F# in Visual Studio

This document describes the experience an F# developer will have inside Visual Studio.

Table of Contents

[1. Introduction 2](#_Toc213148137)

[1.1 Customer Scenarios and Context 2](#_Toc213148138)

[2. Visual Studio Components 3](#_Toc213148139)

[2.1 Editor 3](#_Toc213148140)

[2.1.1 Completion Lists 3](#_Toc213148141)

[2.1.2 Quick Info 5](#_Toc213148142)

[2.1.3 Colorization 6](#_Toc213148143)

[2.1.4 Parameter Help –Limited 6](#_Toc213148144)

[2.2 Project System 7](#_Toc213148145)

[2.2.1 Solution Explorer 7](#_Toc213148146)

[2.2.2 Project Templates 8](#_Toc213148147)

[2.2.3 Item Templates 9](#_Toc213148148)

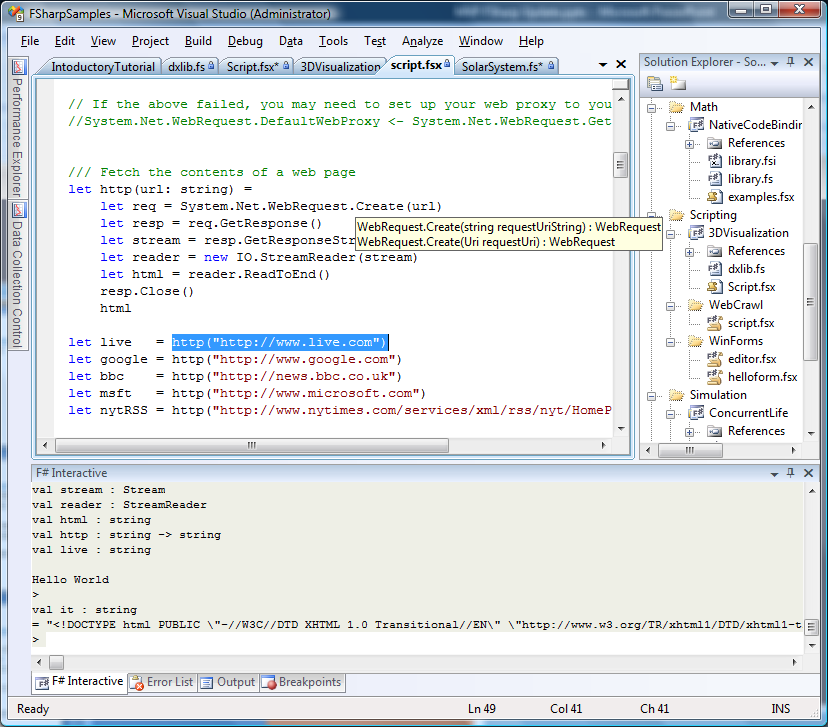
[2.2.4 Project Properties 9](#_Toc213148149)

[3. Visual Studio Features which Do Not Work with F# 12](#_Toc213148150)

[3.1.1 Add Item Templates 12](#_Toc213148151)

# Introduction

F# is integrated into Visual Studio, and F# developers can take advantage of a broad range of the features that Visual Studio provides for writing, managing, debugging, reading and deploying programs written in F#. The remainder of this document walks through the features of Visual Studio, and describes the experience that an F# developer will have with that feature.



## Customer Scenarios and Context

* Andy, professional scientist, developing scripts and small components
* Megan, a tester, doing scripting
* Max, an enterprise software developer, building large application framework with F# on a 20 person team
* Petra, an ISV mixed C#/F# developer

