilice el marco de pila para las funciones recursivas de cola. Esta opción está habilitada de forma predeterminada.</p>
<code>--target:[exe winexe library module] filename</code>	<p>Especifica el tipo y el nombre de archivo del código compilado generado.</p> <ul style="list-style-type: none"> <li>• <code>exe</code> significa una aplicación de consola.</li> <li>• <code>winexe</code> significa una aplicación Windows, que difiere de la aplicación de consola en que no tiene definidos los flujos de entrada/salida estándar (stdin, stdout y stderr).</li> <li>• <code>library</code> es un ensamblado sin un punto de entrada.</li> <li>• <code>module</code> es un módulo de .NET Framework (.netmodule) que se puede combinar posteriormente con otros módulos en un ensamblado.</li> </ul> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/destino (C# opciones del Compilador)</a>.</p>
<code>--times</code>	<p>Muestra información de tiempo para la compilación.</p>
<code>--utf8output</code>	<p>Habilita la impresión del resultado del compilador en la codificación UTF-8.</p>
<code>--warn:warning-level</code>	<p>Establece un nivel de advertencia (de 0 a 5). El nivel predeterminado es 3. Cada ADVERTENCIA recibe un nivel en función de su gravedad. El nivel 5 proporciona advertencias más, pero menos graves que el nivel 1.</p> <p>Las advertencias de nivel 5 son: 21 (uso recursivo comprobado en tiempo de ejecución), 22 (se <code>let rec</code> evalúa desordenado), 45 (abstracción completa) y 52 (copia defensiva). Todas las demás advertencias son de nivel 2.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/warn (C# opciones del Compilador)</a>.</p>
<code>--warnon:warning-number-list</code>	<p>Habilitar advertencias específicas que podrían estar desactivadas de forma predeterminada o deshabilitadas por otra opción de la línea de comandos. En F # 3,0, solo la advertencia 1182 (variables no utilizadas) está desactivada de forma predeterminada.</p>

OPCIÓN DEL COMPILADOR	DESCRIPCIÓN
<code>--warnaserror[+ -] [warning-number-list]</code>	<p>Habilita o deshabilita la opción para notificar las advertencias como errores. Puede proporcionar los números de advertencia específicos que se van a deshabilitar o habilitar. Las opciones más adelante en la línea de comandos invalidan las opciones anteriores en la línea de comandos. Por ejemplo, para especificar las advertencias que no desea que se notifiquen como errores, especifique <code>--warnaserror+</code> <code>--warnaserror-:warning-number-list</code> .</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/warnaserror (C# opciones del Compilador)</a>.</p>
<code>--win32manifest:manifest-filename</code>	<p>Agrega un archivo de manifiesto Win32 a la compilación. Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/win32manifest (C# opciones del Compilador)</a>.</p>
<code>--win32res:resource-filename</code>	<p>Agrega un archivo de recursos de Win32 a la compilación.</p> <p>Esta opción del compilador es equivalente a la opción del compilador de C# con el mismo nombre. Para obtener más información, vea <a href="#">/opciones del compilador Win32 ((C#))</a>.</p>

## Artículos relacionados

TITLE	DESCRIPCIÓN
<a href="#">Opciones de F# Interactive</a>	Describe las opciones de línea de comandos admitidas por el intérprete de F # fsi.exe.
<a href="#">Referencia de propiedades del proyecto</a>	Describe la interfaz de usuario para los proyectos, incluidas las páginas de propiedades de proyecto que proporcionan opciones de compilación.

# Opciones de F# Interactive

08/01/2020 • 8 minutes to read • [Edit Online](#)

## NOTE

En este artículo actualmente solo se describe la experiencia para Windows. Se reescribirá.

En este tema se describen las opciones de línea de F# comandos compatibles con el `fsi.exe` interactivo. F#Interactive acepta muchas de las mismas opciones de línea de F# comandos que el compilador, pero también acepta algunas opciones adicionales.

## Uso F# de Interactive para scripting

F#Interactive, `fsi.exe`, puede iniciarse de forma interactiva, o bien se puede iniciar desde la línea de comandos para ejecutar un script. La sintaxis de la línea de comandos es

```
> fsi.exe [options] [ script-file [arguments] ]
```

La extensión de archivo F# de los archivos de script es `.fsx`.

## Tabla de F# opciones interactivas

En la tabla siguiente se resumen las opciones admitidas por F# el Interactive. Puede establecer estas opciones en la línea de comandos o a través del IDE de Visual Studio. Para establecer estas opciones en el IDE de Visual Studio, abra el menú **herramientas**, seleccione **Opciones...**, expanda el **F# nodo herramientas** y seleccione **F# interactivo**.

Cuando las listas aparecen F# en argumentos de opción interactiva, los elementos de lista se separan mediante signos de punto y coma ( `;` ).

OPCIÓN	DESCRIPCIÓN
--	Se usa para F# indicar a Interactive que trate los argumentos restantes como argumentos F# de la línea de comandos para el programa o script, al que puede tener acceso en el código mediante la lista FSI. <b>CommandLineArgs</b> .
--checked[ +   - ]	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--codepage:<int>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--consolecolors[ +   - ]	Genera mensajes de advertencia y error en color.
--crossoptimize[ +   - ]	Habilitar o deshabilitar las optimizaciones entre módulos.

OPCIÓN	DESCRIPCIÓN
--debug[ +   - ] --debug: [full pdbonly portable embedded] -g[ +   - ] -g: [full pdbonly portable embedded]	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--define:<string>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--deterministic[ +   - ]	Genera un ensamblado determinista (incluido el GUID y la marca de tiempo de la versión del módulo).
--exec	Indica a F# interactivo que salga después de cargar los archivos o de ejecutar el archivo de script proporcionado en la línea de comandos.
--fullpaths	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--gui[ +   - ]	Habilita o deshabilita el bucle de eventos Windows Forms. El valor predeterminado está habilitado.
--help -?	Se usa para mostrar la sintaxis de la línea de comandos y una breve descripción de cada opción.
--lib:<folder-list> -l:<folder-list>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--load:<filename>	Compila el código fuente especificado en el inicio y carga las construcciones compiladas F# en la sesión. Si el origen de destino contiene directivas de scripting como <b>#use</b> o <b>#load</b> , debe usar <b>--use</b> o <b>#use</b> en lugar de <b>--Load</b> o <b>#load</b> .
--mlcompatibility	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--noframework	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, vea <a href="#">Opciones del compilador</a> .
--nologo	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--nowarn:<warning-list>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--Optimize[ +   - ]	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
--preferreduilang:<lang>	Especifica el nombre de la referencia cultural del lenguaje de salida preferido (por ejemplo, es-ES, ja-JP).

OPCIÓN	DESCRIPCIÓN
<code>--quiet</code>	Suprime F# la salida de Interactive en el flujo <b>stdout</b> .
<code>--quotations-debug</code>	Especifica que debe emitirse información de depuración adicional para las expresiones que F# se derivan de los literales de Comillas y de las definiciones reflejadas. La información de depuración se agrega a los atributos F# personalizados de un nodo de árbol de expresión. Vea <a href="#">expresiones de código delimitadas</a> y <a href="#">expr. CustomAttributes</a> .
<code>--readline[ +   - ]</code>	Habilitar o deshabilitar la finalización con tabulación en el modo interactivo.
<code>--Reference:&lt;nombre de archivo&gt;</code> <code>-r:&lt;filename&gt;</code>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
<code>--shadowcopyreferences[ +   - ]</code>	Impide que el F# proceso interactivo bloquee las referencias.
<code>--simpleresolution</code>	Resuelve las referencias de ensamblado mediante reglas basadas en directorios en lugar de la resolución de MSBuild.
<code>--tailcalls[ +   - ]</code>	Habilitar o deshabilitar el uso de la instrucción IL de cola, que hace que se reutilice el marco de pila para las funciones recursivas de cola. Esta opción está habilitada de forma predeterminada.
<code>--targetprofile:&lt;cadena&gt;</code>	Especifica el perfil de la plataforma de destino de este ensamblado. Los valores válidos son mscorlib, netcore o netstandard. El valor predeterminado es mscorlib.
<code>--Use:&lt;nombre de archivo&gt;</code>	Indica al intérprete que use el archivo especificado al iniciar como entrada inicial.
<code>--utf8output</code>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
<code>--warn:&lt;warning-level&gt;</code>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
<code>--warnaserror[ +   - ]</code>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .
<code>--warnaserror[ +   - ]: &lt;int-list&gt;</code>	Igual que la opción del compilador FSC. <b>exe</b> . Para obtener más información, consulte <a href="#">Opciones del compilador</a> .

## Temas relacionados

TITLE	DESCRIPCIÓN
<a href="#">Opciones del compilador</a>	Describe las opciones de línea de comandos F# disponibles para el compilador, FSC. <b>exe</b> .

# Identificadores de línea, archivo y ruta de acceso de origen

23/10/2019 • 2 minutes to read • [Edit Online](#)

Los identificadores `__LINE__`, `__SOURCE_DIRECTORY__` y `__SOURCE_FILE__` son valores integrados que le permiten tener acceso al número de línea de código fuente, el directorio y el nombre de archivo en el código.

## Sintaxis

```
__LINE__  
__SOURCE_DIRECTORY__  
__SOURCE_FILE__
```

## Comentarios

Cada uno de estos valores tiene `string` el tipo.

En la tabla siguiente se resumen los identificadores de línea, archivo y ruta de acceso de origen F# que están disponibles en. Estos identificadores no son macros de preprocesador; son valores integrados que reconoce el compilador.

IDENTIFICADOR PREDEFINIDO	DESCRIPCIÓN
<code>__LINE__</code>	Se evalúa como el número de línea actual, <code>#line</code> teniendo en cuenta las directivas.
<code>__SOURCE_DIRECTORY__</code>	Se evalúa como la ruta de acceso completa actual del directorio de origen <code>#line</code> , teniendo en cuenta las directivas.
<code>__SOURCE_FILE__</code>	Se evalúa como el nombre del archivo de código fuente actual, sin su <code>#line</code> ruta de acceso, teniendo en cuenta las directivas.

Para obtener más información sobre `#line` la Directiva, vea [directivas de compilador](#).

## Ejemplo

En el ejemplo de código siguiente se muestra el uso de estos valores.

```
let printSourceLocation() =  
    printfn "Line: %s" __LINE__  
    printfn "Source Directory: %s" __SOURCE_DIRECTORY__  
    printfn "Source File: %s" __SOURCE_FILE__  
    printSourceLocation()
```

Resultado:

Line: 4

Source Directory: C:\Users\username\Documents\Visual Studio 2017\Projects\SourceInfo\SourceInfo

Source File: Program.fs

## Vea también

- [Directivas de compilador](#)
- [Referencia del lenguaje F#](#)

# Información del Llamador

25/11/2019 • 5 minutes to read • [Edit Online](#)

Mediante los atributos de información del llamador, se puede obtener información sobre el llamador de un método. Puede obtener la ruta de acceso al código fuente, el número de línea en el código fuente y el nombre de miembro del llamador. Esta información resulta útil para el seguimiento y la depuración, así como para crear herramientas de diagnóstico.

Para obtener esta información, se usan los atributos que se aplican a los parámetros opcionales, que tienen valores predeterminados. En la tabla siguiente se enumeran los atributos de información del llamador que se definen en el espacio de nombres [System.Runtime.CompilerServices](#) :

ATRIBUTO	DESCRIPCIÓN	TYPE
<a href="#">CallerFilePath</a>	Ruta de acceso completa del archivo de código fuente que contiene el llamador. Esta es la ruta de acceso en tiempo de compilación.	<code>String</code>
<a href="#">CallerLineNumber</a>	Número de línea en el archivo de código fuente en el que se llama al método.	<code>Integer</code>
<a href="#">CallerMemberName</a>	Método o nombre de propiedad del llamador. Vea la sección nombres de miembro más adelante en este tema.	<code>String</code>

## Ejemplo

En el ejemplo siguiente se muestra cómo puede utilizar estos atributos para realizar el seguimiento de un llamador.

```
open System.Diagnostics
open System.Runtime.CompilerServices
open System.Runtime.InteropServices

type Tracer() =
    member _.DoTrace(message: string,
        [<CallerMemberName; Optional; DefaultParameterValue("")>] memberName: string,
        [<CallerFilePath; Optional; DefaultParameterValue("")>] path: string,
        [<CallerLineNumber; Optional; DefaultParameterValue(0)>] line: int) =
        Trace.WriteLine(sprintf "Message: %s" message)
        Trace.WriteLine(sprintf "Member name: %s" memberName)
        Trace.WriteLine(sprintf "Source file path: %s" path)
        Trace.WriteLine(sprintf "Source line number: %d" line)
```

## Comentarios

Los atributos de información del llamador solo se pueden aplicar a los parámetros opcionales. Los atributos de información del llamador hacen que el compilador escriba el valor adecuado para cada parámetro opcional decorado con un atributo de información del llamador.

Los valores de información del llamador se emiten como literales en el lenguaje intermedio (IL) en tiempo de compilación. A diferencia de los resultados de la propiedad [StackTrace](#) para las excepciones, los resultados no se ven afectados por la ofuscación.



Puede proporcionar explícitamente los argumentos opcionales para controlar la información del llamador u ocultarla.

## Nombres de miembro

Puede usar el atributo `CallerMemberName` para evitar especificar el nombre de miembro como un argumento `String` para el método llamado. Mediante esta técnica, se evita el problema de que la refactorización de cambio de nombre no cambie los valores de `String`. Esta ventaja es especialmente útil para las siguientes tareas:

- Usar el seguimiento y las rutinas de diagnóstico.
- Implementar la interfaz `INotifyPropertyChanged` al enlazar datos. Esta interfaz permite que la propiedad de un objeto notifique a un control enlazado que la propiedad ha cambiado, de forma que el control pueda mostrar información actualizada. Sin el atributo `CallerMemberName`, debe especificar el nombre de la propiedad como un literal.

En el gráfico siguiente se muestran los nombres de miembro que se devuelven cuando se usa el atributo `CallerMemberName`.

LAS LLAMADAS SE PRODUCEN EN	RESULTADO DEL NOMBRE DE MIEMBRO
Método, propiedad o evento	Nombre del método, propiedad o evento en el que se originó la llamada.
Constructor	Cadena ".ctor"
Constructor estático	Cadena ".cctor"
Destructor	Cadena "Finalize"
Conversiones u operadores definidos por el usuario	Nombre generado para el miembro, por ejemplo, "op_Addition".
Constructor de atributo	Nombre del miembro al que se aplica el atributo. Si el atributo es cualquier elemento dentro de un miembro (como un parámetro, un valor devuelto o un parámetro de tipo genérico), el resultado es el nombre del miembro asociado a este elemento.
Ningún miembro contenedor (por ejemplo, nivel de ensamblado o atributos que se aplican a tipos)	El valor predeterminado del parámetro opcional.

## Vea también

- [Atributos](#)
- [Argumentos con nombre](#)
- [Parámetros opcionales](#)

# Sintaxis detallada

16/04/2020 • 3 minutes to read • [Edit Online](#)

Hay dos formas de sintaxis disponibles para muchas construcciones en el lenguaje F: *sintaxis detallada* y *sintaxis ligera*. La sintaxis detallada no es tan comúnmente utilizada, pero tiene la ventaja de ser menos sensible a la sangría. La sintaxis ligera es más corta y utiliza la sangría para indicar el `begin` `end` principio `in` y el final de las construcciones, en lugar de palabras clave adicionales como `,` `,` `,` etc. La sintaxis predeterminada es la sintaxis ligera. En este tema se describe la sintaxis para las construcciones de F- cuando la sintaxis ligera no está habilitada. La sintaxis detallada siempre está habilitada, por lo que incluso si habilita la sintaxis ligera, todavía puede usar sintaxis detallada para algunas construcciones. Puede deshabilitar la sintaxis `#light "off"` ligera mediante la directiva.

## Tabla de Construcciones

En la tabla siguiente se muestra la sintaxis ligera y detallada para las construcciones de lenguaje de F en contextos donde hay una diferencia entre los dos formularios. En esta tabla, los `<>` corchetes angulares ( ) encierran los elementos de sintaxis proporcionados por el usuario. Consulte la documentación de cada construcción de lenguaje para obtener información más detallada sobre la sintaxis utilizada en estas construcciones.

CONSTRUCCIÓN DEL LENGUAJE	SINTAXIS LIGERA	SINTAXIS DETALLADA
expresiones compuestas	<pre>&lt;expression1 /&gt; &lt;expression2 /&gt;</pre>	<pre>&lt;expression1&gt;; &lt;expression2&gt;</pre>
enlaces anidados <code>let</code>	<pre>let f x =   let a = 1   let b = 2   x + a + b</pre>	<pre>let f x =   let a = 1 in   let b = 2 in   x + a + b</pre>
bloque de código	<pre>(   &lt;expression1&gt;   &lt;expression2&gt; )</pre>	<pre>begin   &lt;expression1&gt;;   &lt;expression2&gt;; end</pre>
<code>`for...do`</code>	<pre>for counter = start to finish do   ...</pre>	<pre>for counter = start to finish do   ... done</pre>

`while...do`	<pre>while &lt;condition&gt; do   ...</pre>	<pre>while &lt;condition&gt; do   ... done</pre>
`for...in`	<pre>for var in start .. finish do   ...</pre>	<pre>for var in start .. finish do   ... done</pre>
`do`	<pre>do   ...</pre>	<pre>do   ... in</pre>
registro	<pre>type &lt;record-name&gt; = {   &lt;field-declarations&gt; } &lt;value-or-member- definitions&gt;</pre>	<pre>type &lt;record-name&gt; = {   &lt;field-declarations&gt; } with   &lt;value-or-member- definitions&gt; end</pre>
clase	<pre>type &lt;class-name&gt;(&lt;params&gt;) =   ...</pre>	<pre>type &lt;class-name&gt;(&lt;params&gt;) =   class     ...   end</pre>
structure	<pre>[&lt;StructAttribute&gt;] type &lt;structure-name&gt; =   ...</pre>	<pre>type &lt;structure-name&gt; =   struct     ...   end</pre>
unión discriminada	<pre>type &lt;union-name&gt; =   ...   ... ... &lt;value-or-member definitions&gt;</pre>	<pre>type &lt;union-name&gt; =   ...   ... ... with   &lt;value-or-member- definitions&gt; end</pre>

interfaz	<pre>type &lt;interface-name&gt; =   ...</pre>	<pre>type &lt;interface-name&gt; =   interface     ...   end</pre>
expresión de objeto	<pre>{ new &lt;type-name&gt;   with     &lt;value-or-member- definitions&gt;     &lt;interface- implementations&gt; }</pre>	<pre>{ new &lt;type-name&gt;   with     &lt;value-or-member- definitions&gt;   end   &lt;interface- implementations&gt; }</pre>
implementación de interfaces	<pre>interface &lt;interface-name&gt;   with     &lt;value-or-member- definitions&gt;</pre>	<pre>interface &lt;interface-name&gt;   with     &lt;value-or-member- definitions&gt;   end</pre>
tipo de extensión	<pre>type &lt;type-name&gt;   with     &lt;value-or-member- definitions&gt;</pre>	<pre>type &lt;type-name&gt;   with     &lt;value-or-member- definitions&gt;   end</pre>
module	<pre>module &lt;module-name&gt; =   ...</pre>	<pre>module &lt;module-name&gt; =   begin     ...   end</pre>

## Consulte también

- [Referencia del lenguaje f](#)
- [Directivas de compilador](#)
- [Instrucciones de formato de código](#)

# F#mensajes del compilador

01/02/2020 • 2 minutes to read • [Edit Online](#)

En esta sección se detallan los errores F# y advertencias del compilador que emitirá el compilador para ciertas construcciones. Los conjuntos de errores predeterminados se pueden cambiar:

- Tratar advertencias específicas como si fueran errores mediante la opción del compilador `-warnaserror+`,
- Omitir advertencias específicas mediante la opción del compilador `-nowarn`

Si aún no se ha registrado una advertencia o un error determinado en esta sección, vaya al final de esta página y envíe comentarios que incluyan el número o el texto del error.

## Vea también

- [F#Opciones del compilador](#)

# Guía de estilo de F#

19/03/2020 • 6 minutes to read • [Edit Online](#)

En los artículos siguientes se describen las directrices para dar formato al código de F y a las instrucciones temáticas para las características del lenguaje y cómo se deben usar.

Esta guía se ha formulado sobre la base del uso de F en grandes bases de código con un grupo diverso de programadores. Esta guía generalmente conduce al uso exitoso de F y minimiza las frustraciones cuando los requisitos para los programas cambian con el tiempo.

## Cinco principios del buen código de F

Tenga en cuenta los siguientes principios cada vez que escriba código de F, especialmente en sistemas que cambiarán con el tiempo. Cada pieza de orientación en otros artículos proviene de estos cinco puntos.

### 1. El buen código de F es sucinto, expresivo y componible

F- tiene muchas características que le permiten expresar acciones en menos líneas de código y reutilizar la funcionalidad genérica. La biblioteca principal de F - también contiene muchos tipos y funciones útiles para trabajar con colecciones comunes de datos. La composición de sus propias funciones y las de la biblioteca principal de F ( u otras bibliotecas) forma parte de la programación idiomática de F . Como regla general, si puede expresar una solución a un problema en menos líneas de código, otros desarrolladores (o su futuro yo) se apreciarán. También es muy recomendable que use una biblioteca como FSharp.Core, las [vastas bibliotecas](#) de .NET en las que se ejecuta F, o un paquete de terceros en [NuGet](#) cuando necesite realizar una tarea no trivial.

### 2. Un buen código de F es interoperable

La interoperación puede adoptar varios formularios, incluido el consumo de código en diferentes idiomas. Los límites del código con los que interoperan otros llamadores son elementos críticos para obtener la razón, incluso si los llamadores también están en F . Al escribir F, siempre debe estar pensando en cómo otro código llamará al código que está escribiendo, incluso si lo hacen desde otro idioma como C. Las directrices de [diseño de componentes](#) de FTM describen la interoperabilidad en detalle.

### 3. Un buen código de F hace uso de la programación de objetos, no de la orientación de objetos

F tiene soporte completo para la programación con objetos en .NET, [incluidas clases](#), [interfaces](#), modificadores de [acceso](#), [clases abstractas](#), etc. [classes](#) Para obtener código funcional más complicado, como funciones que deben tener en cuenta el contexto, los objetos pueden encapsular fácilmente información contextual de maneras que las funciones no pueden. Las características como los [parámetros opcionales](#) y el uso cuidadoso de [la sobrecarga](#) pueden facilitar el consumo de esta funcionalidad a los llamadores.

### 4. Un buen código De F funciona bien sin exponer la mutación

No es ningún secreto que para escribir código de alto rendimiento, debes usar la mutación. Después de todo, es cómo funcionan las computadoras. Este código es a menudo propenso a errores y difícil de hacer bien. Evite exponer la mutación a las personas que llaman. En su lugar, [cree una interfaz funcional que oculte una implementación basada en mutaciones](#) cuando el rendimiento sea crítico.

### 5. El buen código de F es útil

Las herramientas son invaluable para trabajar en grandes bases de código, y puede escribir código de F, de

tal manera que se pueda usar de forma más eficaz con las herramientas de lenguaje de F. Un ejemplo es asegurarse de no exagerar con un estilo de programación sin puntos, de modo que los valores intermedios se puedan inspeccionar con un depurador. Otro ejemplo es el uso de [comentarios](#) de documentación XML que describen construcciones de tal manera que la información sobre herramientas en los editores puede mostrar esos comentarios en el sitio de llamada. Piense siempre en cómo el código será leído, navegado, depurado y manipulado por otros programadores con sus herramientas.

## Pasos siguientes

Las [directrices](#) de formato de código de F - proporcionan instrucciones sobre cómo dar formato al código para que sea fácil de leer.

Las convenciones de [codificación](#) de F - proporcionan orientación para los modismos de programación de F - que ayudarán al mantenimiento a largo plazo de bases de código de F .

Las directrices de diseño de [componentes](#) de F , proporcionan instrucciones para la creación de componentes de F, como bibliotecas.

# Instrucciones de formato de código de F#

23/07/2020 • 31 minutes to read • [Edit Online](#)

En este artículo se proporcionan instrucciones sobre cómo dar formato al código para que el código de F # sea:

- Más legible
- De acuerdo con las convenciones aplicadas por las herramientas de formato en Visual Studio y otros editores
- Similar a otro código en línea

Estas instrucciones se basan en [una guía completa de las convenciones de formato de F #](#) por Anh-estírcol Phan.

## Reglas generales para la sangría

F # utiliza de forma predeterminada un espacio en blanco significativo. Las instrucciones siguientes están pensadas para proporcionar orientación sobre cómo llevar a cabo algunos desafíos que esto puede imponer.

### Usar espacios

Cuando se requiere la sangría, debe usar espacios, no pestañas. Se requiere al menos un espacio. Su organización puede crear estándares de codificación para especificar el número de espacios que se usarán para la sangría; dos, tres o cuatro espacios de sangría en cada nivel en que se produce la sangría es típico.

**Se recomiendan cuatro espacios por sangría.**

Dicho esto, la sangría de los programas es una cuestión subjetiva. Las variaciones son correctas, pero la primera regla que debe seguir es la *coherencia de la sangría*. Elija un estilo de sangría aceptado generalmente y úselo sistemáticamente en el código base.

## Aplicar formato a los espacios en blanco

F # es un espacio en blanco sensible. Aunque la mayoría de las semánticas de los espacios en blanco se describen mediante una sangría adecuada, hay otros aspectos que hay que tener en cuenta.

### Aplicar formato a operadores en expresiones aritméticas

Use siempre el espacio en blanco alrededor de las expresiones aritméticas binarias:

```
let subtractThenAdd x = x - 1 + 3
```

- Los operadores unarios siempre deben ir seguidos inmediatamente del valor que están negando:

```
// OK
let negate x = -x

// Bad
let negateBad x = - x
```

Agregar un carácter de espacio en blanco después del `-` operador puede provocar confusión en otros.

En Resumen, es importante que siempre:

- Operadores binarios de envolvente con espacio en blanco
- Nunca debe haber un espacio en blanco al final después de un operador unario



La instrucción del operador aritmético binario es especialmente importante. Si no se rodea un operador binario `-`, cuando se combina con determinadas opciones de formato, podría interpretarlo como unario `-`.

### Envolver una definición de operador personalizado con espacio en blanco

Use siempre el espacio en blanco para rodear una definición de operador:

```
// OK
let ( !> ) x f = f x

// Bad
let (!>) x f = f x
```

Para cualquier operador personalizado que empiece por `*` y que tenga más de un carácter, debe agregar un espacio en blanco al principio de la definición para evitar la ambigüedad del compilador. Por este motivo, se recomienda que solo incluya las definiciones de todos los operadores con un solo carácter de espacio en blanco.

### Flechas de parámetros de función envolvente con espacio en blanco

Al definir la firma de una función, use el espacio en blanco que rodea el `->` símbolo:

```
// OK
type MyFun = int -> int -> string

// Bad
type MyFunBad = int->int->string
```

### Rodear argumentos de función con espacio en blanco

Al definir una función, use el espacio en blanco alrededor de cada argumento.

```
// OK
let myFun (a: decimal) b c = a + b + c

// Bad
let myFunBad (a:decimal)(b)c = a + b + c
```

### Colocar parámetros en una nueva línea para definiciones de miembros largas

Si tiene una definición de miembro muy larga, coloque los parámetros en nuevas líneas y aplíqueles sangría para que coincidan con el nivel de sangría del parámetro subsiguiente.

```
type C() =
    member _.LongMethodWithLotsOfParameters(aVeryLongType: AVeryLongTypeThatYouNeedToUse,
                                              aSecondVeryLongType: AVeryLongTypeThatYouNeedToUse,
                                              aThirdVeryLongType: AVeryLongTypeThatYouNeedToUse) =
        // ... the body of the method follows
```

Esto también se aplica a los constructores:

```
type C(aVeryLongType: AVeryLongTypeThatYouNeedToUse,
      aSecondVeryLongType: AVeryLongTypeThatYouNeedToUse,
      aThirdVeryLongType: AVeryLongTypeThatYouNeedToUse) =
    // ... the body of the class follows
```

Si hay una anotación de tipo de valor devuelto explícita, puede estar al final de `)` y antes de `=`, o en una línea nueva. Si el tipo de valor devuelto también tiene un nombre largo, el último podría ser preferible:

```

type C() =
    member _.LongMethodWithLotsOfParameters(aVeryLongType: AVeryLongTypeThatYouNeedToUse,
                                              aSecondVeryLongType: AVeryLongTypeThatYouNeedToUse,
                                              aThirdVeryLongType: AVeryLongTypeThatYouNeedToUse)
                                              : AVeryLongReturnType =

    // ... the body of the method follows

```

## Anotaciones de tipo

### Anotaciones del tipo de argumento de la función del botón secundario

Al definir argumentos con anotaciones de tipo, use el espacio en blanco después del `:` símbolo:

```

// OK
let complexFunction (a: int) (b: int) c = a + b + c

// Bad
let complexFunctionBad (a :int) (b :int) (c:int) = a + b + c

```

### Anotaciones de tipo de valor devuelto envuelto con espacio en blanco

En una anotación de tipo de valor o función enlazada a (tipo de valor devuelto en el caso de una función), use el espacio en blanco antes y después del `:` símbolo:

```

// OK
let expensiveToCompute : int = 0 // Type annotation for let-bound value
let myFun (a: decimal) b c : decimal = a + b + c // Type annotation for the return type of a function
// Bad
let expensiveToComputeBad1:int = 1
let expensiveToComputeBad2 :int = 2
let myFunBad (a: decimal) b c:decimal = a + b + c

```

## Aplicar formato a líneas en blanco

- Separe las definiciones de clase y de función de nivel superior con dos líneas en blanco.
- Las definiciones de método dentro de una clase se separan mediante una sola línea en blanco.
- Se pueden usar líneas en blanco adicionales (moderadamente) para separar grupos de funciones relacionadas. Es posible que se omitan líneas en blanco entre un grupo de una sola línea relacionada (por ejemplo, un conjunto de implementaciones ficticias).
- Use líneas en blanco en las funciones, con moderación, para indicar secciones lógicas.

## Aplicar formato a comentarios

Normalmente, se prefieren varios comentarios de barra diagonal doble sobre los comentarios de bloque de estilo ML.

```

// Prefer this style of comments when you want
// to express written ideas on multiple lines.

(*
    ML-style comments are fine, but not a .NET-ism.
    They are useful when needing to modify multi-line comments, though.
*)

```

Los comentarios en línea deben poner en mayúscula la primera letra.

```
let f x = x + 1 // Increment by one.
```

## Convenciones de nomenclatura

### Usar camelCase para funciones y valores enlazados a expresiones y enlazados a un patrón

Es común y acepta el estilo F # para usar camelCase para todos los nombres enlazados como variables locales o en coincidencias de patrones y definiciones de función.

```
// OK
let addIAndJ i j = i + j

// Bad
let addIAndJ I J = I+J

// Bad
let AddIAndJ i j = i + j
```

Las funciones enlazadas localmente en clases también deben usar camelCase.

```
type MyClass() =

    let doSomething () =

        let firstResult = ...

        let secondResult = ...

    member x.Result = doSomething()
```

### Usar camelCase para funciones públicas enlazadas a módulos

Cuando una función enlazada a un módulo forma parte de una API pública, debe usar camelCase:

```
module MyAPI =
    let publicFunctionOne param1 param2 param2 = ...

    let publicFunctionTwo param1 param2 param3 = ...
```

### Usar camelCase para funciones y valores enlazados a módulos internos y privados

Use camelCase para valores enlazados a módulos privados, incluidos los siguientes:

- Funciones ad hoc en scripts
- Valores que conforman la implementación interna de un módulo o tipo

```
let emailMyBossTheLatestResults =
    ...
```

### Usar camelCase para los parámetros

Todos los parámetros deben usar camelCase de acuerdo con las convenciones de nomenclatura de .NET.

```

module MyModule =
    let myFunction paramOne paramTwo = ...

type MyClass() =
    member this.MyMethod(paramOne, paramTwo) = ...

```

## Uso de PascalCase para módulos

Todos los módulos (de nivel superior, interno, privado, anidado) deben usar PascalCase.

```

module MyTopLevelModule

module Helpers =
    module private SuperHelpers =
        ...

    ...

```

## Usar PascalCase para las declaraciones de tipos, los miembros y las etiquetas

Todas las clases, interfaces, estructuras, enumeraciones, delegados, registros y uniones discriminadas deben llamarse con PascalCase. Los miembros de tipos y etiquetas para registros y uniones discriminadas también deben usar PascalCase.

```

type IMyInterface =
    abstract Something: int

type MyClass() =
    member this.MyMethod(x, y) = x + y

type MyRecord = { IntVal: int; StringVal: string }

type SchoolPerson =
    | Professor
    | Student
    | Advisor
    | Administrator

```

## Uso de PascalCase para construcciones intrínsecas de .NET

Los espacios de nombres, excepciones, eventos y proyectos/ `.dll` nombres también deben usar PascalCase. Esto no solo hace que el consumo de otros lenguajes .NET parezca más natural a los consumidores, sino que también es coherente con las convenciones de nomenclatura de .NET que es probable que encuentre.

## Evitar el carácter de subrayado en los nombres

Históricamente, algunas bibliotecas de F # usaban guiones bajos en los nombres. Sin embargo, esto ya no se acepta ampliamente, en parte porque entra en conflicto con las convenciones de nomenclatura de .NET. Dicho esto, algunos programadores de F # usan subrayados en gran medida, en parte por motivos históricos, y la tolerancia y el respeto son importantes. Sin embargo, tenga en cuenta que el estilo suele estar deshabilitado por otros usuarios que tienen la opción de usarlos.

Una excepción incluye la interoperabilidad con componentes nativos, donde los guiones bajos son comunes.

## Usar operadores estándar de F #

Los operadores siguientes se definen en la biblioteca estándar de F # y deben usarse en lugar de definir equivalentes. Se recomienda usar estos operadores, ya que tiende a hacer que el código sea más legible y idiomático. Los desarrolladores con un fondo en OCaml u otro lenguaje de programación funcional pueden estar acostumbrados a diferentes expresiones. En la lista siguiente se resumen los operadores de F # recomendados.

```

x |> f // Forward pipeline
f >> g // Forward composition
x |> ignore // Discard away a value
x + y // Overloaded addition (including string concatenation)
x - y // Overloaded subtraction
x * y // Overloaded multiplication
x / y // Overloaded division
x % y // Overloaded modulus
x && y // Lazy/short-cut "and"
x || y // Lazy/short-cut "or"
x <<< y // Bitwise left shift
x >>> y // Bitwise right shift
x ||| y // Bitwise or, also for working with "flags" enumeration
x &&& y // Bitwise and, also for working with "flags" enumeration
x ^^^ y // Bitwise xor, also for working with "flags" enumeration

```

## Use la sintaxis de prefijo para genéricos ( `Foo<T>` ) en preferencia a la sintaxis de postfijo ( `T Foo` )

F # hereda el estilo de ambos postfijos de los tipos genéricos de nomenclatura (por ejemplo,), así `int list` como el estilo de .net de prefijo (por ejemplo, `list<int>` ). Prefiera el estilo .NET, excepto para cinco tipos específicos:

1. En el caso de las listas de F #, use el formato de postfijo: `int list` en lugar de `list<int>` .
2. En el caso de las opciones de F #, use el formato de postfijo: `int option` en lugar de `option<int>` .
3. Para las opciones de valor de F #, use el formato de postfijo: `int voption` en lugar de `voption<int>` .
4. En el caso de las matrices de F #, use el nombre sintáctico `int[]` en lugar de `int array` o `array<int>` .
5. En el caso de las celdas de referencia, use `int ref` en lugar de `ref<int>` o `Ref<int>` .

En el caso de todos los demás tipos, use el formato de prefijo.

## Dar formato a tuplas

Una creación de instancias de tupla debe ir entre paréntesis y las comas delimitadoras deben ir seguidas de un solo espacio, por ejemplo: `(1, 2)` , `(x, y, z)` .

Normalmente se acepta para omitir paréntesis en la coincidencia de patrones de tuplas:

```

let (x, y) = z // Destructuring
let x, y = z // OK

// OK
match x, y with
| 1, _ -> 0
| x, 1 -> 0
| x, y -> 1

```

También se acepta normalmente para omitir los paréntesis si la tupla es el valor devuelto de una función:

```

// OK
let update model msg =
    match msg with
    | 1 -> model + 1, []
    | _ -> model, [ msg ]

```

En Resumen, se prefieren las instancias de tupla entre paréntesis, pero cuando se usan tuplas para la coincidencia de patrones o un valor devuelto, se considera adecuado para evitar paréntesis.

## Dar formato a las declaraciones de Unión discriminadas

Aplicar sangría `|` en la definición de tipo en cuatro espacios:

```
// OK
type Volume =
    | Liter of float
    | FluidOunce of float
    | ImperialPint of float

// Not OK
type Volume =
| Liter of float
| USPint of float
| ImperialPint of float
```

## Aplicar formato a uniones discriminadas

Las uniones discriminadas con instancias que se dividen entre varias líneas deben proporcionar a los datos contenidos un nuevo ámbito con sangría:

```
let tree1 =
    BinaryNode
        (BinaryNode(BinaryValue 1, BinaryValue 2),
         BinaryNode(BinaryValue 3, BinaryValue 4))
```

El paréntesis de cierre también puede estar en una nueva línea:

```
let tree1 =
    BinaryNode(
        BinaryNode(BinaryValue 1, BinaryValue 2),
        BinaryNode(BinaryValue 3, BinaryValue 4)
    )
```

## Dar formato a declaraciones de registro

Aplicar sangría `{` en la definición de tipo en cuatro espacios e iniciar la lista de campos en la misma línea:

```
// OK
type PostalAddress =
    { Address: string
      City: string
      Zip: string }
member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

// Not OK
type PostalAddress =
{ Address: string
  City: string
  Zip: string }
member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

// Unusual in F#
type PostalAddress =
{
    Address: string
    City: string
    Zip: string
}
```

Es preferible colocar el token de apertura en una nueva línea y el token de cierre en una nueva línea si se declaran

implementaciones de interfaz o miembros en el registro:

```
// Declaring additional members on PostalAddress
type PostalAddress =
{
    Address: string
    City: string
    Zip: string
}
member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

type MyRecord =
{
    SomeField: int
}
interface IMyInterface
```

## Aplicar formato a registros

Los registros cortos se pueden escribir en una línea:

```
let point = { X = 1.0; Y = 0.0 }
```

Los registros que son más largas deben usar nuevas líneas para las etiquetas:

```
let rainbow =
{ Boss = "Jeffrey"
  Lackeys = ["Zippy"; "George"; "Bungle"] }
```

Es preferible colocar el token de apertura en una nueva línea, el contenido con pestañas en un ámbito y el token de cierre en una nueva línea si:

- Mover registros en el código con diferentes ámbitos de sangría
- Canalizarlos en una función

```

let rainbow =
{
    Boss1 = "Jeffrey"
    Boss2 = "Jeffrey"
    Boss3 = "Jeffrey"
    Boss4 = "Jeffrey"
    Boss5 = "Jeffrey"
    Boss6 = "Jeffrey"
    Boss7 = "Jeffrey"
    Boss8 = "Jeffrey"
    Lackeys = ["Zippy"; "George"; "Bungle"]
}

type MyRecord =
{
    SomeField: int
}
interface IMyInterface

let foo a =
a
|> Option.map (fun x ->
{
    MyField = x
})

```

Las mismas reglas se aplican a los elementos de lista y matriz.

## Dar formato a expresiones de registro de copia y actualización

Una expresión de registro de copia y actualización sigue siendo un registro, por lo que se aplican instrucciones similares.

Las expresiones cortas pueden caber en una línea:

```
let point2 = { point with X = 1; Y = 2 }
```

Las expresiones más largas deben usar nuevas líneas:

```

let rainbow2 =
{ rainbow with
    Boss = "Jeffrey"
    Lackeys = [ "Zippy"; "George"; "Bungle" ] }

```

Y, al igual que con la guía de registro, puede que desee dedicar líneas independientes a las llaves y aplicar sangría a un ámbito a la derecha con la expresión. En algunos casos especiales, como el ajuste de un valor con un opcional sin paréntesis, puede que tenga que mantener una llave en una línea:



```

type S = { F1: int; F2: string }
type State = { F: S option }

let state = { F = Some { F1 = 1; F2 = "Hello" } }
let newState =
  {
    state with
      F = Some {
        F1 = 0
        F2 = ""
      }
  }

```

## Aplicar formato a listas y matrices

Escribe `x :: 1` espacios alrededor del `::` operador ( `::` es un operador infijo, por lo que está rodeado por espacios).

La lista y las matrices declaradas en una sola línea deben tener un espacio después del corchete de apertura y antes del corchete de cierre:

```

let xs = [ 1; 2; 3 ]
let ys = [| 1; 2; 3; |]

```

Utilice siempre al menos un espacio entre dos operadores distintivos de tipo llave. Por ejemplo, deje un espacio entre `[` y `{`.

```

// OK
[ { IngredientName = "Green beans"; Quantity = 250 }
  { IngredientName = "Pine nuts"; Quantity = 250 }
  { IngredientName = "Feta cheese"; Quantity = 250 }
  { IngredientName = "Olive oil"; Quantity = 10 }
  { IngredientName = "Lemon"; Quantity = 1 } ]

// Not OK
[ { IngredientName = "Green beans"; Quantity = 250 }
  { IngredientName = "Pine nuts"; Quantity = 250 }
  { IngredientName = "Feta cheese"; Quantity = 250 }
  { IngredientName = "Olive oil"; Quantity = 10 }
  { IngredientName = "Lemon"; Quantity = 1 } ]

```

La misma instrucción se aplica a las listas o matrices de tuplas.

Las listas y matrices que se dividen entre varias líneas siguen una regla similar a la de los registros:

```

let pascalsTriangle =
  [|
    [ 1 ]
    [ 1; 1 ]
    [ 1; 2; 1 ]
    [ 1; 3; 3; 1 ]
    [ 1; 4; 6; 4; 1 ]
    [ 1; 5; 10; 10; 5; 1 ]
    [ 1; 6; 15; 20; 15; 6; 1 ]
    [ 1; 7; 21; 35; 35; 21; 7; 1 ]
    [ 1; 8; 28; 56; 70; 56; 28; 8; 1 ]
  |]

```

Y como con los registros, la declaración de los corchetes de apertura y cierre en su propia línea hará que el código

y la canalización en funciones sean más fáciles.

Al generar matrices y listas mediante programación, `->` es preferible `do ... yield` cuando se genera siempre un valor:

```
// Preferred
let squares = [ for x in 1..10 -> x * x ]

// Not preferred
let squares' = [ for x in 1..10 do yield x * x ]
```

Las versiones anteriores del lenguaje F # requerían especificar en situaciones en las que los `yield` datos se pueden generar de forma condicional, o puede haber expresiones consecutivas que se van a evaluar. Prefiera omitir estas `yield` palabras clave a menos que deba compilar con una versión anterior del lenguaje F #:

```
// Preferred
let daysOfWeek includeWeekend =
    [
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    ]

// Not preferred
let daysOfWeek' includeWeekend =
    [
        yield "Monday"
        yield "Tuesday"
        yield "Wednesday"
        yield "Thursday"
        yield "Friday"
        if includeWeekend then
            yield "Saturday"
            yield "Sunday"
    ]
```

En algunos casos, `do...yield` puede ayudar a mejorar la legibilidad. Estos casos, aunque subjetiva, deben tenerse en cuenta.

## Aplicar formato a expresiones if

La sangría de los condicionales depende del tamaño de las expresiones que los componen. Si `cond` `e1` y `e2` son cortos, simplemente escríbalos en una línea:

```
if cond then e1 else e2
```

Si `cond` , `e1` o `e2` son más largas, pero no varias líneas:

```
if cond
then e1
else e2
```

Si alguna de las expresiones es de varias líneas:

```

if cond then
  e1
else
  e2

```

A varios condicionales con `elif` y `else` se les aplica una sangría en el mismo ámbito que el `if` :

```

if cond1 then e1
elif cond2 then e2
elif cond3 then e3
else e4

```

## Construcciones de coincidencia de patrones

Use una `|` cláusula for each de una coincidencia sin sangría. Si la expresión es breve, puede considerar el uso de una sola línea si cada subexpresión también es simple.

```

// OK
match l with
| { him = x; her = "Posh" } :: tail -> x
| _ :: tail -> findDavid tail
| [] -> failwith "Couldn't find David"

// Not OK
match l with
  | { him = x; her = "Posh" } :: tail -> x
  | _ :: tail -> findDavid tail
  | [] -> failwith "Couldn't find David"

```

Si la expresión de la derecha de la flecha de coincidencia de patrones es demasiado grande, muévela a la línea siguiente, con una sangría a un paso de `match` / `|` .

```

match lam with
| Var v -> 1
| Abs(x, body) ->
  1 + sizeLambda body
| App(lam1, lam2) ->
  sizeLambda lam1 + sizeLambda lam2

```

La coincidencia de patrones de funciones anónimas, a partir de `function` , no suele ser una sangría demasiado lejana. Por ejemplo, la sangría de un ámbito es la siguiente:

```

lambdaList
|> List.map (function
  | Abs(x, body) -> 1 + sizeLambda 0 body
  | App(lam1, lam2) -> sizeLambda (sizeLambda 0 lam1) lam2
  | Var v -> 1)

```

La coincidencia de patrones en funciones definidas por `let` o `let rec` debe tener una sangría de cuatro espacios después de iniciarse `let` , incluso si `function` se usa la palabra clave:

```

let rec sizeLambda acc = function
  | Abs(x, body) -> sizeLambda (succ acc) body
  | App(lam1, lam2) -> sizeLambda (sizeLambda acc lam1) lam2
  | Var v -> succ acc

```

No se recomienda alinear las flechas.

## Aplicar formato a expresiones try/with

Se debe aplicar sangría a la coincidencia de patrones en el tipo de excepción en el mismo nivel que `with` .

```
try
  if System.DateTime.Now.Second % 3 = 0 then
    raise (new System.Exception())
  else
    raise (new System.ApplicationException())
with
| :? System.ApplicationException ->
  printfn "A second that was not a multiple of 3"
| _ ->
  printfn "A second that was a multiple of 3"
```

## Aplicación de formato de parámetros de función

En general, la mayoría de las aplicaciones de parámetros de función se realizan en la misma línea.

Si desea aplicar parámetros a una función en una nueva línea, aplíqueles sangría en un ámbito.

```
// OK
sprintf "\t%s - %i\n\r"
    x.IngredientName x.Quantity

// OK
sprintf
    "\t%s - %i\n\r"
    x.IngredientName x.Quantity

// OK
let printVolumes x =
    printf "Volume in liters = %f, in us pints = %f, in imperial = %f"
        (convertVolumeToLiter x)
        (convertVolumeUSPint x)
        (convertVolumeImperialPint x)
```

Las mismas directrices se aplican a las expresiones lambda como argumentos de función. Si el cuerpo de una expresión lambda, el cuerpo puede tener otra línea, con sangría en un ámbito

```
let printListWithOffset a list1 =
    List.iter
        (fun elem -> printfn "%d" (a + elem))
        list1

// OK if lambda body is long enough
let printListWithOffset a list1 =
    List.iter
        (fun elem ->
            printfn "%d" (a + elem))
        list1
```

Sin embargo, si el cuerpo de una expresión lambda tiene más de una línea, considere la posibilidad de factorizarla en una función independiente en lugar de tener una construcción de varias líneas aplicada como un solo argumento a una función.

### Aplicar formato a los operadores de infijo

Operadores independientes por espacios. Las excepciones obvias a esta regla son los `!` `.` operadores y.