# Visual Studio Components

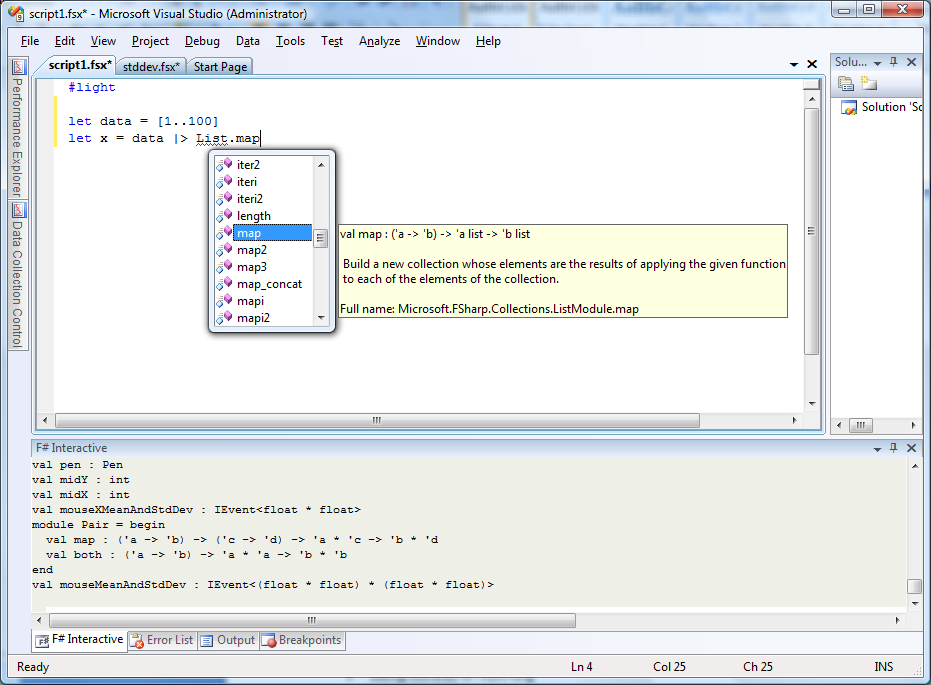
* Editor
* Project System
* Error List
* Output Window
* F# Interactive
* Debugging
* Profiling
* Testing
* Class View and Object Browser

## Editor

The F# editor supports the following features:

* Completion Lists
* Quick Info
* Colorization
* Parameter Help - Limited
* Goto Definition
* Background error reporting
* F1
* Brace Matching

### Completion Lists



The completion list is activated by one of the following actions:

1. Typing “.” In the editor
   1. Only activates a completion list when in a context where a dotted name could be legal F# code.
   2. There is no item selected by default.
2. Executing the “Edit.ListMembers” command
   1. Activates a completion list when the cursor is on any legal F# identifier
   2. If the legal identifier is a prefix of one or more of the items in the completion list, pre-selects the earliest of these in the list

The set of items in the completion list is:

1. If the identifier is part of a dotted name, but not the root of the name
   1. The appropriate list of “members” is shown. This may be sub-namespaces, types or modules in a namespace, members of a module, members of a type, etc. depending on the context.
2. If the identifier is not part of a dotted name, or is the root of a dotted name
   1. The list of all globally scoped names is shown. This includes all top-level namespaces from referenced assemblies, and all names which are in scope according to the rules in the F# language specification.

Once the completion list is active, the following actions can be taken:

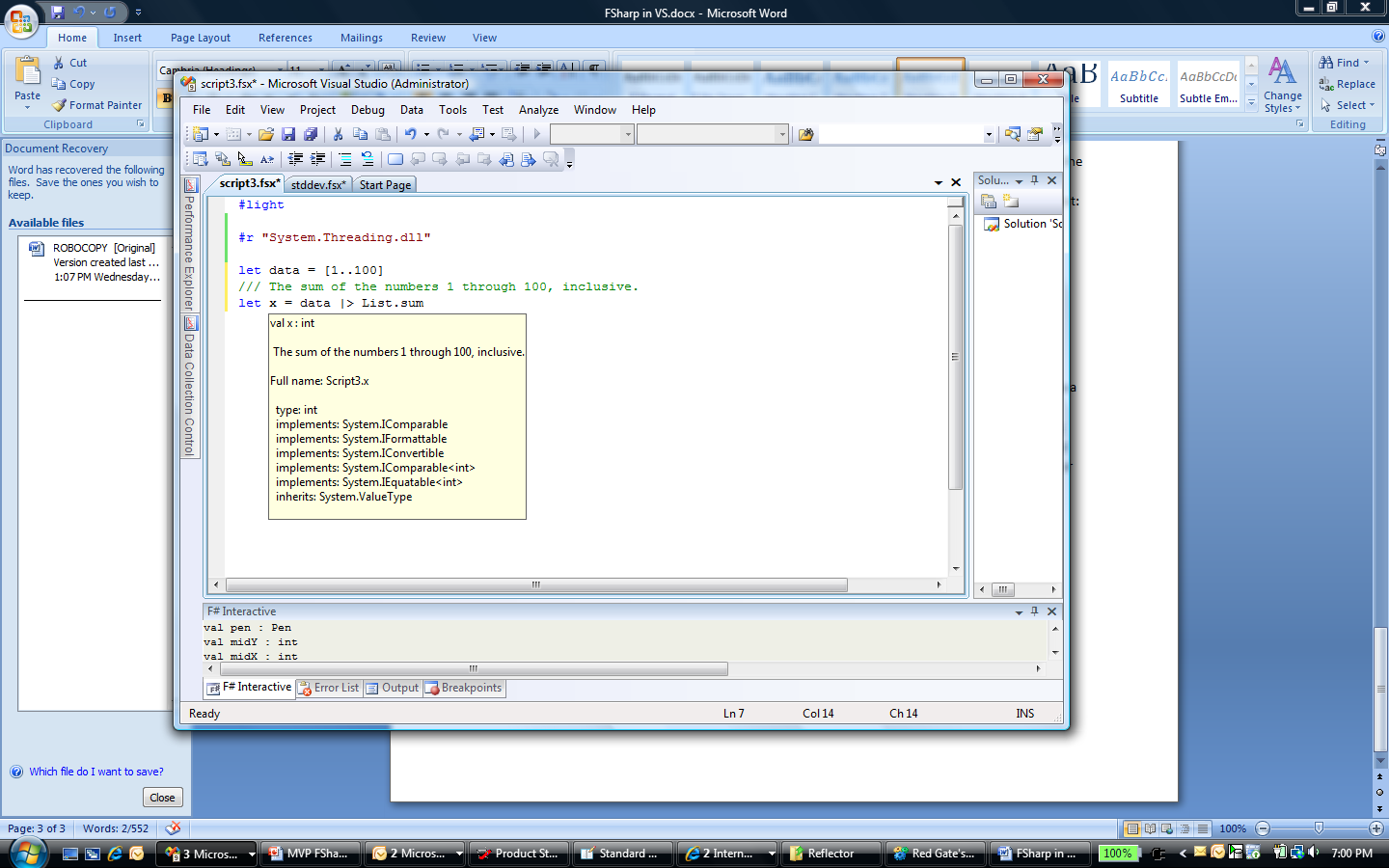
1. The up and down keys, and the mouse, can be used to move the selection around the completion list.
   1. When a selection is changed, a “quick info” for the selected item is displayed next to the item
2. The following keys will commit the selection in the completion list and close the completion list:
   1. Any character which is not part of a legal F# identifier, except escape
3. The following key will close the completion list without committing the selection
   1. <esc>

Notes:

* The items in the list are not filtered beyond this based on any other contextual information.
* The descriptions above are dependent on the current state of the background type checker. This means the exact behavior can depend on:
  + Timing: The background type checker needs to process changes before it can provide a correct completion list.
  + Error Recovery: In code with errors (which is almost always the case while authoring new code), the background typechecker may not always make optimal guesses about what the code is intending t o express. This can result in “incorrect” completion lists or lack of any completion list).
* Error shown in completion list when needed type is not available “<Note>”?

### Quick Info

Quick Info shows a brief description of the given program element.



Quick Info is activated by either:

1. Hovering over an identifier in an F# program
2. Executing the “Edit.QuickInfo” command

The information displayed in QuickInfo is determined based on a given location and the most recent state of the type-checking of the file. Because of the latter, it is possible for QuickInfo results to vary even if no code changes are made. It is a goal for the feature that this is not frequently observed in practice.

**Determining the Symbol(s) to Display**

The fully-qualified identifier at the given location determines a *position*, *long-identifier* and *last-identifier.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Nearby Text and Position** | **Long-identifier** | **Last-identifier** |
| **Quick Info** | Class | - | Class |
| **Quick Info** | C.Method | C | Method |
| **Quick Info** | C<int>.Method | - | Method |
| **+ many other odd cases** |  |  |  |

If

* The position at the end of the last identifier precisely matches a position at the end of a *precise name resolution* that occurred during the most recent type-checking of the file
* AND the display name of that resolution precisely matches the last identifier

then use this single item as the resolution.