Las expresiones infijas son correctas para la misma columna:

```
acc +
(sprintf "%t%s - %i\n\r"
    x.IngredientName x.Quantity)

let function1 arg1 arg2 arg3 arg4 =
    arg1 + arg2 +
    arg3 + arg4
```

## Operadores de canalización de formato

`|>` Los operadores de canalización deben ir debajo de las expresiones en las que operan.

```
// Preferred approach
let methods2 =
    System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun assm -> assm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat

// Not OK
let methods2 = System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun assm -> assm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat
```

## Módulos de formato

Se debe aplicar sangría al código de un módulo local en relación con el módulo, pero no se debe aplicar sangría al código de un módulo de nivel superior. No es necesario aplicar sangría a los elementos de espacio de nombres.

```
// A is a top-level module.
module A

let function1 a b = a - b * b
```

```
// A1 and A2 are local modules.
module A1 =
    let function1 a b = a * a + b * b

module A2 =
    let function2 a b = a * a - b * b
```

## Dar formato a expresiones e interfaces de objeto

Las interfaces y las expresiones de objeto se deben alinear de la misma manera con `member` una sangría después de cuatro espacios.

```
let comparer =
    { new IComparer<string> with
        member x.Compare(s1, s2) =
            let rev (s: String) =
                new String (Array.rev (s.ToCharArray()))
            let reversed = rev s1
            reversed.CompareTo (rev s2) }
```

## Aplicar formato a los espacios en blanco en expresiones

Evite el espacio en blanco extraño en las expresiones de F #.

```
// OK
spam (ham.[1])

// Not OK
spam ( ham.[ 1 ] )
```

Los argumentos con nombre tampoco deben tener espacio alrededor del `=` :

```
// OK
let makeStreamReader x = new System.IO.StreamReader(path=x)

// Not OK
let makeStreamReader x = new System.IO.StreamReader(path = x)
```

## Aplicar formato a atributos

Los [atributos](#) se colocan sobre una construcción:

```
[<SomeAttribute>]
type MyClass() = ...

[<RequireQualifiedAccess>]
module M =
    let f x = x

[<Struct>]
type MyRecord =
    { Label1: int
      Label2: string }
```

## Aplicar formato a atributos en parámetros

Los atributos también se pueden colocar en parámetros. En este caso, coloque entonces en la misma línea que el parámetro y antes del nombre:

```
// Defines a class that takes an optional value as input defaulting to false.
type C() =
    member _.M([<Optional; DefaultValue(false)>] doSomething: bool)
```

## Aplicar formato a varios atributos

Cuando se aplican varios atributos a una construcción que no es un parámetro, deben colocarse de forma que haya un atributo por línea:

```
[<Struct>]
[<IsByRefLike>]
type MyRecord =
    { Label1: int
      Label2: string }
```

Cuando se aplica a un parámetro, deben estar en la misma línea y estar separados por un `;` separador.

## Aplicar formato a literales

Los [literals de F #](#) que usan el `Literal` atributo deben colocar el atributo en su propia línea y usar PascalCase naming:

```
[<Literal>]
let Path = __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let MyUrl = "www.mywebsitethatiamworkingwith.com"
```

Evite colocar el atributo en la misma línea que el valor.

# Convenciones de código de F#

23/07/2020 • 51 minutes to read • [Edit Online](#)

Las siguientes convenciones se formulan a partir de experiencias que trabajan con códigos base de F# de gran tamaño. Los [cinco principios de buen código de F#](#) son la base de cada recomendación. Están relacionados con las [instrucciones de diseño del componente de f#](#), pero son aplicables a cualquier código de f#, no solo a los componentes como las bibliotecas.

## Organizar código

F# presenta dos maneras principales de organizar el código: los módulos y los espacios de nombres. Son similares, pero tienen las siguientes diferencias:

- Los espacios de nombres se compilan como espacios de nombres .NET. Los módulos se compilan como clases estáticas.
- Los espacios de nombres siempre son de nivel superior. Los módulos pueden ser de nivel superior y estar anidados dentro de otros módulos.
- Los espacios de nombres pueden abarcar varios archivos. Los módulos no pueden.
- Los módulos se pueden decorar con `[<RequireQualifiedAccess>]` y `[<AutoOpen>]`.

Las instrucciones siguientes le ayudarán a usarlos para organizar el código.

### Preferir espacios de nombres en el nivel superior

En el caso de cualquier código consumible públicamente, los espacios de nombres son preferentes para los módulos en el nivel superior. Dado que se compilan como espacios de nombres .NET, se pueden consumir desde C# sin ningún problema.

```
// Good!
namespace MyCode

type MyClass() =
    ...
```

El uso de un módulo de nivel superior no puede aparecer de forma diferente cuando se llama solo desde F#, pero para los consumidores de C#, es posible que los autores de llamadas se sorprendan al tener que calificar `MyClass` con el `MyCode` módulo.

```
// Bad!
module MyCode

type MyClass() =
    ...
```

### Aplicar cuidadosamente `[<AutoOpen>]`

La `[<AutoOpen>]` construcción puede contaminar el ámbito de lo que está disponible para los autores de llamadas y la respuesta a la que procede algo de es "Magic". Esto no es una buena cosa. Una excepción a esta regla es la propia biblioteca básica de F# (aunque este hecho también es un poco polémico).

Sin embargo, es una comodidad si tiene una funcionalidad auxiliar para una API pública que desea organizar por separado de esa API pública.



```

module MyAPI =
    [<AutoOpen>]
    module private Helpers =
        let helper1 x y z =
            ...

    let myFunction1 x =
        let y = ...
        let z = ...

        helper1 x y z

```

Esto le permite separar claramente los detalles de implementación de la API pública de una función sin tener que calificar totalmente una aplicación auxiliar cada vez que la llame.

Además, exponer los métodos de extensión y los generadores de expresiones en el nivel de espacio de nombres se puede expresar perfectamente con `[<AutoOpen>]`.

**Usar `[<RequireQualifiedAccess>]` siempre que los nombres puedan entrar en conflicto o se sienta útil para mejorar la legibilidad**

Agregar el `[<RequireQualifiedAccess>]` atributo a un módulo indica que el módulo no se puede abrir y que las referencias a los elementos del módulo requieren un acceso calificado explícito. Por ejemplo, el `Microsoft.FSharp.Collections.List` módulo tiene este atributo.

Esto resulta útil cuando las funciones y los valores del módulo tienen nombres que es probable que entren en conflicto con los nombres de otros módulos. Requerir acceso cualificado puede aumentar considerablemente el mantenimiento a largo plazo y la evolución de una biblioteca.

```

[<RequireQualifiedAccess>]
module StringTokenization =
    let parse s = ...

    ...

    let s = getAString()
    let parsed = StringTokenization.parse s // Must qualify to use 'parse'

```

## Ordenar `open` instrucciones topológicamente

En F #, el orden de las declaraciones es importante, incluidas las `open` instrucciones `with`. Esto es a diferencia de C#, donde el efecto de `using` y `using static` es independiente del orden de esas instrucciones en un archivo.

En F #, los elementos abiertos en un ámbito pueden prevalecer sobre otros. Esto significa que la reordenación `open` de las instrucciones podría modificar el significado del código. Como resultado, no se recomienda ninguna ordenación arbitraria de todas las `open` instrucciones (por ejemplo, alfanuméricamente), consulta generar un comportamiento diferente que podría esperar.

En su lugar, se recomienda ordenarlas **topológicamente**; es decir, ordene las `open` instrucciones en el orden en el que se definen las *capas* del sistema. También se puede tener en cuenta la ordenación alfanumérica en diferentes capas topológicas.

Por ejemplo, esta es la Ordenación topológica del archivo de API pública del servicio del compilador de F #:

```

namespace Microsoft.FSharp.Compiler.SourceCodeServices

open System
open System.Collections.Generic
open System.Collections.Concurrent
open System.Diagnostics
open System.IO
open System.Reflection
open System.Text

open Microsoft.FSharp.Compiler
open Microsoft.FSharp.Compiler.AbstractIL
open Microsoft.FSharp.Compiler.AbstractIL.Diagnostics
open Microsoft.FSharp.Compiler.AbstractIL.IL
open Microsoft.FSharp.Compiler.AbstractIL.ILBinaryReader
open Microsoft.FSharp.Compiler.AbstractIL.Internal
open Microsoft.FSharp.Compiler.AbstractIL.Internal.Library

open Microsoft.FSharp.Compiler.AccessibilityLogic
open Microsoft.FSharp.Compiler.Ast
open Microsoft.FSharp.Compiler.CompileOps
open Microsoft.FSharp.Compiler.CompileOptions
open Microsoft.FSharp.Compiler.Driver
open Microsoft.FSharp.Compiler.ErrorLogger
open Microsoft.FSharp.Compiler.Infos
open Microsoft.FSharp.Compiler.InfoReader
open Microsoft.FSharp.Compiler.Lexhelp
open Microsoft.FSharp.Compiler.Layout
open Microsoft.FSharp.Compiler.Lib
open Microsoft.FSharp.Compiler.NameResolution
open Microsoft.FSharp.Compiler.PrettyNaming
open Microsoft.FSharp.Compiler.Parser
open Microsoft.FSharp.Compiler.Range
open Microsoft.FSharp.Compiler.Tast
open Microsoft.FSharp.Compiler.Tastops
open Microsoft.FSharp.Compiler.TcGlobals
open Microsoft.FSharp.Compiler.TypeChecker
open Microsoft.FSharp.Compiler.SourceCodeServices.SymbolHelpers

open Internal.Utilities
open Internal.Utilities.Collections

```

Tenga en cuenta que un salto de línea separa las capas topológicas, y cada capa se ordena de forma alfanumérica después. Esto organiza correctamente el código sin sombrear los valores accidentalmente.

## Usar clases para contener valores que tengan efectos secundarios

Hay muchas ocasiones en las que la inicialización de un valor puede tener efectos secundarios, como la creación de instancias de un contexto en una base de datos u otro recurso remoto. Es tentador inicializar estos elementos en un módulo y usarlo en las funciones siguientes:

```

// This is bad!
module MyApi =
    let dep1 = File.ReadAllText "/Users/{your name}/connectionstring.txt"
    let dep2 = Environment.GetEnvironmentVariable "DEP_2"

    let private r = Random()
    let dep3() = r.Next() // Problematic if multiple threads use this

    let function1 arg = doStuffWith dep1 dep2 dep3 arg
    let function2 arg = doSutffWith dep1 dep2 dep3 arg

```

Esta suele ser una mala idea por varias razones:

En primer lugar, la configuración de la aplicación se inserta en el código base con `dep1` y `dep2`. Esto es difícil de mantener en códigos base mayores.

En segundo lugar, los datos inicializados de forma estática no deben incluir valores que no sean seguros para subprocesos si el componente utilizará varios subprocesos. Esto es claramente infringido por `dep3`.

Por último, la inicialización del módulo se compila en un constructor estático para toda la unidad de compilación. Si se produce algún error en la inicialización de los valores enlazados en ese módulo, se manifiesta como un `TypeInitializationException` que se almacena en caché para toda la duración de la aplicación. Esto puede ser difícil de diagnosticar. Normalmente, hay una excepción interna que se puede tratar de explicar, pero si no lo está, no hay que indicar cuál es la causa raíz.

En su lugar, use simplemente una clase simple para almacenar las dependencias:

```
type MyParametricApi(dep1, dep2, dep3) =  
    member _.Function1 arg1 = doStuffWith dep1 dep2 dep3 arg1  
    member _.Function2 arg2 = doStuffWith dep1 dep2 dep3 arg2
```

Esto permite lo siguiente:

1. Inserción de cualquier estado dependiente fuera de la propia API.
2. La configuración ahora se puede realizar fuera de la API.
3. No es probable que los errores de inicialización de los valores dependientes se manifiesten como `TypeInitializationException`.
4. La API es ahora más fácil de probar.

## Administración de errores

La administración de errores en grandes sistemas es un esfuerzo complejo y con matices, y no hay ninguna viñeta Silver para asegurarse de que los sistemas son tolerantes a errores y se comportan correctamente. Las siguientes directrices deben proporcionar orientación para navegar por este espacio difícil.

### Representar casos de error y estado no válido en tipos intrínsecos de su dominio

Con las [uniones discriminadas](#), F # ofrece la capacidad de representar el estado de un programa defectuoso en el sistema de tipos. Por ejemplo:

```
type MoneyWithdrawalResult =  
    | Success of amount:decimal  
    | InsufficientFunds of balance:decimal  
    | CardExpired of DateTime  
    | UndisclosedFailure
```

En este caso, se puede producir un error en las tres formas conocidas de retirar dinero de una cuenta bancaria. Cada caso de error se representa en el tipo y, por tanto, se puede tratar con seguridad en todo el programa.

```
let handleWithdrawal amount =  
    let w = withdrawMoney amount  
    match w with  
    | Success am -> printfn "Successfully withdrew %f" am  
    | InsufficientFunds balance -> printfn "Failed: balance is %f" balance  
    | CardExpired expiredDate -> printfn "Failed: card expired on %0" expiredDate  
    | UndisclosedFailure -> printfn "Failed: unknown"
```

En general, si puede modelar las distintas formas en que se puede **producir un error** en el dominio, el código de control de errores ya no se trata como algo que debe abordar además del flujo normal del programa. Es

simplemente parte del flujo normal del programa y no se considera **excepcional**. Esto tiene dos ventajas principales:

1. Es más fácil de mantener a medida que su dominio cambia con el tiempo.
2. Los casos de error son más fáciles de probar unitarios.

### Usar excepciones cuando los errores no se pueden representar con tipos

No todos los errores se pueden representar en un dominio problemático. Estos tipos de errores son *excepcionales* por naturaleza; por lo tanto, la capacidad de generar y detectar excepciones en F #.

En primer lugar, se recomienda leer las [instrucciones de diseño de excepciones](#). También se aplican a F #.

Las construcciones principales disponibles en F # con el fin de generar excepciones se deben considerar en el orden de preferencia siguiente:

FUNCIÓN	SINTAXIS	FINALIDAD
<code>nullArg</code>	<code>nullArg "argumentName"</code>	Genera una <code>System.ArgumentNullException</code> con el nombre de argumento especificado.
<code>invalidArg</code>	<code>invalidArg "argumentName" "message"</code>	Genera una <code>System.ArgumentException</code> con el nombre de argumento y el mensaje especificados.
<code>invalidOp</code>	<code>invalidOp "message"</code>	Genera una <code>System.InvalidOperationException</code> con el mensaje especificado.
<code>raise</code>	<code>raise (ExceptionType("message"))</code>	Mecanismo de uso general para producir excepciones.
<code>failwith</code>	<code>failwith "message"</code>	Genera una <code>System.Exception</code> con el mensaje especificado.
<code>failwithf</code>	<code>failwithf "format string" argForFormatString</code>	Genera una <code>System.Exception</code> con un mensaje determinado por la cadena de formato y sus entradas.

Use `nullArg`, `invalidArg` y `invalidOp` como el mecanismo que se va a iniciar `ArgumentNullException`, `ArgumentException` y `InvalidOperationException` cuando corresponda.

`failwith` y `failwithf` Por lo general, las funciones y deben evitarse porque generan el `Exception` tipo base, no una excepción concreta. En función de las [instrucciones de diseño de excepciones](#), desea producir excepciones más específicas cuando sea posible.

### Usar la sintaxis de control de excepciones

F # admite patrones de excepción a través de la `try...with` Sintaxis:

```
try
    tryGetFileContents()
with
| :? System.IO.FileNotFoundException as e -> // Do something with it here
| :? System.Security.SecurityException as e -> // Do something with it here
```

La reconciliación de la funcionalidad para realizar en el caso de una excepción con la coincidencia de patrones

puede ser un poco complicada si desea mantener el código limpio. Una manera de controlar esto es usar [patrones activos](#) como medio para agrupar la funcionalidad en torno a un caso de error con una excepción. Por ejemplo, puede que esté consumiendo una API que, cuando se produce una excepción, incluye información valiosa en los metadatos de la excepción. Desencapsular un valor útil en el cuerpo de la excepción capturada dentro del modelo activo y devolver ese valor puede ser útil en algunas situaciones.

### No usar el control de errores Monad para reemplazar excepciones

Las excepciones se ven como algo Taboo en la programación funcional. De hecho, las excepciones infringen la pureza, por lo que es seguro considerarlas que no son bastante funcionales. Sin embargo, esto omite la realidad de dónde se debe ejecutar el código y que se pueden producir errores en tiempo de ejecución. En general, escriba código en el supuesto de que la mayoría de las cosas no son puras ni totales, con el fin de minimizar las sorpresas desagradables.

Es importante tener en cuenta los siguientes aspectos principales y aspectos de las excepciones con respecto a su relevancia y adecuadidad en el entorno de tiempo de ejecución .NET y el ecosistema multilingüe:

1. Contienen información de diagnóstico detallada, que es muy útil al depurar un problema.
2. Son perfectamente entendidos por el tiempo de ejecución y otros lenguajes de .NET.
3. Pueden reducir el REUTILIZADOR significativo cuando se comparan con el código que sale de su forma de *evitar* excepciones mediante la implementación de un subconjunto de su semántica de manera ad hoc.

Este tercer punto es crítico. En el caso de las operaciones complejas no triviales, el uso de excepciones puede tener como resultado la gestión de estructuras como esta:

```
Result<Result<MyType, string>, string list>
```

Que puede conducir fácilmente a un código frágil, como la coincidencia de patrones en errores "con tipo de cadena":

```
let result = doStuff()
match result with
| Ok r -> ...
| Error e ->
    if e.Contains "Error string 1" then ...
    elif e.Contains "Error string 2" then ...
    else ... // Who knows?
```

Además, puede ser tentador admitir cualquier excepción en el deseo de una función "simple" que devuelva un tipo "agradable":

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with _ -> None
```

Desgraciadamente, `tryReadAllText` puede producir numerosas excepciones basadas en la infinidad de cosas que pueden producirse en un sistema de archivos, y este código descarta cualquier información sobre lo que podría estar realmente mal en su entorno. Si reemplaza este código por un tipo de resultado, volverá al análisis de mensajes de error "con tipo de cadena":

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Ok
    with e -> Error e.Message

let r = tryReadAllText "path-to-file"
match r with
| Ok text -> ...
| Error e ->
    if e.Contains "uh oh, here we go again..." then ...
    else ...
```

Y colocar el propio objeto de excepción en el `Error` constructor simplemente obliga a tratar correctamente el tipo de excepción en el sitio de llamada en lugar de en la función. De este modo, se crean excepciones comprobadas, que son muy unfuns de tratar como llamador de una API.

Una buena alternativa a los ejemplos anteriores es detectar excepciones *específicas* y devolver un valor significativo en el contexto de esa excepción. Si modifica la función de la `tryReadAllText` manera siguiente, `None` tiene un significado más:

```
let tryReadAllTextIfPresent (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with :? FileNotFoundException -> None
```

En lugar de funcionar como una instrucción catch-all, esta función controlará correctamente el caso cuando no se encuentre un archivo y lo asigne a una devolución. Este valor devuelto se puede asignar a ese caso de error, sin descartar ninguna información contextual ni obligar a los autores de llamadas a tratar con un caso que puede no ser relevante en ese punto del código.

Los tipos como `Result<'Success, 'Error>` son adecuados para las operaciones básicas en las que no están anidados y los tipos opcionales de F # son perfectos para representar cuando algo puede devolver *algo* o *nada*. Sin embargo, no sustituyen a las excepciones y no se deben usar en un intento de reemplazar excepciones. En su lugar, deben aplicarse prudentemente para abordar aspectos específicos de la Directiva de administración de errores y excepciones de manera dirigida.

## Aplicación parcial y programación sin punto

F # admite aplicaciones parciales y, por tanto, varias maneras de programar en un estilo sin punto. Esto puede ser beneficioso para la reutilización de código dentro de un módulo o la implementación de algo, pero no es algo que exponer públicamente. En general, la programación sin puntos no es un marco de sí mismo y puede Agregar una barrera cognitiva significativa para personas que no están sumergidas en el estilo.

### No use la aplicación parcial y currificación en las API públicas

Con poca excepción, el uso de una aplicación parcial en las API públicas puede resultar confuso para los consumidores. Normalmente, `let` los valores enlazados en el código de F # son **valores**, no **valores de función**. La combinación de valores y valores de función puede dar lugar a que se guarde un pequeño número de líneas de código en Exchange para un poco de sobrecarga cognitiva, especialmente si se combina con operadores como `>>` para crear funciones.

### Tenga en cuenta las implicaciones de las herramientas para la programación sin puntos

Las funciones currificadas no etiquetan sus argumentos. Esto tiene implicaciones para las herramientas. Tenga en cuenta las dos funciones siguientes:

```
let func name age =
    printfn "My name is %s and I am %d years old!" name age

let funcWithApplication =
    printfn "My name is %s and I am %d years old!"
```

Ambos son funciones válidas, pero `funcWithApplication` es una función currificada. Al mantener el mouse sobre sus tipos en un editor, verá lo siguiente:

```
val func : name:string -> age:int -> unit

val funcWithApplication : (string -> int -> unit)
```

En el sitio de la llamada, la información sobre herramientas, como Visual Studio, no proporcionará información significativa sobre lo que `string` los `int` tipos de entrada y representan realmente.

Si se encuentra con código sin punto, como el `funcWithApplication` que se puede utilizar públicamente, se recomienda realizar una expansión η completa para que las herramientas puedan recoger los nombres descriptivos de los argumentos.

Además, el código sin punto de depuración puede ser desafiante, si no imposible. Las herramientas de depuración se basan en valores enlazados a nombres (por ejemplo, `let` enlaces) para que pueda inspeccionar valores intermedios a mitad de la ejecución. Cuando el código no tiene valores para inspeccionar, no hay nada que depurar. En el futuro, es posible que las herramientas de depuración evolucionen para sintetizar estos valores en función de las rutas de acceso ejecutadas anteriormente, pero no es una buena idea repartir sus *posibilidades* de depuración.

### Considere la aplicación parcial como una técnica para reducir el uso del texto interno.

A diferencia del punto anterior, la aplicación parcial es una herramienta maravillosa para reducir el contenido Reutilizable dentro de una aplicación o los aspectos internos más profundos de una API. Puede ser útil para las pruebas unitarias de la implementación de API más complicadas, donde reutilizable suele ser un problema. Por ejemplo, en el código siguiente se muestra cómo puede realizar lo que la mayoría de los marcos ficticios le proporciona sin tomar una dependencia externa de este marco de trabajo y tener que aprender una API relacionada relacionada.

Por ejemplo, considere la siguiente topografía de la solución:

```
MySolution.sln
|_ImplementationLogic.fsproj
|_ImplementationLogic.Tests.fsproj
|_API.fsproj
```

`ImplementationLogic.fsproj` podría exponer código como:

```
module Transactions =
    let doTransaction txnContext txnType balance =
        ...

    type Transactor(ctx, currentBalance) =
        member _.ExecuteTransaction(txnType) =
            Transactions.doTransaction ctx txnType currentBalance
        ...
```

Las pruebas unitarias `Transactions.doTransaction` en `ImplementationLogic.Tests.fsproj` son fáciles:

```
namespace TransactionsTestingUtil

open Transactions

module TransactionsTestable =
    let getTestableTransactionRoutine mockContext = Transactions.doTransaction mockContext
```

Aplicar parcialmente `doTransaction` con un objeto de contexto ficticio le permite llamar a la función en todas las pruebas unitarias sin necesidad de crear un contexto ficticio cada vez:

```
namespace TransactionTests

open Xunit
open TransactionTypes
open TransactionsTestingUtil
open TransactionsTestingUtil.TransactionsTestable

let testableContext =
    { new ITransactionContext with
        member _.TheFirstMember() = ...
        member _.TheSecondMember() = ... }

let transactionRoutine = getTestableTransactionRoutine testableContext

[<Fact>]
let ``Test withdrawal transaction with 0.0 for balance``() =
    let expected = ...
    let actual = transactionRoutine TransactionType.Withdraw 0.0
    Assert.Equal(expected, actual)
```

Esta técnica no se debe aplicar universalmente a toda la base de código, pero es una buena forma de reducir los elementos reutilizables para las pruebas unitarias e internas complicadas.

## Control de acceso

F # tiene varias opciones para el [control de acceso](#), que se heredan de lo que está disponible en el tiempo de ejecución de .net. No solo se pueden usar para los tipos. también puede utilizarlos para funciones.

- Prefiere `public` los tipos y miembros que no sean de tipo y hasta que los necesite para que los consuma públicamente. Esto también minimiza lo que los consumidores acoplan a.
- Procure mantener toda la funcionalidad de la aplicación auxiliar `private`.
- Considere el uso de `[<AutoOpen>]` en un módulo privado de funciones auxiliares si son numerosas.

## Inferencia de tipos y genéricos

La inferencia de tipos puede ahorrarle escribir una gran cantidad de texto reutilizable. Y la generalización automática en el compilador de F # pueden ayudarle a escribir código más genérico sin ningún esfuerzo adicional por su parte. Sin embargo, estas características no son universalmente buenas.

- Considere la posibilidad de etiquetar nombres de argumento con tipos explícitos en las API públicas y no confíe en la inferencia de tipos para este.

El **motivo es que debería tener** el control de la forma de la API, no del compilador. Aunque el compilador puede realizar un trabajo adecuado en la inferencia de tipos automáticamente, es posible que la forma de la API cambie si el interno en el que se basa ha cambiado de tipo. Esto puede ser lo que desea, pero es muy probable que se produzca un cambio de API importante que los consumidores de nivel inferior tendrán que tratar. En su lugar, si controla explícitamente la forma de la API pública, puede controlar estos cambios importantes. En términos de DDD, esto puede considerarse como una capa contra daños.



- Considere la posibilidad de asignar un nombre descriptivo a los argumentos genéricos.

A menos que esté escribiendo código realmente genérico que no sea específico de un dominio determinado, un nombre descriptivo puede ayudar a otros programadores a entender el dominio en el que están trabajando. Por ejemplo, un parámetro de tipo denominado `Document` en el contexto de interactuar con una base de datos de documento hace que sea más claro que la función o el miembro con el que está trabajando puede aceptar los tipos de documentos genéricos.

- Considere la posibilidad de asignar nombres a los parámetros de tipo genérico con PascalCase.

Esta es la manera general de hacer cosas en .NET, por lo que se recomienda que use PascalCase en lugar de snake_case o camelCase.

Por último, la generalización automática no siempre es una gran ventaja para las personas que no están familiarizadas con F # o con un código base grande. El uso de componentes que son genéricos es una sobrecarga cognitiva. Además, si las funciones generalizadas automáticamente no se usan con distintos tipos de entrada (por sí solo si están pensadas para usarse como tal), no hay ninguna ventaja real de que sean genéricas en ese momento. Considere siempre si el código que está escribiendo se beneficiará de ser genérico.

## Rendimiento

### Preferir Structs para tipos de datos pequeños

El uso de Structs (también denominados tipos de valor) puede dar lugar a un mayor rendimiento en el código, ya que normalmente evita la asignación de objetos. Sin embargo, los Structs no siempre son un botón "ir más rápido": Si el tamaño de los datos de una estructura supera los 16 bytes, la copia de los datos a menudo puede producir más tiempo de CPU que el uso de un tipo de referencia.

Para determinar si debe usar un struct, tenga en cuenta las siguientes condiciones:

- Si el tamaño de los datos es de 16 bytes o menos.
- Si es probable que tenga muchos de estos tipos de datos residentes en memoria en un programa en ejecución.

Si se aplica la primera condición, generalmente debería usar un struct. Si ambos se aplican, casi siempre debe usar un struct. Puede haber algunos casos en los que se apliquen las condiciones anteriores, pero el uso de una estructura no es mejor o peor que el uso de un tipo de referencia, pero es probable que no sean habituales. Es importante medir siempre cuando se realizan cambios como este, sin embargo, y no funcionan en supuestos o Intuition.

### Preferir tuplas de struct al agrupar tipos de valores pequeños

Tenga en cuenta las dos funciones siguientes:

```

let rec runWithTuple t offset times =
    let offsetValues x y z offset =
        (x + offset, y + offset, z + offset)

    if times <= 0 then
        t
    else
        let (x, y, z) = t
        let r = offsetValues x y z offset
        runWithTuple r offset (times - 1)

let rec runWithStructTuple t offset times =
    let offsetValues x y z offset =
        struct(x + offset, y + offset, z + offset)

    if times <= 0 then
        t
    else
        let struct(x, y, z) = t
        let r = offsetValues x y z offset
        runWithStructTuple r offset (times - 1)

```

Cuando realice una prueba comparativa de estas funciones con una herramienta estadística de pruebas comparativas como [BenchmarkDotNet](#), verá que la `runWithStructTuple` función que usa las tuplas de struct se ejecuta un 40% más rápido y no asigna memoria.

Sin embargo, estos resultados no siempre serán el caso en su propio código. Si marca una función como `inline`, el código que usa tuplas de referencia puede obtener algunas optimizaciones adicionales, o el código que asignaría podría simplemente estar optimizado. Siempre debe medir los resultados siempre que el rendimiento esté relacionado y nunca funcione en función de supuestos o Intuition.

#### Preferir registros struct cuando el tipo de datos sea pequeño

La regla de Thumb descrita anteriormente también contiene los [tipos de registro de F #](#). Tenga en cuenta los siguientes tipos de datos y funciones que los procesan:

```

type Point = { X: float; Y: float; Z: float }

[<Struct>]
type SPoint = { X: float; Y: float; Z: float }

let rec processPoint (p: Point) offset times =
    let inline offsetValues (p: Point) offset =
        { p with X = p.X + offset; Y = p.Y + offset; Z = p.Z + offset }

    if times <= 0 then
        p
    else
        let r = offsetValues p offset
        processPoint r offset (times - 1)

let rec processStructPoint (p: SPoint) offset times =
    let inline offsetValues (p: SPoint) offset =
        { p with X = p.X + offset; Y = p.Y + offset; Z = p.Z + offset }

    if times <= 0 then
        p
    else
        let r = offsetValues p offset
        processStructPoint r offset (times - 1)

```

Esto es similar al código de tupla anterior, pero esta vez el ejemplo usa registros y una función interna insertada.

Al realizar una prueba comparativa de estas funciones con una herramienta estadística de pruebas comparativas como [BenchmarkDotNet](#), observará que `processStructPoint` se ejecuta casi un 60% más rápido y asigna nada en el montón administrado.

#### **Prefieren uniones discriminadas de struct cuando el tipo de datos es pequeño**

Las observaciones anteriores sobre el rendimiento con tuplas y registros de struct también contienen [uniones discriminadas de F #](#). Observe el código siguiente:

```
type Name = Name of string

[<Struct>]
type SName = SName of string

let reverseName (Name s) =
    s.ToCharArray()
    |> Array.rev
    |> string
    |> Name

let structReverseName (SName s) =
    s.ToCharArray()
    |> Array.rev
    |> string
    |> SName
```

Es habitual definir uniones discriminadas de un solo caso como esta para el modelado de dominios. Cuando realice pruebas comparativas de estas funciones con una herramienta estadística comparativa como [BenchmarkDotNet](#), encontrará que `structReverseName` se ejecuta aproximadamente un 25% más rápido que `reverseName` para cadenas pequeñas. En el caso de las cadenas grandes, ambos realizan aproximadamente el mismo. Por lo tanto, en este caso, siempre es preferible usar un struct. Como se mencionó anteriormente, mida siempre y no opere en suposiciones o Intuition.

Aunque en el ejemplo anterior se mostró que una Unión discriminada de struct produjeron un mejor rendimiento, es habitual tener uniones discriminadas más grandes al modelar un dominio. Los tipos de datos de mayor tamaño, como, pueden no funcionar también si son Structs en función de las operaciones que contengan, ya que pueden estar implicados más copias.

#### **Programación funcional y mutación**

Los valores de F # son inmutables de forma predeterminada, lo que permite evitar ciertas clases de errores (especialmente los que implican la simultaneidad y el paralelismo). Sin embargo, en algunos casos, para lograr una eficiencia óptima (o incluso razonable) de tiempo de ejecución o asignaciones de memoria, es posible que se pueda implementar un intervalo de trabajo mediante la mutación en contexto del estado. Esto es posible en función de la inclusión de F # con la `mutable` palabra clave.

El uso de `mutable` en F # puede sentirse en probabilidades de la pureza funcional. Esto es comprensible, pero la pureza funcional en todo el mundo puede ser probable con objetivos de rendimiento. Un compromiso es encapsular la mutación de modo que los llamadores no tengan que preocuparse de lo que sucede cuando llaman a una función. Esto le permite escribir una interfaz funcional sobre una implementación basada en mutación para el código crítico para el rendimiento.

#### **Ajuste del código mutable en interfaces inmutables**

Con la transparencia referencial como objetivo, es fundamental escribir código que no exponga la subclave mutable de las funciones críticas para el rendimiento. Por ejemplo, el código siguiente implementa la

`Array.contains` función en la biblioteca básica de F #:

```
[<CompiledName("Contains")>]
let inline contains value (array: 'T[] ) =
    checkNotNull "array" array
    let mutable state = false
    let mutable i = 0
    while not state && i < array.Length do
        state <- value = array.[i]
        i <- i + 1
    state
```

Si se llama a esta función varias veces, no se cambia la matriz subyacente, ni se requiere que se mantenga ningún estado mutable en su consumo. Es referencialmente transparente, aunque casi todas las líneas de código dentro de él utilicen mutación.

#### Considere la posibilidad de encapsular datos mutables en clases

En el ejemplo anterior se usaba una sola función para encapsular operaciones mediante datos mutables. Esto no siempre es suficiente para conjuntos de datos más complejos. Tenga en cuenta los siguientes conjuntos de funciones:

```
open System.Collections.Generic

let addToClosureTable (key, value) (t: Dictionary<_,_>) =
    if not (t.ContainsKey(key)) then
        t.Add(key, value)
    else
        t.[key] <- value

let closureTableCount (t: Dictionary<_,_>) = t.Count

let closureTableContains (key, value) (t: Dictionary<_, HashSet<_>>) =
    match t.TryGetValue(key) with
    | (true, v) -> v.Equals(value)
    | (false, _) -> false
```

Este código tiene un rendimiento, pero expone la estructura de datos basada en mutación que los llamadores son responsables de mantener. Esto puede ajustarse dentro de una clase sin miembros subyacentes que puedan cambiar:

```
open System.Collections.Generic

/// The results of computing the LALR(1) closure of an LR(0) kernel
type Closure1Table() =
    let t = Dictionary<Item0, HashSet<TerminalIndex>>()

    member _.Add(key, value) =
        if not (t.ContainsKey(key)) then
            t.Add(key, value)
        else
            t.[key] <- value

    member _.Count = t.Count

    member _.Contains(key, value) =
        match t.TryGetValue(key) with
        | (true, v) -> v.Equals(value)
        | (false, _) -> false
```

`Closure1Table` encapsula la estructura de datos basada en mutación subyacente, de modo que no obliga a los llamadores a mantener la estructura de datos subyacente. Las clases son una manera eficaz de encapsular datos y rutinas que se basan en la mutación sin exponer los detalles a los llamadores.

### Prefiere `let mutable` a las celdas de referencia

Las celdas de referencia son una manera de representar la referencia a un valor en lugar del propio valor. Aunque se pueden usar para el código crítico para el rendimiento, no se recomiendan. Considere el ejemplo siguiente:

```
let kernels =
    let acc = ref Set.empty

    processWorkList startKernels (fun kernel ->
        if not ((!acc).Contains(kernel)) then
            acc := (!acc).Add(kernel)
        ...)

!acc |> Seq.toList
```

El uso de una celda de referencia ahora "contamina" todo el código subsiguiente con tener que desreferenciar y volver a hacer referencia a los datos subyacentes. En su lugar, tenga en cuenta lo `let mutable` siguiente:

```
let kernels =
    let mutable acc = Set.empty

    processWorkList startKernels (fun kernel ->
        if not (acc.Contains(kernel)) then
            acc <- acc.Add(kernel)
        ...)

acc |> Seq.toList
```

Aparte del punto único de mutación en el medio de la expresión lambda, el resto del código que toca `acc` puede hacerlo de una manera diferente al uso de un `let` valor inmutable enlazado normal. Esto hará que sea más fácil cambiar con el tiempo.

## Programación de objetos

F # es totalmente compatible con los objetos y los conceptos orientados a objetos (OO). Aunque muchos conceptos de OO son eficaces y útiles, no todos ellos son ideales para su uso. En las listas siguientes se ofrecen instrucciones sobre las categorías de características de OO en un nivel alto.

**Considere la posibilidad de usar estas características en muchas situaciones:**

- Notación de puntos ( `x.Length` )
- Miembros de instancia
- Constructores implícitos
- Miembros estáticos
- Notación de indexador ( `arr.[x]` )
- Argumentos opcionales y con nombre
- Interfaces e implementaciones de interfaz

**No se debe llegar a estas características en primer lugar, pero se aplican de forma prudente cuando sea conveniente solucionar un problema:**

- Sobrecarga de métodos
- Datos mutables encapsulados
- Operadores en tipos
- Propiedades automáticas
- Implementación `IDisposable` de y `IEnumerable`

- Extensiones de tipo
- Eventos
- Estructuras
- Delegados
- Enumeraciones

Normalmente, evite estas características a menos que deba utilizarlas:

- Jerarquías de tipos basados en herencia y herencia de implementación
- Valores NULL y `Unchecked.defaultof<_>`

### Preferir composición sobre herencia

La [composición sobre la herencia](#) es una expresión de larga duración que puede cumplir el código de F#. El principio fundamental es que no se debe exponer una clase base y forzar que los llamadores hereden de esa clase base para obtener la funcionalidad.

### Usar expresiones de objeto para implementar interfaces si no se necesita una clase

Las [expresiones de objeto](#) permiten implementar interfaces sobre la marcha, enlazando la interfaz implementada a un valor sin necesidad de hacerlo dentro de una clase. Esto resulta cómodo, especialmente si *solo* necesita implementar la interfaz y no necesita una clase completa.

Por ejemplo, este es el código que se ejecuta en [lonide](#) para proporcionar una acción de corrección de código si ha agregado un símbolo para el que no tiene una `open` instrucción:

```
let private createProvider () =
    { new CodeActionProvider with
        member this.provideCodeActions(doc, range, context, ct) =
            let diagnostics = context.diagnostics
            let diagnostic = diagnostics |> Seq.tryFind (fun d -> d.message.Contains "Unused open
statement")
            let res =
                match diagnostic with
                | None -> [[]]
                | Some d ->
                    let line = doc.lineAt d.range.start.line
                    let cmd = createEmpty<Command>
                    cmd.title <- "Remove unused open"
                    cmd.command <- "fsharp.unusedOpenFix"
                    cmd.arguments <- Some ([| doc |> unbox; line.range |> unbox; [] |> ResizeArray)
                    [|cmd |]
            res
            |> ResizeArray
            |> U2.Case1
    }
```

Dado que no hay necesidad de una clase al interactuar con la API de Visual Studio Code, las expresiones de objeto son una herramienta ideal para esto. También son útiles para las pruebas unitarias, cuando se desea realizar un código auxiliar de una interfaz con rutinas de prueba de forma ad hoc.

## Considere las abreviaturas de tipo para acortar las firmas

Las [abreviaturas de tipo](#) son una manera cómoda de asignar una etiqueta a otro tipo, como una firma de función o un tipo más complejo. Por ejemplo, el alias siguiente asigna una etiqueta a lo que se necesita para definir un cálculo con [CNTK](#), una biblioteca de aprendizaje profundo:

```
open CNTK
```

```
// DeviceDescriptor, Variable, and Function all come from CNTK
type Computation = DeviceDescriptor -> Variable -> Function
```

El `Computation` nombre es una manera cómoda de indicar cualquier función que coincida con la firma a la que se asignan alias. El uso de abreviaturas de tipo como esta es cómodo y permite el código más conciso.

### Evite el uso de abreviaturas de tipo para representar su dominio

Aunque las abreviaturas de tipo son convenientes para asignar un nombre a las firmas de función, pueden resultar confusos al abreviar otros tipos. Tenga en cuenta esta abreviatura:

```
// Does not actually abstract integers.
type BufferSize = int
```

Esto puede resultar confuso de varias maneras:

- `BufferSize` no es una abstracción; es simplemente otro nombre para un entero.
- Si `BufferSize` se expone en una API pública, se puede malinterpretar fácilmente que significa algo más que simplemente `int`. Por lo general, los tipos de dominio tienen varios atributos y no son tipos primitivos como `int`. Esta abreviatura infringe esa hipótesis.
- El uso de mayúsculas y minúsculas de `BufferSize` (PascalCase) implica que este tipo contiene más datos.
- Este alias no ofrece mayor claridad en comparación con proporcionar un argumento con nombre a una función.
- La abreviatura no emitirá un manifiesto en IL compilado; es simplemente un entero y este alias es una construcción en tiempo de compilación.

```
module Networking =
    ...
    let send data (bufferSize: int) = ...
```

En Resumen, el inconveniente de las abreviaturas de tipo es que **no** son abstracciones sobre los tipos que se van a abreviar. En el ejemplo anterior, `BufferSize` se trata simplemente de una parte `int` inferior, sin datos adicionales ni de las ventajas del sistema de tipos, además de lo que `int` ya tiene.

Un enfoque alternativo al uso de las abreviaturas de tipo para representar un dominio es utilizar uniones discriminadas de un solo caso. El ejemplo anterior se puede modelar como sigue:

```
type BufferSize = BufferSize of int
```

Si escribes código que funciona en términos de `BufferSize` y su valor subyacente, debe construir uno en lugar de pasarlo en cualquier entero arbitrario:

```
module Networking =
    ...
    let send data (BufferSize size) =
    ...
```

Esto reduce la probabilidad de pasar erróneamente un entero arbitrario a la `send` función, ya que el llamador debe construir un `BufferSize` tipo para encapsular un valor antes de llamar a la función.

# Instrucciones de diseño de componentes de F#

13/05/2020 • 54 minutes to read • [Edit Online](#)

Este documento es un conjunto de directrices de diseño de componentes para la programación en F #, en función de las instrucciones de diseño de componentes de F #, V14, Microsoft Research y una versión que originalmente se seleccionada y mantiene con F # Software Foundation.

En este documento se supone que está familiarizado con la programación en F #. Muchas gracias a la comunidad de F # por sus contribuciones e información útil sobre las distintas versiones de esta guía.

## Información general

En este documento se analizan algunos de los problemas relacionados con el diseño y la codificación de componentes de F #. Un componente puede significar cualquiera de los siguientes:

- Una capa del proyecto de F # que tiene consumidores externos dentro de ese proyecto.
- Biblioteca diseñada para el consumo por código de F # a través de los límites del ensamblado.
- Biblioteca diseñada para su consumo por cualquier lenguaje .NET a través de los límites de los ensamblados.
- Biblioteca diseñada para su distribución a través de un repositorio de paquetes, como [NuGet](#).

Las técnicas descritas en este artículo siguen los [cinco principios de buen código de F #](#), por tanto, usan la programación funcional y de objetos según corresponda.

Independientemente de la metodología, el diseñador de componentes y bibliotecas se enfrenta a una serie de problemas prácticos y prosaicos al intentar crear una API que es más fácil de usar para los desarrolladores. La aplicación Conscientious de las instrucciones de diseño de la [biblioteca de .net](#) le dirigirá a la creación de un conjunto coherente de API que son agradables de consumir.

## Directrices generales

Hay algunas directrices universales que se aplican a las bibliotecas de F #, independientemente de la audiencia prevista para la biblioteca.

### Obtener información sobre las directrices de diseño de la biblioteca .NET

Independientemente del tipo de código de F # que esté haciendo, es útil tener conocimientos prácticos de las instrucciones de diseño de la [biblioteca de .net](#). La mayoría de los programadores de F # y .NET estarán familiarizados con estas directrices y esperamos que el código .NET se ajuste a ellos.

Las instrucciones de diseño de la biblioteca de .NET proporcionan instrucciones generales sobre la nomenclatura, el diseño de clases e interfaces, el diseño de miembros (propiedades, métodos, eventos, etc.) y otros, y son un primer punto de referencia útil para una variedad de instrucciones de diseño.

### Agregar comentarios de documentación XML al código

La documentación XML de las API públicas garantiza que los usuarios puedan obtener grandes IntelliSense y QuickInfo al usar estos tipos y miembros, y habilitar la creación de archivos de documentación para la biblioteca. Consulte la [documentación XML](#) sobre las distintas etiquetas XML que se pueden usar para el marcado adicional en los comentarios de XmlDoc.



```

/// A class for representing (x,y) coordinates
type Point =

    /// Computes the distance between this point and another
    member DistanceTo: otherPoint:Point -> float

```

Puede usar los comentarios XML de forma corta ( `/// comment` ) o los comentarios XML estándar ( `///<summary>comment</summary>` ).

## Considere la posibilidad de usar archivos de signatura explícitos (. FSI) para la biblioteca estable y las API de componentes

El uso de archivos de firmas explícitos en una biblioteca de F # proporciona un resumen conciso de la API pública, lo que ayuda a garantizar que conoce la superficie pública completa de la biblioteca y proporciona una separación limpia entre la documentación pública y los detalles internos de la implementación. Los archivos de signatura agregan fricción al cambio de la API pública, ya que requieren que se realicen cambios en los archivos de implementación y de firma. Como resultado, los archivos de firma normalmente solo deben introducirse cuando una API se ha contratado y ya no se espera que cambie de forma significativa.

## Siga siempre los procedimientos recomendados para el uso de cadenas en .NET

Siga las [prácticas recomendadas para usar cadenas en la guía de .net](#) . En concreto, indique siempre explícitamente el *objetivo cultural* en la conversión y la comparación de cadenas (si procede).

## Directrices para las bibliotecas orientadas a F #

En esta sección se presentan recomendaciones para el desarrollo de bibliotecas públicas con conexión a F #. es decir, las bibliotecas exponen las API públicas que están pensadas para que las usen los desarrolladores de F #. Hay una variedad de recomendaciones de diseño de biblioteca que se aplican específicamente a F #. En ausencia de las recomendaciones específicas que se indican a continuación, las instrucciones de diseño de la biblioteca de .NET son la guía de reserva.

### Convenciones de nomenclatura

#### Usar convenciones de mayúsculas y minúsculas de nomenclatura .NET

En la tabla siguiente se siguen las convenciones de nomenclatura y capitalización de .NET. Hay pequeñas adiciones que también incluyen construcciones de F #.

CONSTRUCCIÓN	CASO	PARTE	EJEMPLOS	NOTAS
Tipos concretos	PascalCase	Nombre/Adjetivo	Lista, doble, complejo	Los tipos concretos son Structs, clases, enumeraciones, delegados, registros y uniones. Aunque los nombres de tipo están tradicionalmente en minúsculas en OCaml, F # ha adoptado el esquema de nomenclatura de .NET para los tipos.
DLL	PascalCase		Fabrikam. Core. dll	

CONSTRUCCIÓN	CASO	PARTE	EJEMPLOS	NOTAS
Etiquetas de Unión	PascalCase	Nombre	Algunos, agregar, correcto	<p>No use un prefijo en las API públicas. Opcionalmente, use un prefijo cuando sea interno, como</p> <pre>type Teams =   TAlpha   TBeta   TDelta.</pre>
Evento	PascalCase	Verbo	ValueChanged/Value Changing	
Excepciones	PascalCase		WebException	El nombre debe terminar con "Exception".
Campo	PascalCase	Nombre	CurrentName	
Tipos de interfaz	PascalCase	Nombre/Adjetivo	IDisposable	El nombre debe comenzar con "I".
Método	PascalCase	Verbo	ToString	
Espacio de nombres	PascalCase		Microsoft.FSharp.Core	<p>Por lo general</p> <pre>&lt;Organization&gt;. &lt;Technology&gt;[. &lt;Subnamespace&gt;]</pre> <p>, use, aunque Quite la organización si la tecnología es independiente de la organización.</p>
Parámetros	camelCase	Nombre	typeName, transformación, intervalo	
permitir valores (interno)	camelCase o PascalCase	Nombre/verbo	getValue, MyTable	
permitir valores (externos)	camelCase o PascalCase	Nombre/verbo	List. map, fechas. hoy	los valores de Let enlazados suelen ser públicos cuando se siguen los patrones de diseño funcionales tradicionales. Sin embargo, por lo general, se usa PascalCase cuando el identificador se puede usar desde otros lenguajes .NET.