The Quick Info for any single resolution for a name contains the following sections:

1. The pretty-print of the signature of the program element.
2. (If the value is a .NET constant) “= <val>” where <val> is the string representation of the value of the constant
3. (If available) The XML documentation associated with the element. This contains:
   1. The formatted contents of the <summary> section
   2. A listing of exceptions from the <exception> sections
4. The fully qualified name
5. The type hierarchy for the type being shown, or the type of a variable being shown
   1. Includes the name of the type, a list of interfaces it implements, and a list of base classes.

If there are multiple resolutions for the given name, each is displayed, with a “-----------“ marker between each entry. This can happen, for example, with generic arity overloads or name clashes between different kinds of names (types vs. variables).

### Colorization

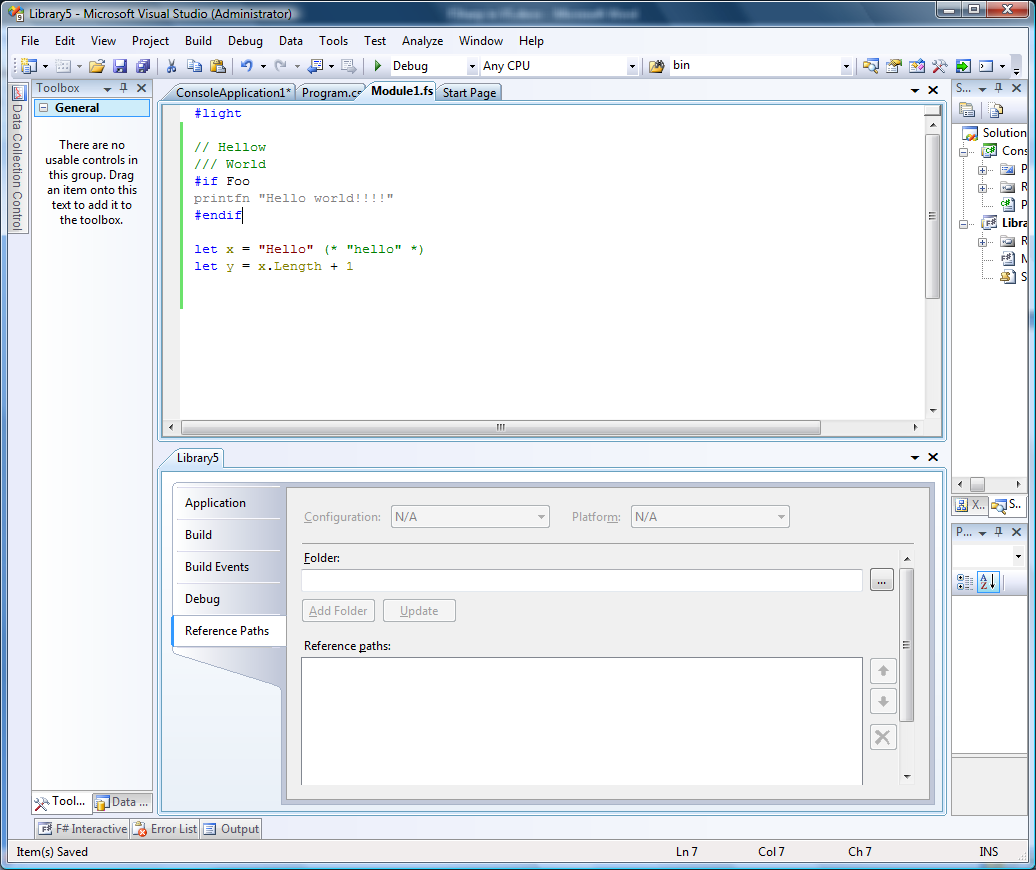
F# colorizes code as it is typed or viewed in the Visual Studio editor.