CONSTRUCCIÓN	CASO	PARTE	EJEMPLOS	NOTAS
Propiedad	PascalCase	Nombre/Adjetivo	IsEndOfFile, BackColor	Normalmente, las propiedades booleanas usan <code>Is</code> y pueden y deben ser afirmativas, como en <code>IsEndOfFile</code> , no <code>IsNotEndOfFile</code> .

#### Evitar abreviaturas

Las instrucciones de .NET desaconsejan el uso de abreviaturas (por ejemplo, "usar `OnClick`" en lugar de `OnBtnClick`). Se toleran las abreviaturas comunes, como `Async` "asíncronas". A veces, esta instrucción se omite en la programación funcional; por ejemplo, `List.iter` usa una abreviatura para "iterar". Por esta razón, el uso de abreviaturas tiende a tolerar un mayor grado en la programación de F # a F #, pero normalmente se debe evitar en el diseño de componentes públicos.

#### Evitar conflictos de nombres con mayúsculas y minúsculas

Las directrices de .NET indican que no se puede usar con mayúsculas y minúsculas para eliminar la ambigüedad de los conflictos de nombres, ya que algunos lenguajes de cliente (por ejemplo, Visual Basic) no distinguen mayúsculas de minúsculas.

#### Usar acrónimos cuando corresponda

Los acrónimos como XML no son abreviaturas y se usan ampliamente en las bibliotecas de .NET en formato inversado (XML). Solo se deben usar acrónimos conocidos y ampliamente reconocidos.

#### Usar PascalCase para nombres de parámetros genéricos

Use PascalCase para los nombres de parámetros genéricos en las API públicas, incluidas las bibliotecas orientadas a F #. En concreto, use nombres como `T`, `U`, `T1`, `T2` para los parámetros genéricos arbitrarios y, cuando tengan sentido los nombres específicos, en el caso de las bibliotecas orientadas a F #, use nombres como `Key`, `Value`, `Arg` (pero no por ejemplo `TKey`).

#### Uso de PascalCase o camelCase para funciones y valores públicos en módulos de F #

camelCase se usa para las funciones públicas que están diseñadas para usarse sin calificar (por ejemplo, `invalidArg`) y para las "funciones de colección estándar" (por ejemplo, `List.map`). En ambos casos, los nombres de función actúan de forma muy parecida a las palabras clave en el lenguaje.

### Diseño de objetos, tipos y módulos

#### Usar espacios de nombres o módulos para contener los tipos y módulos

Cada archivo de F # de un componente debe comenzar con una declaración de espacio de nombres o una declaración de módulo.

```
namespace Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
    ...

module CommonOperations =
    ...
```

or

```

module Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
    ...

module CommonOperations =
    ...

```

Las diferencias entre el uso de módulos y espacios de nombres para organizar el código en el nivel superior son las siguientes:

- Los espacios de nombres pueden abarcar varios archivos
- Los espacios de nombres no pueden contener funciones de F # a menos que estén dentro de un módulo interno
- El código de cualquier módulo determinado debe estar incluido en un único archivo.
- Los módulos de nivel superior pueden contener funciones de F # sin necesidad de un módulo interno

La elección entre un módulo o un espacio de nombres de nivel superior afecta al formulario compilado del código y, por tanto, afectará a la vista de otros lenguajes .NET en los casos en que la API se consuma fuera del código de F #.

#### Usar métodos y propiedades para operaciones intrínsecas a tipos de objeto

Al trabajar con objetos, es mejor asegurarse de que la funcionalidad consumible se implementa como métodos y propiedades en ese tipo.

```

type HardwareDevice() =

    member this.ID = ...

    member this.SupportedProtocols = ...

type HashTable<'Key, 'Value>(comparer: IEqualityComparer<'Key>) =

    member this.Add(key, value) = ...

    member this.ContainsKey(key) = ...

    member this.ContainsValue(value) = ...

```

No es necesario implementar la mayor parte de la funcionalidad para un miembro determinado en ese miembro, pero la parte consumible de esa funcionalidad debe ser.

#### Usar clases para encapsular el estado mutable

En F #, esto solo se debe hacer si el estado no está encapsulado por otra construcción de lenguaje, como un cierre, una expresión de secuencia o un cálculo asíncrono.

```

type Counter() =
    // let-bound values are private in classes.
    let mutable count = 0

    member this.Next() =
        count <- count + 1
        count

```

#### Usar interfaces para agrupar operaciones relacionadas

Use los tipos de interfaz para representar un conjunto de operaciones. Esto es preferible a otras opciones, como tuplas de funciones o registros de funciones.

```
type Serializer =  
  abstract Serialize<'T>: preserveRefEq: bool -> value: 'T -> string  
  abstract Deserialize<'T>: preserveRefEq: bool -> pickle: string -> 'T
```

En preferencia a:

```
type Serializer<'T> = {  
  Serialize: bool -> 'T -> string  
  Deserialize: bool -> string -> 'T  
}
```

Las interfaces son conceptos de primera clase en .NET, que puede usar para lograr lo que normalmente le daría. Además, se pueden usar para codificar tipos existencial en el programa, que no pueden tener registros de funciones.

#### Usar un módulo para agrupar funciones que actúan en colecciones

Al definir un tipo de colección, considere la posibilidad de proporcionar un conjunto estándar de operaciones como `CollectionType.map` y `CollectionType.iter` ) para los nuevos tipos de colección.

```
module CollectionType =  
  let map f c =  
    ...  
  let iter f c =  
    ...
```

Si incluye este módulo, siga las convenciones de nomenclatura estándar para las funciones que se encuentran en FSharp. Core.

#### Usar un módulo para agrupar funciones para funciones canónicas comunes, especialmente en bibliotecas matemáticas y DSL

Por ejemplo, `Microsoft.FSharp.Core.Operators` es una colección abierta automáticamente de funciones de nivel superior (como `abs` y `sin` ) proporcionada por FSharp. Core. dll.

Del mismo modo, una biblioteca de estadísticas podría incluir un módulo con funciones `erf` y `erfc` , donde este módulo está diseñado para que se abra explícita o automáticamente.

#### Considere la posibilidad de usar `RequireQualifiedAccess` y aplicar cuidadosamente los atributos `AutoOpen`

Agregar el `[<RequireQualifiedAccess>]` atributo a un módulo indica que el módulo no se puede abrir y que las referencias a los elementos del módulo requieren un acceso calificado explícito. Por ejemplo, el `Microsoft.FSharp.Collections.List` módulo tiene este atributo.

Esto resulta útil cuando las funciones y los valores del módulo tienen nombres que es probable que entren en conflicto con los nombres de otros módulos. Requerir acceso cualificado puede aumentar considerablemente el mantenimiento a largo plazo y la evolución de una biblioteca.

Agregar el `[<AutoOpen>]` atributo a un módulo significa que el módulo se abrirá cuando se abra el espacio de nombres que lo contiene. El `[<AutoOpen>]` atributo también se puede aplicar a un ensamblado para indicar un módulo que se abre automáticamente cuando se hace referencia al ensamblado.

Por ejemplo, una biblioteca de estadísticas **MathsHeaven. Statistics** puede contener una

```
module MathsHeaven.Statistics.Operators
```

 función contenedora `erf` y `erfc` . Es razonable marcar este módulo como `[<AutoOpen>]` . Esto significa `open MathsHeaven.Statistics` que también abrirá este módulo y incluirá los nombres `erf` y `erfc` en el ámbito. Otro uso adecuado de `[<AutoOpen>]` es para los módulos que contienen métodos de extensión.

[<AutoOpen>] El uso excesivo de los clientes potenciales en los espacios de nombres contaminados y el atributo debe usarse con cuidado. En el caso de bibliotecas específicas de dominios específicos, el uso prudente de [AutoOpen] puede dar lugar a una mejor facilidad de uso.

**Considere la posibilidad de definir miembros de operador en clases en las que el uso de operadores Well-Knows sea adecuado.**

A veces, las clases se utilizan para modelar construcciones matemáticas como vectores. Cuando el dominio que se está modelando tiene operadores conocidos, es útil definirlos como miembros intrínsecos de la clase.

```
type Vector(x: float) =  
  
    member v.X = x  
  
    static member (*) (vector: Vector, scalar: float) = Vector(vector.X * scalar)  
  
    static member (+) (vector1: Vector, vector2: Vector) = Vector(vector1.X + vector2.X)  
  
let v = Vector(5.0)  
  
let u = v * 10.0
```

Esta guía se corresponde con las instrucciones generales de .NET para estos tipos. Sin embargo, también puede ser importante en la codificación de F #, ya que permite usar estos tipos junto con funciones y métodos de F # con restricciones de miembro, como List.sumBy (.

**Considere el uso de CompiledName (para proporcionar un. Nombre descriptivo para otros consumidores del lenguaje .NET**

En ocasiones, es posible que desee asignar un nombre a algo en un estilo para los consumidores de F # (por ejemplo, un miembro estático en minúsculas para que aparezca como si fuera una función enlazada a un módulo), pero tener un estilo diferente para el nombre cuando se compila en un ensamblado. Puede usar el [CompiledName] atributo para proporcionar un estilo diferente para el código que no es de F # que usa el ensamblado.

```
type Vector(x:float, y:float) =  
  
    member v.X = x  
    member v.Y = y  
  
    [CompiledName("Create")]  
    static member create x y = Vector (x, y)  
  
let v = Vector.create 5.0 3.0
```

Mediante el uso de [CompiledName] , puede usar las convenciones de nomenclatura de .net para los consumidores que no son de F # del ensamblado.

**Usar la sobrecarga de métodos para las funciones miembro, si esto proporciona una API más sencilla**

La sobrecarga de métodos es una herramienta eficaz para simplificar una API que puede necesitar realizar una funcionalidad similar, pero con diferentes opciones o argumentos.

```
type Logger() =  
  
    member this.Log(message) =  
        ...  
    member this.Log(message, retryPolicy) =  
        ...
```

En F #, es más común sobrecargar el número de argumentos en lugar de los tipos de argumentos.

**Ocultar las representaciones de los tipos de registro y Unión si es probable que evolucione el diseño de estos tipos**

Evite revelar representaciones concretas de objetos. Por ejemplo, la [DateTime](#) API pública externa del diseño de la biblioteca de .net no revela la representación concreta de los valores. En tiempo de ejecución, Common Language Runtime conoce la implementación confirmada que se utilizará a lo largo de la ejecución. Sin embargo, el código compilado no selecciona las dependencias en la representación concreta.

#### Evitar el uso de la herencia de implementación para la extensibilidad

En F #, rara vez se utiliza la herencia de implementación. Además, las jerarquías de herencia suelen ser complejas y difíciles de cambiar cuando llegan nuevos requisitos. La implementación de la herencia todavía existe en F # por compatibilidad y casos raros en los que es la mejor solución para un problema, pero se deben buscar técnicas alternativas en los programas de F # al diseñar el polimorfismo, como la implementación de la interfaz.

#### Firmas de función y de miembro

##### Usar tuplas para valores devueltos al devolver un número pequeño de varios valores no relacionados

Este es un buen ejemplo del uso de una tupla en un tipo de valor devuelto:

```
val divrem: BigInteger -> BigInteger -> BigInteger * BigInteger
```

En el caso de los tipos de valor devueltos que contienen muchos componentes, o en los que los componentes están relacionados con una sola entidad identificable, considere la posibilidad de usar un tipo con nombre en lugar de una tupla.

##### Se usa `Async<T>` para la programación asíncrona en los límites de la API de F #

Si hay una operación sincrónica correspondiente denominada `Operation` que devuelve un `T`, se debe asignar un nombre a la operación asíncrona `AsyncOperation` si devuelve `Async<T>` o `OperationAsync` si devuelve `Task<T>`. En el caso de los tipos de .NET que se usan habitualmente y que exponen métodos Begin/end, considere la posibilidad de usar `Async.FromBeginEnd` para escribir métodos de extensión como una fachada para proporcionar el modelo de programación de F # Async a esas API de .net.

```
type SomeType =
    member this.Compute(x:int): int =
        ...
    member this.AsyncCompute(x:int): Async<int> =
        ...

type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        ...
```

#### Excepciones

Consulte [Administración de errores](#) para obtener información sobre el uso adecuado de excepciones, resultados y opciones.

#### Miembros de extensión

##### Aplicar cuidadosamente los miembros de extensión de F # en componentes de F # a-F #

Normalmente, los miembros de extensión de F # solo se deben usar para las operaciones que se encuentran en el cierre de operaciones intrínsecas asociadas a un tipo en la mayoría de los modos de uso. Un uso común es proporcionar API más idiomático a F # para varios tipos de .NET:

```

type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        Async.FromBeginEnd(this.BeginReceive, this.EndReceive)

type System.Collections.Generic.IDictionary<'Key, 'Value> with
    member this.TryGet key =
        let ok, v = this.TryGetValue key
        if ok then Some v else None

```

## Tipos de Unión

### Usar uniones discriminadas en lugar de jerarquías de clases para los datos estructurados por árbol

Las estructuras de tipo árbol se definen de forma recursiva. Esto es complicado con la herencia, pero elegante con uniones discriminadas.

```

type BST<'T> =
    | Empty
    | Node of 'T * BST<'T> * BST<'T>

```

Representar datos similares a los árboles con uniones discriminadas también le permite beneficiarse de la en la coincidencia de patrones.

### Usar `[<RequireQualifiedAccess>]` en tipos de Unión cuyos nombres de caso no son suficientemente únicos

Puede que se encuentre en un dominio en el que el mismo nombre sea el mejor para cosas diferentes, como casos de Unión discriminada. Puede usar para eliminar la `[<RequireQualifiedAccess>]` ambigüedad de los nombres de mayúsculas y minúsculas con el fin de evitar que se produzcan errores confusos debido al sombreado dependiente del orden de las `open` instrucciones.

### Ocultar las representaciones de las uniones discriminadas para las API compatibles con binario si es probable que evolucione el diseño de estos tipos

Los tipos uniones se basan en formularios de coincidencia de patrones de F # para un modelo de programación conciso. Como se mencionó anteriormente, debe evitar revelar representaciones de datos concretas si es probable que evolucione el diseño de estos tipos.

Por ejemplo, la representación de una Unión discriminada se puede ocultar mediante una declaración privada o interna, o mediante un archivo de signature.

```

type Union =
    private
    | CaseA of int
    | CaseB of string

```

Si revela las uniones discriminadas indiscriminadamente, puede que le resulte difícil crear una versión de la biblioteca sin interrumpir el código de usuario. En su lugar, considere la posibilidad de mostrar uno o varios patrones activos para permitir la coincidencia de patrones con los valores de su tipo.

Los modelos activos proporcionan una forma alternativa de proporcionar a los consumidores de F # la coincidencia de patrones, al mismo tiempo que evitan exponer directamente los tipos de unión de F #.

## Funciones insertadas y restricciones de miembro

### Definir algoritmos numéricos genéricos mediante funciones insertadas con restricciones de miembros implícitas y tipos genéricos resueltos estáticamente

Las restricciones de miembros aritméticos y las restricciones de comparación de F # son un estándar para la programación en F #. Por ejemplo, considere el siguiente código:



```
let inline highestCommonFactor a b =
    let rec loop a b =
        if a = LanguagePrimitives.GenericZero<_> then b
        elif a < b then loop a (b - a)
        else loop (a - b) b
    loop a b
```

El tipo de esta función es el siguiente:

```
val inline highestCommonFactor : ^T -> ^T -> ^T
    when ^T : (static member Zero : ^T)
    and ^T : (static member ( - ) : ^T * ^T -> ^T)
    and ^T : equality
    and ^T : comparison
```

Se trata de una función adecuada para una API pública en una biblioteca matemática.

### **Evite el uso de restricciones de miembro para simular clases de tipos y tipos de patos**

Es posible simular "Duck Typing" con las restricciones de miembro de F#. Sin embargo, los miembros que hacen uso de esto no deben usarse en general en los diseños de la biblioteca de F# a-F#. Esto se debe a que los diseños de biblioteca basados en restricciones implícitas no conocidas o no estándar tienden a hacer que el código de usuario sea inflexible y vinculado a un patrón de marco de trabajo determinado.

Además, existe la posibilidad de que el uso intensivo de restricciones de miembros de esta manera pueda producir tiempos de compilación muy largos.

## **Definiciones de operador**

### **Evite definir operadores simbólicos personalizados**

Los operadores personalizados son esenciales en algunas situaciones y son dispositivos de notación muy útiles en un gran número de código de implementación. En el caso de los nuevos usuarios de una biblioteca, las funciones con nombre suelen ser más fáciles de usar. Además, los operadores simbólicos personalizados pueden ser difíciles de documentar y los usuarios le resultarán más difíciles de buscar ayuda sobre los operadores, debido a las limitaciones existentes en el IDE y los motores de búsqueda.

Como resultado, es mejor publicar su funcionalidad como funciones y miembros con nombre y, además, exponer operadores para esta funcionalidad solo si las ventajas de la notación son más importantes que la documentación y el costo cognitivo de tenerlos.

## **Unidades de medida**

### **Usar cuidadosamente unidades de medida para la seguridad de tipos agregada en código de F#**

La información de escritura adicional para las unidades de medida se borra cuando la ven otros lenguajes de .NET. Tenga en cuenta que los componentes, las herramientas y la reflexión de .NET verán tipos-sans-units. Por ejemplo, los consumidores de C# verán `float` en lugar de `float<kg>`.

## **Abreviaturas de tipo**

### **Usar cuidadosamente las abreviaturas de tipo para simplificar el código de F#**

Los componentes, las herramientas y la reflexión de .NET no verán los nombres abreviados de los tipos. El uso significativo de las abreviaturas de tipo también puede hacer que un dominio parezca más complejo de lo que realmente es, lo que puede confundir a los consumidores.

### **Evite las abreviaturas de tipo para los tipos públicos cuyos miembros y propiedades deben ser intrínsecamente diferentes a los disponibles en el tipo que se va a abreviar.**

En este caso, el tipo que se va a abreviar revela demasiada información sobre la representación del tipo real que se está definiendo. En su lugar, considere la posibilidad de ajustar la abreviatura en un tipo de clase o en una Unión discriminada de un caso (o, cuando el rendimiento sea esencial, considere la posibilidad de usar un tipo de struct para encapsular la abreviatura).

Por ejemplo, es tentador definir un mapa múltiple como un caso especial de un mapa de F #, por ejemplo:

```
type MultiMap<'Key, 'Value> = Map<'Key, 'Value list>
```

Sin embargo, las operaciones de notación de puntos lógicas de este tipo no son las mismas que las operaciones en un mapa; por ejemplo, es razonable que la asignación del operador de búsqueda. [Key] devuelve la lista vacía si la clave no está en el diccionario, en lugar de generar una excepción.

## Directrices para las bibliotecas para su uso desde otros lenguajes .NET

Al diseñar bibliotecas para su uso desde otros lenguajes .NET, es importante cumplir las directrices de [diseño](#) de la biblioteca de .net. En este documento, estas bibliotecas se etiquetan como bibliotecas de .NET de vainilla, a diferencia de las bibliotecas de F # orientadas a las que usan construcciones de F # sin restricciones. El diseño de las bibliotecas .NET de vainilla permite proporcionar API conocidas y idiomáticas coherentes con el resto de los .NET Framework al minimizar el uso de construcciones específicas de F # en la API pública. Las reglas se explican en las secciones siguientes.

### Espacio de nombres y diseño de tipos (para bibliotecas para su uso desde otros lenguajes .NET)

#### Aplicar las convenciones de nomenclatura de .NET a la API pública de los componentes

Preste especial atención al uso de nombres abreviados y a las directrices de capitalización de .NET.

```
type pCoord = ...
    member this.theta = ...

type PolarCoordinate = ...
    member this.Theta = ...
```

#### Usar espacios de nombres, tipos y miembros como la estructura organizativa principal de los componentes

Todos los archivos que contienen funcionalidad pública deben comenzar con una `namespace` declaración y las únicas entidades de acceso público de los espacios de nombres deben ser tipos. No use módulos de F #.

Use módulos no públicos para almacenar el código de implementación, los tipos de utilidad y las funciones de utilidad.

Los tipos estáticos deben ser preferibles a los módulos, ya que permiten la evolución futura de la API para usar la sobrecarga y otros conceptos de diseño de la API de .NET que no se pueden usar en los módulos de F #.

Por ejemplo, en lugar de la siguiente API pública:

```
module Fabrikam

module Utilities =
    let Name = "Bob"
    let Add2 x y = x + y
    let Add3 x y z = x + y + z
```

En su lugar, considere:

```
namespace Fabrikam

[<AbstractClass; Sealed>]
type Utilities =
    static member Name = "Bob"
    static member Add(x,y) = x + y
    static member Add(x,y,z) = x + y + z
```

### Usar tipos de registro de F # en las API de .NET de vainilla si el diseño de los tipos no evolucionará

Los tipos de registro de F # se compilan en una clase .NET simple. Son adecuadas para algunos tipos simples y estables en las API. Considere la posibilidad de usar los `[<NoEquality>]` `[<NoComparison>]` atributos y para suprimir la generación automática de interfaces. Evite también el uso de campos de registro mutables en las API de .NET de vainilla, ya que exponen un campo público. Considere siempre si una clase proporcionaría una opción más flexible para la evolución futura de la API.

Por ejemplo, el siguiente código de F # expone la API pública a un consumidor de C#:

F#:

```
[<NoEquality; NoComparison>]
type MyRecord =
    { FirstThing: int
      SecondThing: string }
```

C#:

```
public sealed class MyRecord
{
    public MyRecord(int firstThing, string secondThing);
    public int FirstThing { get; }
    public string SecondThing { get; }
}
```

### Ocultar la representación de tipos de unión de F # en las API de .NET de vainilla

Los tipos de unión de f # no se usan habitualmente en los límites de los componentes, ni siquiera para la codificación de F # a F #. Son un dispositivo de implementación excelente cuando se usan internamente en componentes y bibliotecas.

Al diseñar una API de .NET de vainilla, considere la posibilidad de ocultar la representación de un tipo de Unión mediante una declaración privada o un archivo de signatura.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True
```

También puede aumentar los tipos que usan internamente una representación de unión con los miembros para proporcionar el deseado. API orientada a redes.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True

    /// A public member for use from C#
    member x.Evaluate =
        match x with
        | And(a,b) -> a.Evaluate && b.Evaluate
        | Not a -> not a.Evaluate
        | True -> true

    /// A public member for use from C#
    static member CreateAnd(a,b) = And(a,b)
```

## Diseñar GUI y otros componentes mediante los modelos de diseño del marco de trabajo

Hay muchos marcos de trabajo diferentes disponibles en .NET, como WinForms, WPF y ASP.NET. Las convenciones de nomenclatura y diseño de cada una deben usarse si se diseñan componentes para su uso en estos marcos de trabajo. Por ejemplo, para la programación de WPF, adopte patrones de diseño de WPF para las clases que está diseñando. En el caso de los modelos de programación de la interfaz de usuario, utilice patrones de diseño como eventos y colecciones basadas en notificaciones, como las que se encuentran en [System.Collections.ObjectModel](#).

## Diseño de objetos y miembros (para bibliotecas para su uso desde otros lenguajes .NET)

### Usar el atributo CLIEvent para exponer eventos de .NET

Construya un `DelegateEvent` con un tipo de delegado .net específico que toma un objeto y `EventArgs` (en lugar de un `Event`), que simplemente usa el `FSharpHandler` tipo de forma predeterminada) para que los eventos se publiquen de la manera más familiar que otros lenguajes .net.

```
type MyBadType() =
    let myEv = new Event<int>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish

type MyEventArgs(x: int) =
    inherit System.EventArgs()
    member this.X = x

    /// A type in a component designed for use from other .NET languages
type MyGoodType() =
    let myEv = new DelegateEvent<EventHandler<MyEventArgs>>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish
```

### Exponer operaciones asíncronas como métodos que devuelven tareas de .NET

Las tareas se usan en .NET para representar los cálculos asíncronos activos. En general, las tareas son menos compuestas que los objetos de F # `Async<T>`, ya que representan las tareas que ya se están ejecutando y no se pueden componer juntas de forma que realicen la composición paralela, o que oculten la propagación de señales de cancelación y otros parámetros contextuales.

Sin embargo, a pesar de ello, los métodos que devuelven tareas son la representación estándar de la programación asíncrona en .NET.

```
/// A type in a component designed for use from other .NET languages
type MyType() =

    let compute (x: int): Async<int> = async { ... }

    member this.ComputeAsync(x) = compute x |> Async.StartAsTask
```

Normalmente, también querrá aceptar un token de cancelación explícito:

```
/// A type in a component designed for use from other .NET languages
type MyType() =
    let compute(x: int): Async<int> = async { ... }
    member this.ComputeAsTask(x, cancellationToken) = Async.StartAsTask(compute x, cancellationToken)
```

### Usar tipos de delegado de .NET en lugar de tipos de función de F #

Aquí "tipos de funciones de F #" significan tipos de "flecha" como `int -> int`.

En lugar de esto:

```
member this.Transform(f: int->int) =
    ...
```

Haga esto:

```
member this.Transform(f: Func<int,int>) =
    ...
```

El tipo de función de F # aparece como `class FSharpFunc<T,U>` en otros lenguajes .net y es menos adecuado para las características del lenguaje y las herramientas que comprenden los tipos de delegado. Al crear un método de orden superior que tenga como destino .NET Framework 3,5 o superior, `System.Func` los `System.Action` delegados y son las API adecuadas para publicar y permitir a los desarrolladores de .net usar estas API de manera insuficiente. (Cuando el destino es .NET Framework 2,0, los tipos de delegado definidos por el sistema son más limitados; considere la posibilidad de usar tipos de delegado predefinidos como `System.Converter<T,U>` o definir un tipo de delegado específico).

En el lado de volteo, los delegados de .NET no son naturales para las bibliotecas orientadas a F # (vea la siguiente sección sobre bibliotecas orientadas a F #). Como resultado, una estrategia de implementación común al desarrollar métodos de orden superior para las bibliotecas de .NET de vainilla es crear toda la implementación con los tipos de función de F # y, a continuación, crear la API pública mediante delegados como una fachada fina sobre la implementación real de F #.

**Use el patrón TryGetValue en lugar de devolver valores de opción de F # y preferir la sobrecarga del método para tomar los valores de las opciones de F # como argumentos.**

Los patrones comunes de uso para el tipo de opción de F # en las API se implementan mejor en las API de .NET de vainilla mediante técnicas de diseño estándar de .NET. En lugar de devolver un valor de opción de F #, considere la posibilidad de usar el tipo de valor devuelto bool junto con un parámetro out como en el patrón "TryGetValue". Y en lugar de tomar los valores de las opciones de F # como parámetros, considere la posibilidad de usar la sobrecarga de métodos o argumentos opcionales.

```
member this.ReturnOption() = Some 3

member this.ReturnBoolAndOut(outVal: byref<int>) =
    outVal <- 3
    true

member this.ParamOption(x: int, y: int option) =
    match y with
    | Some y2 -> x + y2
    | None -> x

member this.ParamOverload(x: int) = x

member this.ParamOverload(x: int, y: int) = x + y
```

**Usar los tipos de interfaz de colección de .NET IEnumerable < T > y la < clave IDictionary, el valor > para los parámetros y los valores devueltos**

Evite el uso de tipos de colección concretos como matrices de .NET `T[]`, tipos de F # y `list<T>` `Map<Key,Value>` `Set<T>`, y tipos de colección concretos de .net como `Dictionary<Key,Value>`. Las instrucciones de diseño de la biblioteca de .NET tienen buenos consejos sobre Cuándo usar varios tipos de colección como `IEnumerable<T>`. El uso de matrices ( `T[]` ) es aceptable en algunas circunstancias, por motivos de rendimiento. Tenga en cuenta especialmente que `seq<T>` es solo el alias de F # para `IEnumerable<T>` y, por lo tanto, SEQ suele ser un tipo adecuado para una API de .net de vainilla.

En lugar de listas de F #:

```
member this.PrintNames(names: string list) =  
    ...
```

Usar secuencias de F #:

```
member this.PrintNames(names: seq<string>) =  
    ...
```

**Use el tipo de unidad como el único tipo de entrada de un método para definir un método de argumento cero, o como el único tipo de valor devuelto para definir un método que devuelve void.**

Evite otros usos del tipo de unidad. Estos son buenos:

```
✓ member this.NoArguments() = 3  
  
✓ member this.ReturnVoid(x: int) = ()
```

Esto no es válido:

```
member this.WrongUnit( x: unit, z: int) = ((), ())
```

#### Comprobar si hay valores NULL en los límites de la API de .NET de vainilla

El código de implementación de F # tiende a tener menos valores NULL, debido a las restricciones y patrones de diseño inmutables sobre el uso de literales null para los tipos de F #. Otros lenguajes .NET suelen usar NULL como valor con mucha más frecuencia. Por este motivo, el código de F # que expone una API de .NET de vainilla debe comprobar los parámetros null en el límite de la API y evitar que estos valores fluyan más en el código de implementación de F #. `isNotNull` Se puede usar la función o coincidencia de patrones en el `null` patrón.

```
let checkNonNull argName (arg: obj) =  
    match arg with  
    | null -> nullArg argName  
    | _ -> ()  
  
let checkNonNull` argName (arg: obj) =  
    if isNull arg then nullArg argName  
    else ()
```

#### Evite el uso de tuplas como valores devueltos

En su lugar, se prefiere devolver un tipo con nombre que contenga los datos agregados o usar parámetros out para devolver varios valores. Aunque las tuplas de estructuras y de struct existen en .NET (incluida la compatibilidad con el lenguaje C# para las tuplas de struct), a menudo no proporcionarán la API ideal y la esperada para los desarrolladores de .NET.

#### Evitar el uso de curificación de parámetros

En su lugar, use convenciones de llamada de .NET `Method(arg1, arg2, ..., argN)` .

```
member this.TupledArguments(str, num) = String.replicate num str
```

Sugerencia: Si está diseñando bibliotecas para su uso desde cualquier lenguaje .NET, no habrá ningún sustituto para realizar realmente alguna programación experimental de C# y Visual Basic para asegurarse de que las bibliotecas "se sienten bien" desde estos lenguajes. También puede usar herramientas como .NET reflector y el Examinador de objetos de Visual Studio para asegurarse de que las bibliotecas y su documentación aparecen como se espera para los desarrolladores.

# Apéndice

## Ejemplo de un extremo a otro del diseño de código de F # para su uso por parte de otros lenguajes .NET

Considere la siguiente clase:

```
open System

type Point1(angle,radius) =
    new() = Point1(angle=0.0, radius=0.0)
    member x.Angle = angle
    member x.Radius = radius
    member x.Stretch(l) = Point1(angle=x.Angle, radius=x.Radius * l)
    member x.Warp(f) = Point1(angle=f(x.Angle), radius=x.Radius)
    static member Circle(n) =
        [ for i in 1..n -> Point1(angle=2.0*Math.PI/float(n), radius=1.0) ]
```

El tipo de F # deducido de esta clase es el siguiente:

```
type Point1 =
    new : unit -> Point1
    new : angle:double * radius:double -> Point1
    static member Circle : n:int -> Point1 list
    member Stretch : l:double -> Point1
    member Warp : f:(double -> double) -> Point1
    member Angle : double
    member Radius : double
```

Echemos un vistazo a cómo se muestra este tipo de F # a un programador con otro lenguaje .NET. Por ejemplo, la "firma" de C# aproximada es la siguiente:

```
// C# signature for the unadjusted Point1 class
public class Point1
{
    public Point1();

    public Point1(double angle, double radius);

    public static Microsoft.FSharp.Collections.List<Point1> Circle(int count);

    public Point1 Stretch(double factor);

    public Point1 Warp(Microsoft.FSharp.Core.FastFunc<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

Hay algunos aspectos importantes que se deben tener en cuenta sobre cómo F # representa las construcciones aquí. Por ejemplo:

- Se han conservado los metadatos, como los nombres de los argumentos.
- Los métodos de F # que toman dos argumentos se convierten en métodos de C# que toman dos argumentos.
- Las funciones y las listas se convierten en referencias a los tipos correspondientes de la biblioteca de F #.

En el código siguiente se muestra cómo ajustar este código para tener en cuenta estos aspectos.

```

namespace SuperDuperFSharpLibrary.Types

type RadialPoint(angle:double, radius:double) =

    /// Return a point at the origin
    new() = RadialPoint(angle=0.0, radius=0.0)

    /// The angle to the point, from the x-axis
    member x.Angle = angle

    /// The distance to the point, from the origin
    member x.Radius = radius

    /// Return a new point, with radius multiplied by the given factor
    member x.Stretch(factor) =
        RadialPoint(angle=angle, radius=radius * factor)

    /// Return a new point, with angle transformed by the function
    member x.Warp(transform:Func<_,>) =
        RadialPoint(angle=transform.Invoke angle, radius=radius)

    /// Return a sequence of points describing an approximate circle using
    /// the given count of points
    static member Circle(count) =
        seq { for i in 1..count ->
            RadialPoint(angle=2.0*Math.PI/float(count), radius=1.0) }

```

El tipo de F # deducido del código es el siguiente:

```

type RadialPoint =
    new : unit -> RadialPoint
    new : angle:double * radius:double -> RadialPoint
    static member Circle : count:int -> seq<RadialPoint>
    member Stretch : factor:double -> RadialPoint
    member Warp : transform:System.Func<double,double> -> RadialPoint
    member Angle : double
    member Radius : double

```

La firma de C# es ahora como sigue:

```

public class RadialPoint
{
    public RadialPoint();

    public RadialPoint(double angle, double radius);

    public static System.Collections.Generic.IEnumerable<RadialPoint> Circle(int count);

    public RadialPoint Stretch(double factor);

    public RadialPoint Warp(System.Func<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}

```

Las correcciones realizadas para preparar este tipo para su uso como parte de una biblioteca .NET de vainilla son las siguientes:

- Se han ajustado varios nombres: `Point1`, `n`, y se han convertido en `1` `f` `RadialPoint`, `count`, `factor` y `transform`, respectivamente.



- Se usa un tipo de valor devuelto de en `seq<RadialPoint>` lugar de `RadialPoint list` cambiar una construcción de lista mediante `[ ... ]` a una construcción de secuencia mediante `IEnumerable<RadialPoint>` .
- Se usa el tipo de delegado de .NET `System.Func` en lugar de un tipo de función de F #.

Esto hace que sea mucho más agradable usar en el código de C#.

# Uso de F# en Azure

23/07/2020 • 10 minutes to read • [Edit Online](#)

F# es un lenguaje excelente para la programación en la nube y se suele usar para escribir aplicaciones web, servicios en la nube, microservicios hospedados en la nube y para el procesamiento de datos escalables.

En las secciones siguientes encontrará recursos sobre cómo usar varios servicios de Azure con F#.

## NOTE

Si un servicio de Azure determinado no aparece en este conjunto de documentos, consulte la documentación de Azure Functions o .NET para ese servicio. Algunos servicios de Azure son independientes del lenguaje y no requieren ninguna documentación específica del lenguaje y no se muestran aquí.

## Uso de Azure Virtual Machines con F#

Azure admite una amplia gama de configuraciones de máquina virtual (VM), vea [Linux and Azure Virtual Machines](#) (Linux y Azure Virtual Machines).

Para instalar F# en una máquina virtual para ejecución, compilación o scripting, vea [Using F# on Linux](#) (Uso de F# en Linux) y [Using F# on Windows](#) (Uso de F# en Windows).

## Uso de Azure Functions con F#

[Azure Functions](#) es una solución para ejecutar fácilmente pequeños fragmentos de código, o "funciones", en la nube. Se puede escribir simplemente el código necesario para el problema en cuestión, sin preocuparse por toda la aplicación o la infraestructura para ejecutarlo. Las funciones se conectan a los eventos en el almacenamiento de Azure y otros recursos hospedados en la nube. Los datos fluyen a las funciones de F# a través de argumentos de función. Puede usar el lenguaje que prefiera y confiar en Azure para escalar según las necesidades.

Azure Functions admite F# como lenguaje de primera clase con ejecución reactiva, eficaz y escalable de código de F#. Vea la [Referencia para desarrolladores de F# de Azure Functions](#) para obtener documentación de referencia sobre cómo usar F# con Azure Functions.

Otros recursos para usar Azure Functions y F#:

- [Scale Up Azure Functions in F# Using Suave](#) (Escalar Azure Functions en F# con Suave)
- [How to create Azure function in F#](#) (Cómo crear Azure Functions en F#)
- [Uso del proveedor de tipos de Azure con Azure Functions](#)

## Uso de Azure Storage con F#

Azure Storage es una capa base de servicios de almacenamiento para aquellas aplicaciones modernas que necesitan durabilidad, disponibilidad y escalabilidad para satisfacer las necesidades de los clientes. Los programas de F# pueden interactuar directamente con los servicios de almacenamiento de Azure usando las técnicas que se describen en los siguientes artículos.

- [Introducción a Azure Blob Storage mediante F#](#)
- [Introducción a Azure File Storage mediante F#](#)
- [Introducción a Azure Queue Storage mediante F#](#)
- [Introducción a Azure Table Storage mediante F#](#)

Azure Storage también puede usarse junto con Azure Functions a través de configuración declarativa en lugar de llamadas de API explícitas. Vea [Desencadenadores y enlaces de Azure Functions para Azure Storage](#) que incluye ejemplos de F#.

## Uso de Azure App Service con F#

[Azure App Service](#) es una plataforma en la nube para crear aplicaciones web y móviles eficaces que se conectan a los datos en cualquier lugar, en la nube o de forma local.

- [Ejemplo de API web de Azure de F#](#)
- [Hospedaje de F# en una aplicación web en Azure](#)

## Uso de Apache Spark con F# con Azure HDInsight

[Apache Spark para Azure HDInsight](#) es una plataforma de procesamiento de código abierto que ejecuta aplicaciones de análisis de datos a gran escala. Azure hace que Apache Spark sea fácil y rentable de implementar. Desarrolle su aplicación de Spark en F# con [Mobius](#), una API de .NET para Spark.

- [Implementing Spark Apps in F# using Mobius](#) (Implementar aplicaciones de Spark en F# con Mobius)
- [Aplicaciones de Spark de F# de ejemplo con Mobius](#)

## Uso de Azure Cosmos DB con .NET#

[Azure Cosmos DB](#) es un servicio NoSQL para aplicaciones altamente disponibles y distribuidas de forma global.

Azure Cosmos DB se puede usar con F# de dos maneras:

1. Mediante la creación de Azure Functions de F# que reaccionen ante los cambios o los provoquen en colecciones de Cosmos DB. Consulte [Enlaces de Azure Cosmos DB para Azure Functions](#).
2. Use el [SDK de .NET para Azure Cosmos DB para la API de SQL](#). Los ejemplos relacionados están en C#.

## Uso de Azure Event Hubs con F#

[Azure Event Hubs](#) proporciona la ingesta de telemetría de escala de nube de sitios web, aplicaciones y dispositivos.

Azure Event Hubs se puede usar con F# de dos maneras:

1. Mediante la creación de Azure Functions de F# desencadenadas por eventos. Vea [Desencadenadores de Azure Functions para Event Hubs](#), o bien
2. Mediante el uso del [SDK de .NET para Azure](#). Tenga en cuenta que estos ejemplos son de C#.

## Uso de Azure Notification Hubs con F#

[Azure Notification Hubs](#) es una infraestructura de inserción multiplataforma y escalada que permite enviar notificaciones de inserción móviles desde cualquier back-end (en la nube o local) para cualquier plataforma móvil.

Azure Notification Hubs se puede usar con F# de dos maneras:

1. Mediante la creación de Azure Functions de F# que envían resultados a un centro de notificaciones. Vea [Desencadenadores de salida de Azure Functions para Notification Hubs](#), o bien
2. Mediante el uso del [SDK de .NET para Azure](#). Tenga en cuenta que estos ejemplos son de C#.

## Implementación de WebHooks en Azure con F#

Un [Webhook](#) es una devolución de llamada que se desencadena a través de una solicitud web. Los Webhooks se usan en sitios como GitHub para señalar eventos.

Los Webhooks pueden implementarse en F# y hospedarse en Azure a través de [Azure Functions en F# con un enlace de Webhook](#).

## Uso de Webjobs con F#

[Webjobs](#) son programas que se pueden ejecutar en la aplicación web de servicio de aplicaciones de tres maneras: bajo demanda, de forma continua o según una programación.

[Ejemplo de Webjob de F#](#)

## Implementación de temporizadores en Azure con F#

Los desencadenadores de temporizador llaman a las funciones de acuerdo a una programación, una vez o de manera periódica.

Los temporizadores pueden implementarse en F# y hospedarse en Azure a través de un [Desencadenador de temporizador de Azure Functions en F#](#).

## Implementación y administración de recursos de Azure con scripts de F#

Las máquinas virtuales de Azure se pueden implementar y administrar mediante programación desde scripts de F# con los paquetes y API de Microsoft.Azure.Management. Por ejemplo, vea [Introducción a las bibliotecas de administración para .NET](#) y [Uso de Azure Resource Manager](#).

Del mismo modo, también se pueden implementar y administrar otros recursos de Azure desde scripts de F# mediante el uso de los mismos componentes. Por ejemplo, se pueden crear cuentas de almacenamiento, implementar Azure Cloud Services, crear instancias de Azure Cosmos DB y administrar Azure Notification Hubs mediante programación desde scripts de F#.

Normalmente no es necesario usar scripts de F# para implementar y administrar recursos. Por ejemplo, los recursos de Azure también se pueden implementar directamente desde descripciones de plantillas de JSON, que pueden tener parámetros. Vea [Plantillas de Azure Resource Manager](#) con ejemplos como las [Plantillas de inicio rápido de Azure](#).

## Otros recursos

- [Documentación completa de todos los servicios de Azure](#)

# Introducción a Azure BLOB Storage mediante F#

10/02/2020 • 22 minutes to read • [Edit Online](#)

Almacenamiento de blobs de Azure es un servicio que almacena datos no estructurados en la nube como objetos o blobs. El Almacenamiento de blobs puede almacenar cualquier tipo de datos binarios o texto, como un documento, un archivo multimedia o un instalador de aplicación. El Almacenamiento de blobs a veces se conoce como "almacenamiento de objetos".

En este artículo se muestra cómo realizar tareas comunes con el almacenamiento de blobs. Los ejemplos se escriben F# mediante el uso de la biblioteca de cliente de Azure Storage para .net. Entre las tareas descritas se incluyen cómo cargar, enumerar, descargar y eliminar BLOBs.

Para obtener información general conceptual sobre el almacenamiento de blobs, consulte [la guía de .net para BLOB Storage](#).

## Prerequisites

Para usar esta guía, primero debe [crear una cuenta de almacenamiento de Azure](#). También necesitará la clave de acceso de almacenamiento para esta cuenta.

## Creación de F# un script e F# Inicio interactivo

Los ejemplos de este artículo se pueden usar en una F# aplicación o en un F# script. Para crear un F# script, cree un archivo con la extensión `.fsx`, por ejemplo `blobs.fsx`, en el F# entorno de desarrollo.

A continuación, use un [Administrador de paquetes](#) como [Paket](#) o [NuGet](#) para instalar los paquetes de `WindowsAzure.Storage` y `Microsoft.WindowsAzure.ConfigurationManager`, y haga referencia a `WindowsAzure.Storage.dll` y `Microsoft.WindowsAzure.Configuration.dll` en el script mediante una directiva de `#r`.

### Incorporación de declaraciones de espacio de nombres

Agregue las siguientes instrucciones `open` en la parte superior del archivo `blobs.fsx`:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Blob // Namespace for Blob storage types
```

### Obtención de la cadena de conexión

Necesita una cadena de conexión Azure Storage para este tutorial. Para obtener más información sobre las cadenas de conexión, consulte Configuración de las [cadenas de conexión de almacenamiento](#).

En el tutorial, escriba la cadena de conexión en el script, como se indica a continuación:

```
let storageConnString = "..." // fill this in from your storage account
```

Sin embargo, esto **no se recomienda** para los proyectos reales. La clave de la cuenta de almacenamiento es similar a la contraseña raíz de la cuenta de almacenamiento. Siempre debe proteger la clave de la cuenta de almacenamiento. Evite distribuirla a otros usuarios, codificarla de forma rígida o guardarla en un archivo de texto que sea accesible a otros usuarios. Puede volver a generar la clave mediante Azure portal si cree que puede estar en peligro.

En el caso de las aplicaciones reales, la mejor manera de mantener la cadena de conexión de almacenamiento se encuentra en un archivo de configuración. Para capturar la cadena de conexión de un archivo de configuración, puede hacer lo siguiente:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

El uso del Administrador de configuración Azure es opcional. También puede usar una API, como el tipo de `ConfigurationManager` del .NET Framework.

### Análisis de la cadena de conexión

Para analizar la cadena de conexión, use:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Esto devuelve un `CloudStorageAccount`.

### Crear algunos datos ficticios locales

Antes de empezar, cree algunos datos locales ficticios en el directorio del script. Más adelante se cargan estos datos.

```
// Create a dummy file to upload
let localFile = __SOURCE_DIRECTORY__ + "/myfile.txt"
File.WriteAllText(localFile, "some data")
```

### Creación del cliente de Blob service

El tipo de `CloudBlobClient` permite recuperar contenedores y blobs almacenados en el almacenamiento de blobs. Esta es una forma de crear el cliente de servicio:

```
let blobClient = storageAccount.CreateCloudBlobClient()
```

Ahora ya puede escribir código que lee y escribe datos en el Almacenamiento de blobs.

## Crear un contenedor

En este ejemplo se muestra cómo crear un contenedor si todavía no existe:

```
// Retrieve a reference to a container.
let container = blobClient.GetContainerReference("mydata")

// Create the container if it doesn't already exist.
container.CreateIfNotExists()
```

De manera predeterminada, el nuevo contenedor es privado, lo que significa que debe especificar su clave de acceso de almacenamiento para descargar blobs de él. Si desea poner los archivos del contenedor a disposición de todo el mundo, puede convertir el contenedor en público utilizando el código siguiente:

```
let permissions = BlobContainerPermissions(PublicAccess=BlobContainerPublicAccessType.Blob)
container.SetPermissions(permissions)
```

Cualquier usuario de Internet puede ver los blobs de los contenedores públicos, pero solo es posible modificarlos o

eliminarlos si se dispone de la clave de acceso apropiada o una firma de acceso compartido.

## Cargar un blob en un contenedor

Azure Blob Storage admite blobs en bloques y en páginas. En la mayoría de los casos, un BLOB en bloques es el tipo recomendado que se va a usar.

Para cargar un archivo en un blob en bloques, obtenga una referencia de contenedor y utilícela para obtener una referencia de blob en bloques. Una vez que tenga una referencia de BLOB, puede cargar cualquier secuencia de datos en ella llamando al método `UploadFromFile`. Esta operación crea el BLOB si no existía anteriormente, o lo sobrescribe si existe.

```
// Retrieve reference to a blob named "myblob.txt".
let blockBlob = container.GetBlockBlobReference("myblob.txt")

// Create or overwrite the "myblob.txt" blob with contents from the local file.
do blockBlob.UploadFromFile(localFile)
```

## Enumerar los blobs de un contenedor

Para enumerar los blobs de un contenedor, primero obtenga una referencia de contenedor. Después, puede usar el método `ListBlobs` del contenedor para recuperar los blobs o directorios que contiene. Para tener acceso al conjunto completo de propiedades y métodos para un `IBlobItem` devuelto, debe convertirlo en un objeto `CloudBlockBlob`, `CloudPageBlob` o `CloudBlobDirectory`. Si se desconoce el tipo, puede realizar una comprobación de tipo para determinar el formato al que se debe convertir. El código siguiente muestra cómo recuperar y consultar el URI de cada elemento del contenedor `mydata`:

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, false) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %0" blob.Properties.Length blob.Uri

    | :? CloudPageBlob as pageBlob ->
        printfn "Page blob of length %d: %0" pageBlob.Properties.Length pageBlob.Uri

    | :? CloudBlobDirectory as directory ->
        printfn "Directory: %0" directory.Uri

    | _ ->
        printfn "Unknown blob type: %0" (item.GetType())
```

También puede asignar nombres a los blobs con la información de la ruta de acceso. De este modo crea una estructura de directorios virtuales que puede organizar y recorrer tal como lo haría en un sistema de archivos tradicional. Tenga en cuenta que la estructura de directorios es solo virtual: los únicos recursos disponibles en el almacenamiento de blobs son contenedores y blobs. Sin embargo, la biblioteca de cliente de almacenamiento ofrece un objeto `CloudBlobDirectory` para hacer referencia a un directorio virtual y simplificar el proceso de trabajar con blobs organizados de este modo.

Por ejemplo, observe el siguiente conjunto de blobs en bloques incluidos en un contenedor denominado `photos`:

`\photo1.jpg`

Cuando se llama a `ListBlobs` en un contenedor (como en el ejemplo anterior), se devuelve una lista jerárquica. Si contiene objetos `CloudBlobDirectory` y `CloudBlockBlob`, que representan los directorios y los blobs del contenedor, respectivamente, la salida resultante tiene un aspecto similar al siguiente:

```
Directory: https://<accountname>.blob.core.windows.net/photos/2015/
Directory: https://<accountname>.blob.core.windows.net/photos/2016/
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

Opcionalmente, puede establecer el parámetro `UseFlatBlobListing` del método `ListBlobs` en `true`. En este caso, cada BLOB del contenedor se devuelve como un objeto `CloudBlockBlob`. La llamada a `ListBlobs` para devolver una lista plana tiene el siguiente aspecto:

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, true) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %0" blob.Properties.Length blob.Uri

    | _ ->
        printfn "Unexpected blob type: %0" (item.GetType())
```

y, en función del contenido actual del contenedor, los resultados son similares a los siguientes:

```
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2015/architecture/description.txt
Block blob of length 314618: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo3.jpg
Block blob of length 522713: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo4.jpg
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2016/architecture/description.txt
Block blob of length 419048: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo5.jpg
Block blob of length 506388: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo6.jpg
Block blob of length 399751: https://<accountname>.blob.core.windows.net/photos/2016/photo7.jpg
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

## Descargar blobs

Para descargar blobs, primero recupere una referencia de BLOB y, a continuación, llame al método `DownloadToStream`. En el ejemplo siguiente se usa el método `DownloadToStream` para transferir el contenido del BLOB a un objeto de secuencia que se puede guardar en un archivo local.

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDownload = container.GetBlockBlobReference("myblob.txt")

// Save blob contents to a file.
do
    use fileStream = File.OpenWrite(__SOURCE_DIRECTORY__ + "/path/download.txt")
    blobToDownload.DownloadToStream(fileStream)
```

También puede utilizar el método `DownloadToStream` para descargar el contenido de un BLOB como una cadena de texto.

```
let text =
    use memoryStream = new MemoryStream()
    blobToDownload.DownloadToStream(memoryStream)
    Text.Encoding.UTF8.GetString(memoryStream.ToArray())
```

## Eliminar blobs

Para eliminar un BLOB, primero obtenga una referencia de BLOB y, a continuación, llame al método `Delete` en ella.



```
// Retrieve reference to a blob named "myblob.txt".
let blobToDelete = container.GetBlockBlobReference("myblob.txt")

// Delete the blob.
blobToDelete.Delete()
```

## Enumerar blobs en páginas de forma asíncrona

Si enumera un gran número de blobs o desea controlar el número de resultados que devuelve en una operación de listado, puede enumerar blobs en páginas de resultados. En este ejemplo se muestra cómo devolver resultados en páginas asíncronicamente de forma que la ejecución no se bloquee mientras se espera a devolver un conjunto grande de resultados.

En este ejemplo se muestra una lista plana de blobs, pero también puede realizar una lista jerárquica estableciendo el parámetro `useFlatBlobListing` del método `ListBlobsSegmentedAsync` en `false`.

En el ejemplo se define un método asíncrono mediante un bloque `async`. La palabra clave `let!` suspende la ejecución del método de ejemplo hasta que se completa la tarea de enumeración.

```
let ListBlobsSegmentedInFlatListing(container:CloudBlobContainer) =
    async {

        // List blobs to the console window, with paging.
        printfn "List blobs in pages:"

        // Call ListBlobsSegmentedAsync and enumerate the result segment
        // returned, while the continuation token is non-null.
        // When the continuation token is null, the last page has been
        // returned and execution can exit the loop.

        let rec loop continuationToken (i:int) =
            async {
                let! ct = Async.CancellationToken
                // This overload allows control of the page size. You can return
                // all remaining results by passing null for the maxResults
                // parameter, or by calling a different overload.
                let! resultSegment =
                    container.ListBlobsSegmentedAsync(
                        "", true, BlobListingDetails.All, Nullable 10,
                        continuationToken, null, null, ct)
                |> Async.AwaitTask

                if (resultSegment.Results |> Seq.length > 0) then
                    printfn "Page %d:" i

                for blobItem in resultSegment.Results do
                    printfn "\t%0" blobItem.StorageUri.PrimaryUri

                printfn ""

                // Get the continuation token.
                let continuationToken = resultSegment.ContinuationToken
                if (continuationToken <> null) then
                    do! loop continuationToken (i+1)
            }

        do! loop null 1
    }
```

Ahora podemos usar esta rutina asíncrona como se indica a continuación. En primer lugar, cargue algunos datos ficticios (mediante el archivo local creado anteriormente en este tutorial).

```
// Create some dummy data by uploading the same file over and over again
for i in 1 .. 100 do
    let blob = container.GetBlockBlobReference("myblob" + string i + ".txt")
    use fileStream = System.IO.File.OpenRead(localFile)
    blob.UploadFromFile(localFile)
```

Ahora, llame a la rutina. Utilice `Async.RunSynchronously` para forzar la ejecución de la operación asíncrona.

```
ListBlobsSegmentedInFlatListing container |> Async.RunSynchronously
```

## Escritura en un blob en anexos

Un blob en anexos se optimiza para las operaciones de anexado, como el registro. Como un blob en bloques, un blob en anexos se compone también de bloques, pero en el caso del blob en anexos cuando se agrega un nuevo bloque, siempre se anexa al final del blob. No se puede actualizar o eliminar un bloque existente en un blob en anexos. Los identificadores de bloque para un blob en anexos no está expuestos como lo están en el caso de los blobs en bloques.

Cada bloque en un blob en anexos puede tener un tamaño diferente, hasta un máximo de 4 MB y el blob puede incluir un máximo de 50.000 bloques. El tamaño máximo de un blob en anexos es, por tanto, ligeramente superior a 195 GB (4 MB X 50.000 bloques).

En el ejemplo siguiente se crea un nuevo BLOB en anexos y se anexan algunos datos, simulando una operación de registro simple.

```
// Get a reference to a container.
let appendContainer = blobClient.GetContainerReference("my-append-blobs")

// Create the container if it does not already exist.
appendContainer.CreateIfNotExists() |> ignore

// Get a reference to an append blob.
let appendBlob = appendContainer.GetAppendBlobReference("append-blob.log")

// Create the append blob. Note that if the blob already exists, the
// CreateOrReplace() method will overwrite it. You can check whether the
// blob exists to avoid overwriting it by using CloudAppendBlob.Exists().
appendBlob.CreateOrReplace()

let numBlocks = 10

// Generate an array of random bytes.
let rnd = new Random()
let bytes = Array.zeroCreate<byte>(numBlocks)
rnd.NextBytes(bytes)

// Simulate a logging operation by writing text data and byte data to the
// end of the append blob.
for i in 0 .. numBlocks - 1 do
    let msg = sprintf "Timestamp: %u \tLog Entry: %d\n" DateTime.UtcNow bytes.[i]
    appendBlob.AppendText(msg)

// Read the append blob to the console window.
let downloadedText = appendBlob.DownloadText()
printfn "%s" downloadedText
```

Consulte [Descripción Blobs en bloques, en anexos y en páginas](#) para obtener más información acerca de las diferencias entre los tres tipos de blobs.

# simultáneo

Para permitir el acceso simultáneo a un blob desde varios clientes o varias instancias de proceso, puede usar etiquetas **ETag** o **concesiones**.

- **Etag** : proporciona una manera de detectar que otro proceso ha modificado el blob o el contenedor
- **Concesión** : proporciona una manera de obtener acceso exclusivo y renovable de escritura o eliminación a un blob durante un período de tiempo

Para obtener más información, vea [administrar la simultaneidad en Microsoft Azure Storage](#).

## Nomenclatura de contenedores

Cada blob del almacenamiento de Azure debe residir en un contenedor. El contenedor forma parte del nombre del blob. Por ejemplo, `mydata` es el nombre del contenedor de estos URI de blob de ejemplo:

- `https://storagesample.blob.core.windows.net/mydata/blob1.txt`
- `https://storagesample.blob.core.windows.net/mydata/photos/myphoto.jpg`

Un nombre de contenedor debe ser un nombre DNS válido y cumplir las reglas de nomenclatura siguientes:

1. Los nombres de contenedor deben comenzar por una letra o un número, y pueden contener solo letras, números y el carácter de guión (-).
2. Todos los caracteres de guión (-) deben estar inmediatamente precedidos y seguidos por una letra o un número; no se permiten guiones consecutivos en nombres de contenedor.
3. Todas las letras del nombre de un contenedor deben aparecer en minúsculas.
4. Los nombres de contenedor deben tener entre 3 y 63 caracteres de longitud.

Tenga en cuenta que el nombre de un contenedor siempre debe estar en minúsculas. Si se incluye una letra mayúscula en un nombre de contenedor o se infringe de algún otro modo las reglas de nomenclatura de contenedores, recibirá un error 400 (solicitud incorrecta).

## Administración de la seguridad para blobs

De forma predeterminada, Azure Storage protege sus datos al limitar el acceso al propietario de la cuenta, quien está en posesión de las claves de acceso a ella. Cuando necesite compartir datos Blob en su cuenta de almacenamiento, es importante hacerlo sin poner en peligro la seguridad de las claves de acceso de la cuenta. Además, puede cifrar los datos Blob para cerciorarse de que es seguro pasar por la red y Azure Storage.

### Control del acceso a datos Blob

De forma predeterminada, los datos Blob de su cuenta de almacenamiento solo son accesibles para el propietario de la cuenta de almacenamiento. Para autenticar las solicitudes en el Almacenamiento de blobs, se necesita la clave de acceso de la cuenta de forma predeterminada. Sin embargo, es posible que desee poner determinados datos de BLOB a disposición de otros usuarios.

### Cifrado de datos Blob

Azure Storage admite el cifrado de datos de BLOB tanto en el cliente como en el servidor.

## Pasos siguientes

Ahora que está familiarizado con los aspectos básicos del Almacenamiento de blobs, siga estos vínculos para obtener más información.

### Herramientas

- [F#\ AzureStorageTypeProvider](#)

- [FSharp. Azure. Storage](#)

Una F# API para usar Microsoft Azure servicio Table Storage

- [Explorador de Microsoft Azure Storage \(Mase\)](#)

Una aplicación independiente y gratuita de Microsoft que le permite trabajar visualmente con datos de Azure Storage en Windows, OS X y Linux.

### **Referencia de Almacenamiento de blobs**

- [API de Azure Storage para .NET](#)
- [Referencia de la API REST de los servicios de Azure Storage](#)

### **Guías relacionadas**

- [Azure Blob Storage Samples for .NET](#) (Ejemplos de Azure Blob Storage para .NET)
- [Introducción a AzCopy](#)
- [Configuración de las cadenas de conexión de Azure Storage](#)
- [Blog del equipo de Azure Storage](#)
- [Inicio rápido: uso de .NET para crear un BLOB en el almacenamiento de objetos](#)

# Introducción a Azure File Storage con F#

21/04/2020 • 12 minutes to read • [Edit Online](#)

Almacenamiento de archivos de Azure es un servicio que ofrece recursos compartidos de archivos en la nube mediante el [protocolo Bloque de mensajes del servidor \(SMB\)](#) estándar. Se admiten SMB 2.1 y SMB 3.0. Con Almacenamiento de archivos de Azure puede migrar aplicaciones heredadas basadas en recursos compartidos de archivos a Azure con rapidez y sin necesidad de costosas reescrituras. Las aplicaciones que se ejecutan en máquinas virtuales de Azure o en servicios en la nube o desde clientes locales pueden montar un recurso compartido de archivos en la nube, igual que una aplicación de escritorio monta un recurso compartido SMB típico. Cualquier número de componentes de aplicación puede montar y acceder simultáneamente al recurso compartido de Almacenamiento de archivos.

Para obtener información general conceptual sobre el almacenamiento de archivos, consulte [la guía de .NET para el almacenamiento de archivos](#).

## Prerrequisitos

Para usar esta guía, primero debe crear una cuenta de [Azure Storage](#). También necesitará la clave de acceso de almacenamiento para esta cuenta.

## Crear una secuencia de comandos de F e iniciar F. Interactive

Los ejemplos de este artículo se pueden usar en una aplicación de F o en una secuencia de comandos de F. Para crear una secuencia de comandos `.fsx` de F, cree un archivo con la extensión, por ejemplo, `files.fsx` en el entorno de desarrollo de F.

A continuación, use un administrador de [paquetes](#) `WindowsAzure.Storage` como [Paket](#) o [NuGet](#) para instalar el paquete y la referencia `WindowsAzure.Storage.dll` en el script mediante una `#r` directiva.

### Incorporación de declaraciones de espacio de nombres

Agregue las siguientes instrucciones `open` en la parte superior del archivo `files.fsx`:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.File // Namespace for File storage types
```

### Obtención de la cadena de conexión

Necesitará una cadena de conexión de Azure Storage para este tutorial. Para obtener más información acerca de las cadenas de conexión, vea Configurar cadenas de conexión de [almacenamiento](#).

Para el tutorial, escribirá la cadena de conexión en el script, de la siguiente manera:

```
let storageConnString = "..." // fill this in from your storage account
```

Sin embargo, esto no se **recomienda** para proyectos reales. La clave de la cuenta de almacenamiento es similar a la contraseña raíz de la cuenta de almacenamiento. Siempre debe proteger la clave de la cuenta de almacenamiento. Evite distribuirla a otros usuarios, codificarla de forma rígida o guardarla en un archivo de texto que sea accesible a otros usuarios. Puede volver a generar la clave mediante Azure Portal si cree que se ha visto

comprometida.

Para aplicaciones reales, la mejor manera de mantener la cadena de conexión de almacenamiento es en un archivo de configuración. Para capturar la cadena de conexión de un archivo de configuración, puede hacer lo siguiente:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

El uso del Administrador de configuración Azure es opcional. También puede usar una API como el `ConfigurationManager` tipo de .NET Framework.

### Análisis de la cadena de conexión

Para analizar la cadena de conexión, utilice:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Esto devolverá `CloudStorageAccount` un archivo .

### Crear el cliente de servicio de archivos

El `CloudFileClient` tipo le permite usar mediante programación archivos almacenados en almacenamiento de archivos. Esta es una forma de crear el cliente de servicio:

```
let fileClient = storageAccount.CreateCloudFileClient()
```

Ahora está listo para escribir código que lea y escriba datos en almacenamiento de archivos.

## Creación de un recurso compartido de archivos

Este ejemplo muestra cómo crear un recurso compartido de archivos si aún no existe:

```
let share = fileClient.GetShareReference("myfiles")
share.CreateIfNotExists()
```

## Crear un directorio raíz y un subdirectorio

Aquí, obtendrá el directorio raíz y obtenga un subdirectorio de la raíz. Creas ambos si aún no existen.

```
let rootDir = share.GetRootDirectoryReference()
let subDir = rootDir.GetDirectoryReference("myLogs")
subDir.CreateIfNotExists()
```

## Cargar texto como archivo

En este ejemplo se muestra cómo cargar texto como un archivo.

```
let file = subDir.GetFileReference("log.txt")
file.UploadText("This is the content of the log file")
```

### Descargar un archivo en una copia local del archivo

Aquí se descarga el archivo recién creado, añadiendo el contenido a un archivo local.

```
file.DownloadToFile("log.txt", FileMode.Append)
```

### Establecer el tamaño máximo para un recurso compartido de archivos

En el ejemplo siguiente se muestra cómo comprobar el uso actual de un recurso compartido y cómo establecer la cuota para el recurso compartido. `FetchAttributes` debe llamarse para rellenar `Properties` el `SetProperties` recurso compartido y para propagar los cambios locales al almacenamiento de archivos de Azure.

```
// stats.Usage is current usage in GB
let stats = share.GetStats()
share.FetchAttributes()

// Set the quota to 10 GB plus current usage
share.Properties.Quota <- stats.Usage + 10 |> Nullable
share.SetProperties()

// Remove the quota
share.Properties.Quota <- Nullable()
share.SetProperties()
```

### Generar una firma de acceso compartido para un archivo o recurso compartido de archivos

Puede generar una firma de acceso compartido (SAS) para un recurso compartido de archivos o para un archivo individual. También puede crear una directiva de acceso compartido en un recurso compartido de archivos para administrar firmas de acceso compartido. Se recomienda la creación de una directiva de acceso compartido, ya que ofrece un medio de revocar la SAS si esta se encuentra en peligro.

Aquí, se crea una directiva de acceso compartido en un recurso compartido y, a continuación, se usa esa directiva para proporcionar las restricciones para una SAS en un archivo del recurso compartido.

```
// Create a 24-hour read/write policy.
let policy =
    SharedAccessFilePolicy
        (SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable),
         Permissions = (SharedAccessFilePermissions.Read ||| SharedAccessFilePermissions.Write))

// Set the policy on the share.
let permissions = share.GetPermissions()
permissions.SharedAccessPolicies.Add("policyName", policy)
share.SetPermissions(permissions)

let sasToken = file.GetSharedAccessSignature(policy)
let sasUri = Uri(file.StorageUri.PrimaryUri.ToString() + sasToken)

let fileSas = CloudFile(sasUri)
fileSas.UploadText("This write operation is authenticated via SAS")
```

Para más información sobre la creación y el uso de firmas de acceso compartido, consulte [Uso de firmas de acceso compartido \(SAS\)](#) y [Creación y uso de una SAS con Blob Storage](#).

### Copiar archivos

Puede copiar un archivo en otro archivo o en un blob o en un blob en un archivo. Si va a copiar un blob en un archivo o un archivo en un blob, *debe* usar una firma de acceso compartido (SAS) para autenticar el objeto de origen, incluso si va a copiar dentro de la misma cuenta de almacenamiento.

### Copiar un archivo en otro

Aquí, copia un archivo en otro archivo del mismo recurso compartido. Dado que en esta operación de copia se copia entre archivos de la misma cuenta de almacenamiento, puede usar la autenticación de clave compartida para realizar la copia.

```
let destFile = subDir.GetFileReference("log_copy.txt")
destFile.StartCopy(file)
```

### Copiar un archivo en un blob

Aquí, se crea un archivo y se copia en un blob dentro de la misma cuenta de almacenamiento. Crear una SAS para el archivo de origen, que el servicio utiliza para autenticar el acceso al archivo de origen durante la operación de copia.

```
// Get a reference to the blob to which the file will be copied.
let blobClient = storageAccount.CreateCloudBlobClient()
let container = blobClient.GetContainerReference("myContainer")
container.CreateIfNotExists()
let destBlob = container.GetBlockBlobReference("log_blob.txt")

let filePolicy =
    SharedAccessFilePolicy
        (Permissions = SharedAccessFilePermissions.Read,
         SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable))

let fileSas2 = file.GetSharedAccessSignature(filePolicy)
let sasUri2 = Uri(file.StorageUri.PrimaryUri.ToString() + fileSas2)
destBlob.StartCopy(sasUri2)
```

Puede copiar un blob en un archivo de la misma manera. Si el objeto de origen es un blob, cree una SAS para autenticar el acceso a dicho blob durante la operación de copia.

## Solución de problemas de almacenamiento de archivos mediante métricas

Azure Storage Analytics admite métricas para el almacenamiento de archivos. Con los datos de las métricas, es posible seguir paso a paso las solicitudes y diagnosticar problemas.

Puede habilitar las métricas para el almacenamiento de archivos desde [Azure Portal](#) o puede hacerlo desde F#, de la siguiente manera:

```
open Microsoft.Azure.Storage.File.Protocol
open Microsoft.Azure.Storage.Shared.Protocol

let props =
    FileServiceProperties(
        (HourMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (14 |> Nullable),
            Version = "1.0"),
         MinuteMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (7 |> Nullable),
            Version = "1.0"))

fileClient.SetServiceProperties(props)
```

## Pasos siguientes



Para obtener más información acerca de Azure File Storage, consulte estos vínculos.

### **Artículos y vídeos conceptuales**

- [Almacenamiento de archivos de Azure: un sistema de archivos SMB en la nube sin dificultades para Windows y Linux](#)
- [Uso de Azure File Storage con Linux](#)

### **Compatibilidad de herramientas con el almacenamiento de archivos**

- [Usar Azure PowerShell con Azure Storage](#)
- [Uso de AzCopy con Microsoft Azure Storage](#)
- [Creación, descarga y enumeración de blobs mediante la CLI de Azure](#)

### **Referencia**

- [Referencia de la biblioteca de clientes de almacenamiento para .NET](#)
- [Referencia de la API REST del servicio de archivos](#)

### **Publicaciones de blog**

- [El almacenamiento de archivos de Azure ya está disponible de manera general](#)
- [Dentro de Azure File Storage](#)
- [Introducing Microsoft Azure File Service \(Introducción al servicio de archivos de Microsoft Azure\)](#)
- [Persisting connections to Microsoft Azure Files \(Persistencia de conexiones en archivos de Microsoft Azure\)](#)

# Introducción a Azure Queue Storage mediante F#

15/01/2020 • 12 minutes to read • [Edit Online](#)

El almacenamiento en cola de Azure proporciona mensajería en la nube entre componentes de aplicaciones. A la hora de diseñar aplicaciones para escala, los componentes de las mismas suelen desacoplarse para poder escalarlos de forma independiente. El almacenamiento en cola ofrece mensajería asíncrona para la comunicación entre los componentes de las aplicaciones, independientemente de si se ejecutan en la nube, en el escritorio, en un servidor local o en un dispositivo móvil. Además, este tipo de almacenamiento admite la administración de tareas asíncronas y la creación de flujos de trabajo de procesos.

## Acerca de este tutorial

En este tutorial se muestra cómo F# escribir código para algunas tareas comunes mediante el almacenamiento de colas de Azure. Entre las tareas descritas se incluyen la creación y eliminación de colas y la adición, lectura y eliminación de mensajes de la cola.

Para obtener información general conceptual sobre Queue Storage, consulte [la guía de .net para Queue Storage](#).

## Requisitos previos

Para usar esta guía, primero debe [crear una cuenta de almacenamiento de Azure](#). También necesitará la clave de acceso de almacenamiento para esta cuenta.

## Creación de F# un script e F# Inicio interactivo

Los ejemplos de este artículo se pueden usar en una F# aplicación o en un F# script. Para crear un F# script, cree un archivo con la extensión `.fsx`, por ejemplo `queues.fsx`, en el F# entorno de desarrollo.

A continuación, use un [Administrador de paquetes](#) como [Paket](#) o [NuGet](#) para instalar el paquete

`WindowsAzure.Storage` y haga referencia `WindowsAzure.Storage.dll` en el script mediante una directiva `#r`.

## Agregar declaraciones de espacios de nombres

Agregue las siguientes instrucciones `open` en la parte superior del archivo `queues.fsx`:

```
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Queue // Namespace for Queue storage types
```

## Obtención de la cadena de conexión

Necesitará una cadena de conexión Azure Storage para este tutorial. Para obtener más información sobre las cadenas de conexión, consulte Configuración de las [cadenas de conexión de almacenamiento](#).

En el tutorial, escribirá la cadena de conexión en el script, de la siguiente manera:

```
let storageConnString = "... " // fill this in from your storage account
```

Sin embargo, esto **no se recomienda** para los proyectos reales. La clave de la cuenta de almacenamiento es similar a la contraseña raíz de la cuenta de almacenamiento. Siempre debe proteger la clave de la cuenta de almacenamiento. Evite distribuirla a otros usuarios, codificarla de forma rígida o guardarla en un archivo de texto que sea accesible a otros usuarios. Puede volver a generar la clave mediante Azure portal si cree que puede estar en peligro.

En el caso de las aplicaciones reales, la mejor manera de mantener la cadena de conexión de almacenamiento se encuentra en un archivo de configuración. Para capturar la cadena de conexión de un archivo de configuración, puede hacer lo siguiente:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

El uso del Administrador de configuración Azure es opcional. También puede usar una API, como el tipo de `ConfigurationManager` del .NET Framework.

### Análisis de la cadena de conexión

Para analizar la cadena de conexión, use:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Esto devolverá un `CloudStorageAccount`.

### Creación del cliente del servicio Cola

La clase `CloudQueueClient` permite recuperar las colas almacenadas en Queue Storage. Esta es una forma de crear el cliente de servicio:

```
let queueClient = storageAccount.CreateCloudQueueClient()
```

Ahora ya puede escribir código que lee y escribe datos en el Almacenamiento en cola.

## Crear una cola

En este ejemplo se muestra cómo crear una cola si aún no existe:

```
// Retrieve a reference to a container.
let queue = queueClient.GetQueueReference("myqueue")

// Create the queue if it doesn't already exist
queue.CreateIfNotExists()
```

## un mensaje en una cola

Para insertar un mensaje en una cola existente, cree primero un nuevo `CloudQueueMessage`. A continuación, llame al método `AddMessage`. Un `CloudQueueMessage` se puede crear a partir de una cadena (en formato UTF-8) o de una matriz de `byte`, de la siguiente manera:

```
// Create a message and add it to the queue.
let message = new CloudQueueMessage("Hello, World")
queue.AddMessage(message)
```

## siguiente mensaje

Puede inspeccionar el mensaje situado en la parte delantera de una cola, sin quitarlo de la cola, llamando al método `PeekMessage`.

```
// Peek at the next message.
let peekedMessage = queue.PeekMessage()
let msgAsString = peekedMessage.AsString
```

## Obtener el siguiente mensaje para el procesamiento

Puede recuperar el mensaje al principio de una cola para su procesamiento llamando al método `GetMessage`.

```
// Get the next message. Successful processing must be indicated via DeleteMessage later.
let retrieved = queue.GetMessage()
```

Más adelante indicará el procesamiento correcto del mensaje mediante `DeleteMessage`.

## contenido de un mensaje en cola

Puede cambiar el contenido de un mensaje recuperado en la cola. Si el mensaje representa una tarea de trabajo, puede usar esta característica para actualizar el estado de la tarea de trabajo. El siguiente código actualiza el mensaje de la cola con contenido nuevo y amplía el tiempo de espera de la visibilidad en 60 segundos más. De este modo, se guarda el estado de trabajo asociado al mensaje y se le proporciona al cliente un minuto más para que siga elaborando el mensaje. Esta técnica se puede utilizar para realizar un seguimiento de los flujos de trabajo de varios pasos en los mensajes en cola, sin que sea necesario volver a empezar desde el principio si se produce un error en un paso del proceso a causa de un error de hardware o software. Normalmente, también tendría que mantener un número de reintentos y, si el mensaje se vuelve a intentar más de un número de veces, lo eliminaría. Esto proporciona protección frente a un mensaje que produce un error en la aplicación cada vez que se procesa.

```
// Update the message contents and set a new timeout.
retrieved.SetMessageContent("Updated contents.")
queue.UpdateMessage(retrieved,
    TimeSpan.FromSeconds(60.0),
    MessageUpdateFields.Content ||| MessageUpdateFields.Visibility)
```

## siguiente mensaje de la cola

El código quita un mensaje de una cola en dos pasos. Cuando se llama a `GetMessage`, se obtiene el siguiente mensaje en una cola. Un mensaje devuelto por `GetMessage` se hace invisible a cualquier otro código de lectura de mensajes de esta cola. De forma predeterminada, este mensaje permanece invisible durante 30 segundos. Para terminar de quitar el mensaje de la cola, también debe llamar a `DeleteMessage`. Este proceso de extracción de un mensaje que consta de dos pasos garantiza que si su código no puede procesar un mensaje a causa de un error de hardware o software, otra instancia de su código puede obtener el mismo mensaje e intentarlo de nuevo. El código siguiente llama a `DeleteMessage` justo después de haberse procesado el mensaje.

```
// Process the message in less than 30 seconds, and then delete the message.
queue.DeleteMessage(retrieved)
```

## Uso de flujos de trabajo asincrónicos con API comunes de almacenamiento de colas

En este ejemplo se muestra cómo usar un flujo de trabajo asincrónico con API comunes de almacenamiento de colas.

```

async {
    let! exists = queue.CreateIfNotExistsAsync() |> Async.AwaitTask

    let! retrieved = queue.GetMessageAsync() |> Async.AwaitTask

    // ... process the message here ...

    // Now indicate successful processing:
    do! queue.DeleteMessageAsync(retrieved) |> Async.AwaitTask
}

```

## Opciones adicionales para quitar mensajes de la cola

Hay dos formas de personalizar la recuperación de mensajes de una cola. En primer lugar, puede obtener un lote de mensajes (hasta 32). En segundo lugar, puede establecer un tiempo de espera de la invisibilidad más largo o más corto para que el código disponga de más o menos tiempo para procesar cada mensaje. En el ejemplo de código siguiente se usa `GetMessages` para obtener 20 mensajes en una llamada y, a continuación, procesa cada mensaje. También establece el tiempo de espera de la invisibilidad en cinco minutos para cada mensaje. Tenga en cuenta que los 5 minutos se inician para todos los mensajes al mismo tiempo, por lo que después de pasar 5 minutos desde la llamada a `GetMessages`, los mensajes que no se han eliminado volverán a estar visibles.

```

for msg in queue.GetMessages(20, Nullable(TimeSpan.FromMinutes(5))) do
    // Process the message here.
    queue.DeleteMessage(msg)

```

## la longitud de la cola

Puede obtener una estimación del número de mensajes existentes en una cola. El método `FetchAttributes` pide al Queue service que recupere los atributos de la cola, incluido el recuento de mensajes. La propiedad `ApproximateMessageCount` devuelve el último valor recuperado por el método `FetchAttributes`, sin llamar a la Queue service.

```

queue.FetchAttributes()
let count = queue.ApproximateMessageCount.GetValueOrDefault()

```

## Eliminar una cola

Para eliminar una cola y todos los mensajes contenidos en ella, llame al método `Delete` en el objeto Queue.

```

// Delete the queue.
queue.Delete()

```

## Pasos siguientes

Ahora que está familiarizado con los aspectos básicos del almacenamiento de colas, utilice estos vínculos para obtener más información acerca de tareas de almacenamiento más complejas.

- [API de Azure Storage para .NET](#)
- [Azure Storage proveedor de tipos](#)
- [Blog del equipo de Azure Storage](#)
- [Configuración de las cadenas de conexión de Azure Storage](#)

- [Azure Storage Services REST API Reference](#) (Referencia de la API REST de los servicios de Azure Storage)

# Introducción a Azure Table Storage y el Table API de Azure Cosmos DB con F#

15/01/2020 • 17 minutes to read • [Edit Online](#)

Almacenamiento de tablas de Azure es un servicio que almacena datos de NoSQL estructurados en la nube. Almacenamiento de tablas es un almacén de claves/atributos con un diseño sin esquema. Como Almacenamiento de tablas carece de esquema, es fácil adaptar los datos a medida que evolucionan las necesidades de la aplicación. El acceso a los datos es rápido y rentable para todos los tipos de aplicaciones y, además, su coste es muy inferior al del SQL tradicional para volúmenes de datos similares.

Table Storage se puede usar para almacenar conjuntos de datos flexibles, como datos de usuarios para aplicaciones web, libretas de direcciones, información de dispositivos y cualquier otro tipo de metadatos requerido por el servicio. Una tabla puede almacenar un número cualquiera de entidades y una cuenta de almacenamiento puede incluir un número cualquiera de tablas, hasta alcanzar el límite de capacidad de este tipo de cuenta.

Azure Cosmos DB proporciona el Table API para las aplicaciones escritas para Azure Table Storage y que necesitan funcionalidades Premium como:

- Distribución global llave en mano.
- Rendimiento dedicado en todo el mundo.
- Latencias en milisegundos de un solo dígito en el percentil 99.
- Alta disponibilidad garantizada.
- Indexación secundaria automática.

Las aplicaciones escritas para Azure Table Storage pueden migrarse a Azure Cosmos DB mediante la API Table sin realizar ningún cambio en el código y pueden sacar provecho de las funcionalidades premium. Table API tiene SDK de cliente disponibles para .NET, Java, Python y Node.js.

Para obtener más información, vea [Introducción a la Table API de Azure Cosmos dB](#).

## Acerca de este tutorial

En este tutorial se muestra cómo F# escribir código para realizar algunas tareas comunes con el almacenamiento de tablas de Azure o el Table API Azure Cosmos dB, lo que incluye crear y eliminar una tabla e insertar, actualizar, eliminar y consultar datos de tabla.

## Requisitos previos

Para usar esta guía, primero debe [crear una cuenta de almacenamiento de Azure o una cuenta de Azure Cosmos dB](#).

## Creación de F# un script e F# Inicio interactivo

Los ejemplos de este artículo se pueden usar en una F# aplicación o en un F# script. Para crear un F# script, cree un archivo con la extensión `.fsx`, por ejemplo `tables.fsx`, en el F# entorno de desarrollo.

A continuación, use un [Administrador de paquetes](#) como [Paket](#) o [NuGet](#) para instalar el paquete `WindowsAzure.Storage` y haga referencia `WindowsAzure.Storage.dll` en el script mediante una directiva `#r`. Vuelva a hacerlo por `Microsoft.WindowsAzure.ConfigurationManager` para obtener el espacio de nombres Microsoft. Azure.

### Agregar declaraciones de espacios de nombres

Agregue las siguientes instrucciones `open` en la parte superior del archivo `tables.fsx`:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Table // Namespace for Table storage types
```

### Obtener la cadena de conexión de Azure Storage

Si se va a conectar a Azure Storage Table service, necesitará la cadena de conexión para este tutorial. Puede copiar la cadena de conexión desde el Azure Portal. Para obtener más información sobre las cadenas de conexión, consulte Configuración de las [cadenas de conexión de almacenamiento](#).

### Obtener la cadena de conexión de Azure Cosmos DB

Si se va a conectar a Azure Cosmos DB, necesitará la cadena de conexión para este tutorial. Puede copiar la cadena de conexión desde el Azure Portal. En el Azure Portal, en la cuenta de Cosmos DB, vaya a **configuración** > **cadena de conexión** y haga clic en el botón **copiar** para copiar la cadena de conexión principal.

En el tutorial, escriba la cadena de conexión en el script, como en el ejemplo siguiente:

```
let storageConnString = "..." // fill this in from your storage account
```

Sin embargo, esto **no se recomienda** para los proyectos reales. La clave de la cuenta de almacenamiento es similar a la contraseña raíz de la cuenta de almacenamiento. Siempre debe proteger la clave de la cuenta de almacenamiento. Evite distribuirla a otros usuarios, codificarla de forma rígida o guardarla en un archivo de texto que sea accesible a otros usuarios. Puede volver a generar la clave mediante Azure portal si cree que puede estar en peligro.

En el caso de las aplicaciones reales, la mejor manera de mantener la cadena de conexión de almacenamiento se encuentra en un archivo de configuración. Para capturar la cadena de conexión de un archivo de configuración, puede hacer lo siguiente:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

El uso del Administrador de configuración Azure es opcional. También puede usar una API, como el tipo de `ConfigurationManager` del .NET Framework.

### Análisis de la cadena de conexión

Para analizar la cadena de conexión, use:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Esto devuelve un `CloudStorageAccount`.

### Creación del cliente de Table service

La clase `CloudTableClient` permite recuperar tablas y entidades en el almacenamiento de tablas. Esta es una forma de crear el cliente de servicio:



```
// Create the table client.  
let tableClient = storageAccount.CreateCloudTableClient()
```

Ahora ya puede escribir código que lee y escribe datos en Almacenamiento de tablas.

### Creación de una tabla

En este ejemplo se muestra cómo crear una tabla si todavía no existe:

```
// Retrieve a reference to the table.  
let table = tableClient.GetTableReference("people")  
  
// Create the table if it doesn't exist.  
table.CreateIfNotExists()
```

### Adición de una entidad a una tabla

Una entidad debe tener un tipo que herede de `TableEntity`. Puede extender `TableEntity` de la forma que desee, pero el tipo *debe* tener un constructor sin parámetros. Solo las propiedades que tienen `get` y `set` se almacenan en la tabla de Azure.

La clave de partición y de fila de una entidad identifica de forma única la entidad en la tabla. Las entidades con la misma clave de partición pueden consultarse más rápidamente que aquellas con diferentes claves de partición, pero el uso de claves de partición diversas permite una mayor escalabilidad de las operaciones paralelas.

A continuación se muestra un ejemplo de un `Customer` que usa el `lastName` como clave de partición y la `firstName` como clave de fila.

```
type Customer(firstName, lastName, email: string, phone: string) =  
    inherit TableEntity(partitionKey=lastName, rowKey=firstName)  
    new() = Customer(null, null, null, null)  
    member val Email = email with get, set  
    member val PhoneNumber = phone with get, set  
  
let customer =  
    Customer("Walter", "Harp", "Walter@contoso.com", "425-555-0101")
```

Ahora, agregue `Customer` a la tabla. Para ello, cree una `TableOperation` que se ejecute en la tabla. En este caso, se crea una operación de `Insert`.

```
let insertOp = TableOperation.Insert(customer)  
table.Execute(insertOp)
```

### Inserción de un lote de entidades

Puede insertar un lote de entidades en una tabla mediante una sola operación de escritura. Las operaciones por lotes permiten combinar operaciones en una sola ejecución, pero tienen algunas restricciones:

- Puede realizar actualizaciones, eliminaciones e inserciones en la misma operación por lotes.
- Una operación por lotes puede incluir hasta 100 entidades.
- Todas las entidades de una operación por lotes deben tener la misma clave de partición.
- Aunque es posible realizar una consulta en una operación por lotes, debe ser la única operación del lote.

Este es un código que combina dos inserciones en una operación por lotes:

```
let customer1 =
    Customer("Jeff", "Smith", "Jeff@contoso.com", "425-555-0102")

let customer2 =
    Customer("Ben", "Smith", "Ben@contoso.com", "425-555-0103")

let batchOp = TableBatchOperation()
batchOp.Insert(customer1)
batchOp.Insert(customer2)
table.ExecuteBatch(batchOp)
```

### todas las entidades de una partición

Para consultar una tabla a fin de obtener todas las entidades de una partición, use un objeto `TableQuery`. Aquí, se filtran las entidades en las que "Smith" es la clave de partición.

```
let query =
    TableQuery<Customer>().Where(
        TableQuery.GenerateFilterCondition(
            "PartitionKey", QueryComparisons.Equal, "Smith"))

let result = table.ExecuteQuery(query)
```

Ahora imprime los resultados:

```
for customer in result do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

### Recuperación de un rango de entidades de una partición

Si no desea consultar todas las entidades de una partición, puede especificar un rango combinando el filtro de clave de partición con un filtro de clave de fila. En este caso, se usan dos filtros para obtener todas las entidades de la partición "Smith" en las que la clave de fila (nombre de pila) empieza por una letra anterior a "M" en el alfabeto.

```
let range =
    TableQuery<Customer>().Where(
        TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition(
                "PartitionKey", QueryComparisons.Equal, "Smith"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition(
                "RowKey", QueryComparisons.LessThan, "M")))

let rangeResult = table.ExecuteQuery(range)
```

Ahora imprime los resultados:

```
for customer in rangeResult do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

### una sola entidad

Puede enviar una consulta para recuperar una sola entidad concreta. Aquí se usa un `TableOperation` para especificar el cliente "Ben Smith". En lugar de una colección, obtiene una `Customer`. La forma más rápida de recuperar una sola entidad de la Table service es especificar la clave de partición y la clave de fila en una consulta.

```
let retrieveOp = TableOperation.Retrieve<Customer>("Smith", "Ben")

let retrieveResult = table.Execute(retrieveOp)
```

Ahora imprime los resultados:

```
// Show the result
let retrieveCustomer = retrieveResult.Result :?> Customer
printfn "customer: %A %A" retrieveCustomer.RowKey retrieveCustomer.PartitionKey
```

## una entidad

Para actualizar una entidad, recupere el Table service, modifique el objeto entidad y, a continuación, guarde los cambios de nuevo en el Table service mediante una operación de `Replace`. Esto hace que la entidad se reemplace por completo en el servidor, a menos que la entidad del servidor haya cambiado desde que se recuperó, en cuyo caso se produce un error en la operación. Este error se debe evitar que la aplicación sobrescriba los cambios de otros orígenes accidentalmente.

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0103"
    let replaceOp = TableOperation.Replace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeded")
with e ->
    Console.WriteLine("Update failed")
```

## Inserción o reemplazo de una entidad

A veces, no se sabe si existe una entidad en la tabla. Y, si es así, los valores actuales almacenados en él ya no son necesarios. Puede usar `InsertOrReplace` para crear la entidad o reemplazarla si existe, independientemente de su estado.

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0104"
    let replaceOp = TableOperation.InsertOrReplace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeded")
with e ->
    Console.WriteLine("Update failed")
```

## Consulta de un subconjunto de propiedades de las entidades

Una consulta de tabla puede recuperar solo algunas propiedades de una entidad en lugar de todas ellas. Esta técnica, denominada proyección, puede mejorar el rendimiento de las consultas, especialmente en el caso de entidades de gran tamaño. En este caso, solo se devuelven direcciones de correo electrónico mediante `DynamicTableEntity` y `EntityResolver`. Tenga en cuenta que no se admite la proyección en el emulador de almacenamiento local, por lo que este código solo se ejecuta cuando está usando una cuenta en Table service.

```
// Define the query, and select only the Email property.
let projectionQ = TableQuery<DynamicTableEntity>().Select [|"Email"|]

// Define an entity resolver to work with the entity after retrieval.
let resolver = EntityResolver<string>(fun pk rk ts props etag ->
    if props.ContainsKey("Email") then
        props["Email"].StringValue
    else
        null
)

let resolvedResults = table.ExecuteQuery(projectionQ, resolver, null, null)
```

## Recuperación de entidades en páginas de forma asincrónica

Si está leyendo un gran número de entidades y desea procesarlas a medida que se recuperan en lugar de esperar a que se devuelvan todas, puede usar una consulta segmentada. Aquí se devuelven los resultados en páginas mediante un flujo de trabajo asincrónico para que la ejecución no se bloquee mientras se espera a que se devuelva un conjunto grande de resultados.

```
let tableQ = TableQuery<Customer>()

let asyncQuery =
    let rec loop (cont: TableContinuationToken) = async {
        let! ct = Async.CancellationToken
        let! result = table.ExecuteQuerySegmentedAsync(tableQ, cont, ct) |> Async.AwaitTask

        // ...process the result here...

        // Continue to the next segment
        match result.ContinuationToken with
        | null -> ()
        | cont -> return! loop cont
    }
    loop null
```

Ahora ejecuta este cálculo de forma sincrónica:

```
let asyncResults = asyncQuery |> Async.RunSynchronously
```

## Eliminación de una entidad

Puede eliminar una entidad después de recuperarla. Al igual que con la actualización de una entidad, se produce un error si la entidad ha cambiado desde que se recuperó.

```
let deleteOp = TableOperation.Delete(customer)
table.Execute(deleteOp)
```

## Eliminar una tabla

Puede eliminar una tabla de una cuenta de almacenamiento. Una tabla que se ha eliminada no podrá volver a crearse durante un tiempo tras la eliminación.

```
table.DeleteIfExists()
```

## Pasos siguientes

Ahora que ha aprendido los aspectos básicos del almacenamiento de tablas, siga estos vínculos para obtener más información acerca de las tareas de almacenamiento más complejas y el Table API de Azure Cosmos DB.

- [Introducción a Table API de Azure Cosmos DB](#)
- [Referencia de la biblioteca de clientes de almacenamiento para .NET](#)
- [Azure Storage proveedor de tipos](#)
- [Blog del equipo de Azure Storage](#)
- [Configuración de cadenas de conexión](#)

# Administración de paquetes para las dependencias de Azure de F #

23/10/2019 • 3 minutes to read • [Edit Online](#)

La obtención de paquetes para el desarrollo de Azure es fácil cuando se usa un administrador de paquetes. Las dos opciones son [Paket](#) y [NuGet](#).

## Usar Paket

Si usa [Paket](#) como administrador de dependencias, puede usar la `paket.exe` herramienta para agregar dependencias de Azure. Por ejemplo:

```
> paket add nuget WindowsAzure.Storage
```

O bien, si usa [mono](#) para el desarrollo .net multiplataforma:

```
> mono paket.exe add nuget WindowsAzure.Storage
```

Esto agregará `WindowsAzure.Storage` al conjunto de dependencias de paquete del proyecto en el directorio actual, modificará el `paket.dependencies` archivo y descargará el paquete. Si ha configurado previamente las dependencias o está trabajando con un proyecto en el que otro desarrollador ha configurado las dependencias, puede resolver e instalar las dependencias localmente de la siguiente manera:

```
> paket install
```

O para el desarrollo en mono:

```
> mono paket.exe install
```

Puede actualizar todas las dependencias del paquete a la versión más reciente de la siguiente manera:

```
> paket update
```

O para el desarrollo en mono:

```
> mono paket.exe update
```

## Uso de Nuget

Si usa [NuGet](#) como administrador de dependencias, puede usar la `nuget.exe` herramienta para agregar dependencias de Azure. Por ejemplo:

```
> nuget install WindowsAzure.Storage -ExcludeVersion
```

O para el desarrollo en mono:

```
> mono nuget.exe install WindowsAzure.Storage -ExcludeVersion
```

Esto agregará `WindowsAzure.Storage` al conjunto de dependencias de paquete del proyecto en el directorio actual y descargará el paquete. Si ha configurado previamente las dependencias o está trabajando con un proyecto en el que otro desarrollador ha configurado las dependencias, puede resolver e instalar las dependencias localmente de la siguiente manera:

```
> nuget restore
```

O para el desarrollo en mono:

```
> mono nuget.exe restore
```

Puede actualizar todas las dependencias del paquete a la versión más reciente de la siguiente manera:

```
> nuget update
```

O para el desarrollo en mono:

```
> mono nuget.exe update
```

## Referencia a ensamblados

Para poder usar los paquetes en el F# script, debe hacer referencia a los ensamblados incluidos en los paquetes mediante una `#r` Directiva. Por ejemplo:

```
> #r "packages/WindowsAzure.Storage/lib/net40/Microsoft.WindowsAzure.Storage.dll"
```

Como puede ver, debe especificar la ruta de acceso relativa a la DLL y el nombre completo de la DLL, que puede no ser exactamente igual que el nombre del paquete. La ruta de acceso incluirá una versión de .NET Framework y, posiblemente, un número de versión del paquete. Para encontrar todos los ensamblados instalados, puede usar algo parecido a esto en una línea de comandos de Windows:

```
> cd packages/WindowsAzure.Storage  
> dir /s/b *.dll
```

O en un shell de UNIX, algo parecido a esto:

```
> find packages/WindowsAzure.Storage -name "*.dll"
```

Esto le proporcionará las rutas de acceso a los ensamblados instalados. Desde allí, puede seleccionar la ruta de acceso correcta para la versión de .NET Framework.