The following categories are used:

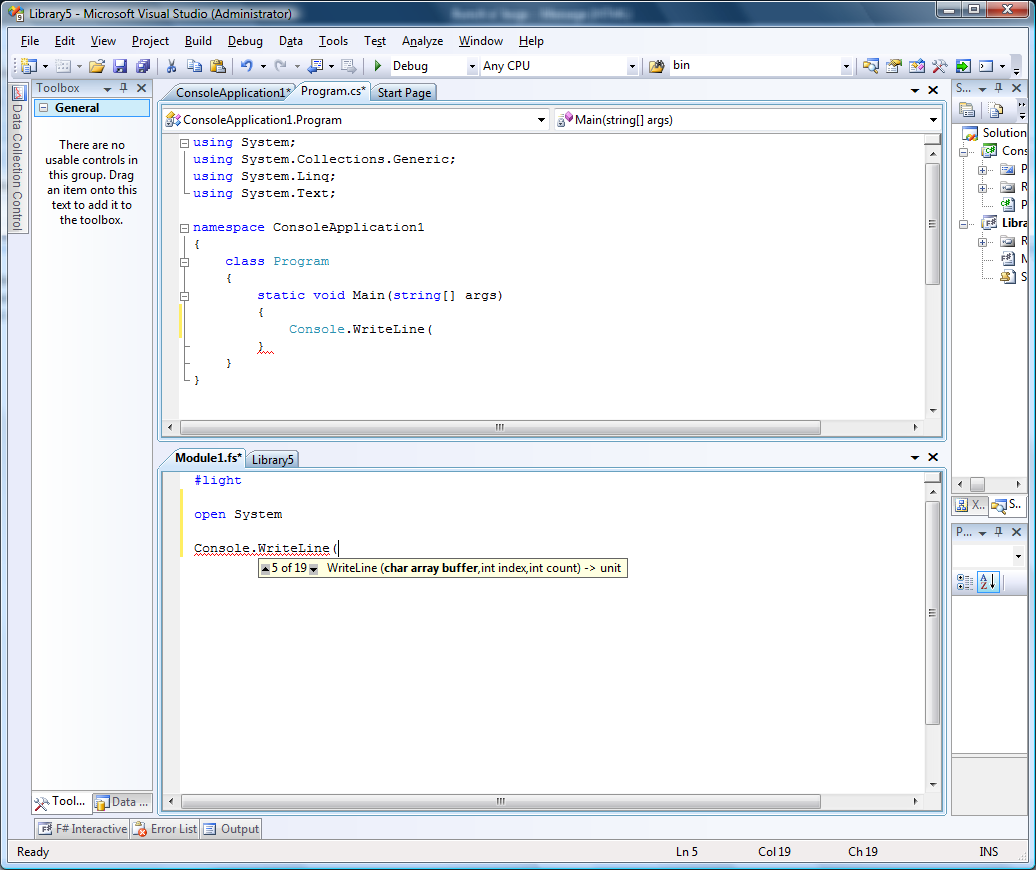
* Comment
* Identifier
* Keyword
* String
* Text
* Number
* Inactive Code
* Preprocessor Keyword

The definitions of each of these construct follow from the F# Language Specification.

Each of these can be customized in Tools->Options using the Display Item of the same name.



### Parameter Help –Limited



Parameter help is invoked when either:

* “(“ is typed immediately following a reference to a known function value or method
* The Edit.ParameterInfo command is executed within a parenthesized list following a reference to a known function value or method

In either case, a tooltip is displayed containing the following information:

* If there are overloads, the number of overloads, and an index representing the current overload
* The signature of the function or method
* A highlighted item in the

## Project System

### Solution Explorer

F# projects are MSBuild files with an .fsproj extension. They can be part of Visual Studio solutions. When opened in Visual Studio, these projects are displayed and can be manipulated through the Solution Explorer. Overall the behavior will be the same as C# or VB projects. There are the following exceptions:

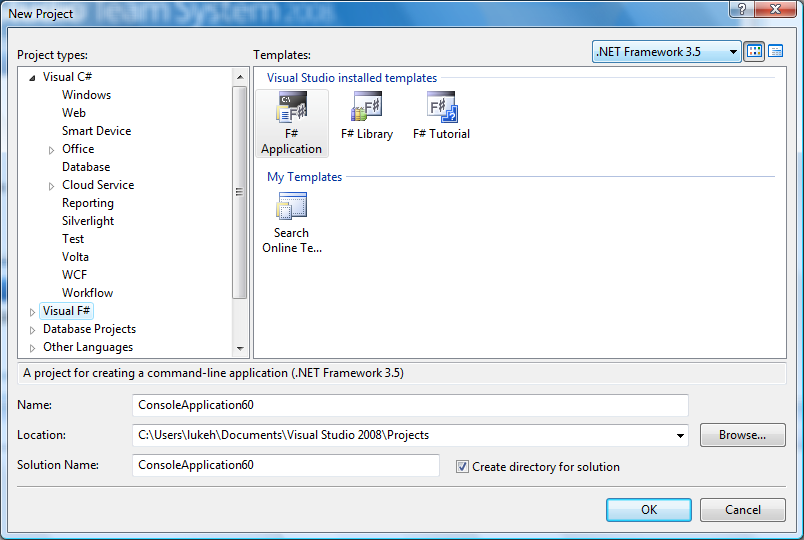
**Ordering**

The order of files in an F# project matters. This results in the following features:

* Project items are not sorted alphabetically.
* There are a few F#-specific commands available in the project item context menu
  + Move Up (alt-up)
  + Move Down (alt-down)
  + Add Above >
    - New Item…
    - Existing Item…
  + Add Below >
    - New Item…
    - Existing Item…
* There is no support for adding folders to F# projects

No Designers

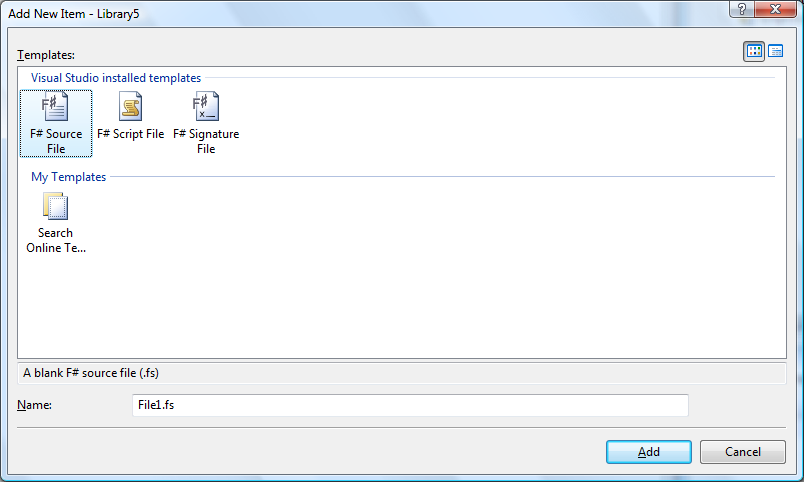
### Project Templates



Three templates are available:

* **F# Application**: A “Console Application” containing a single .fs file
* **F# Library**: A “Class Library” containing a single .fs file and a single .fsx file configured to load and open the contents of the .fs file.
* **F# Tutorial**: A tour of some basic F# language constructs

### Item Templates



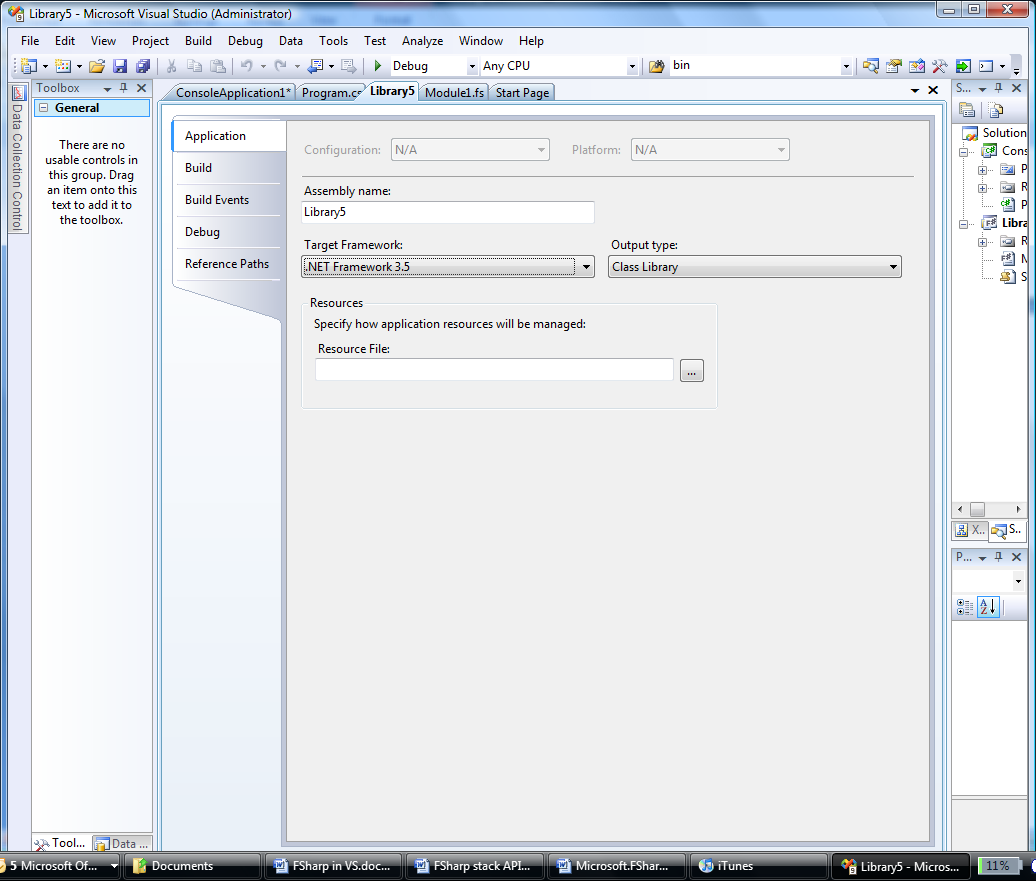
Three templates are available:

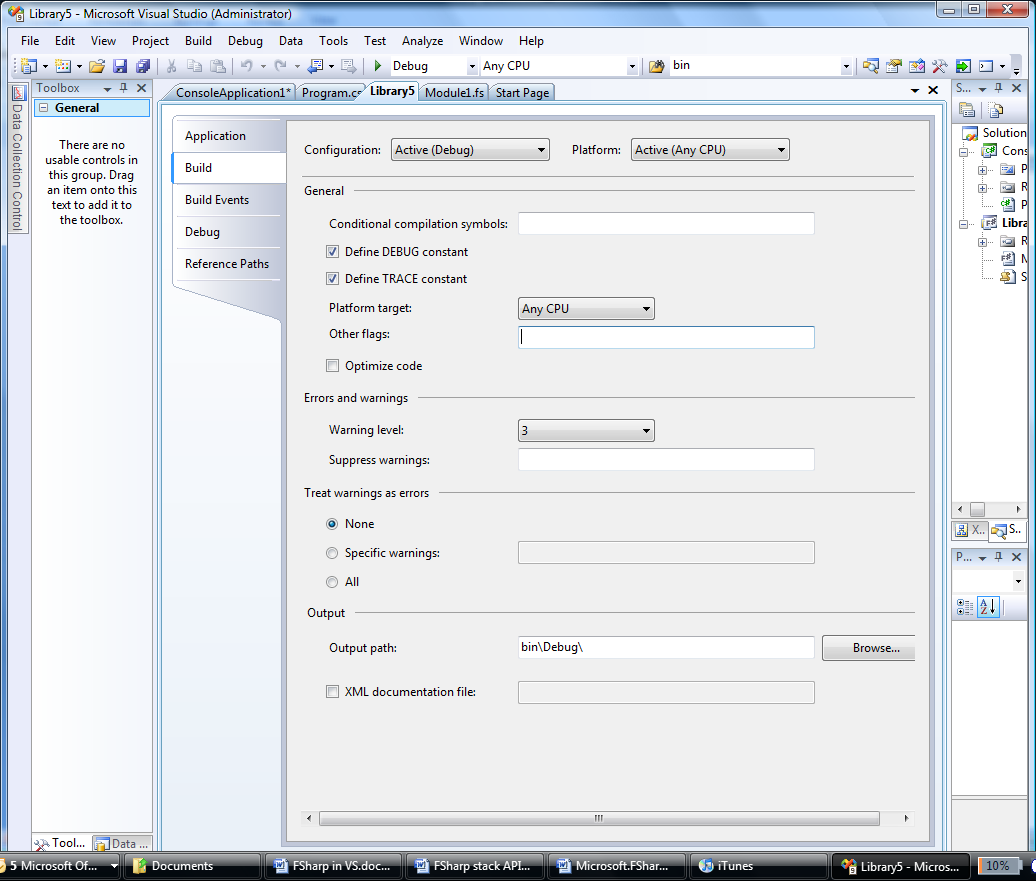
* **F# Source File**: A blank “.fs” file
* **F# Script File**: A blank “.fsx” file
* **F# Signature File**: A blank “.fsi” file

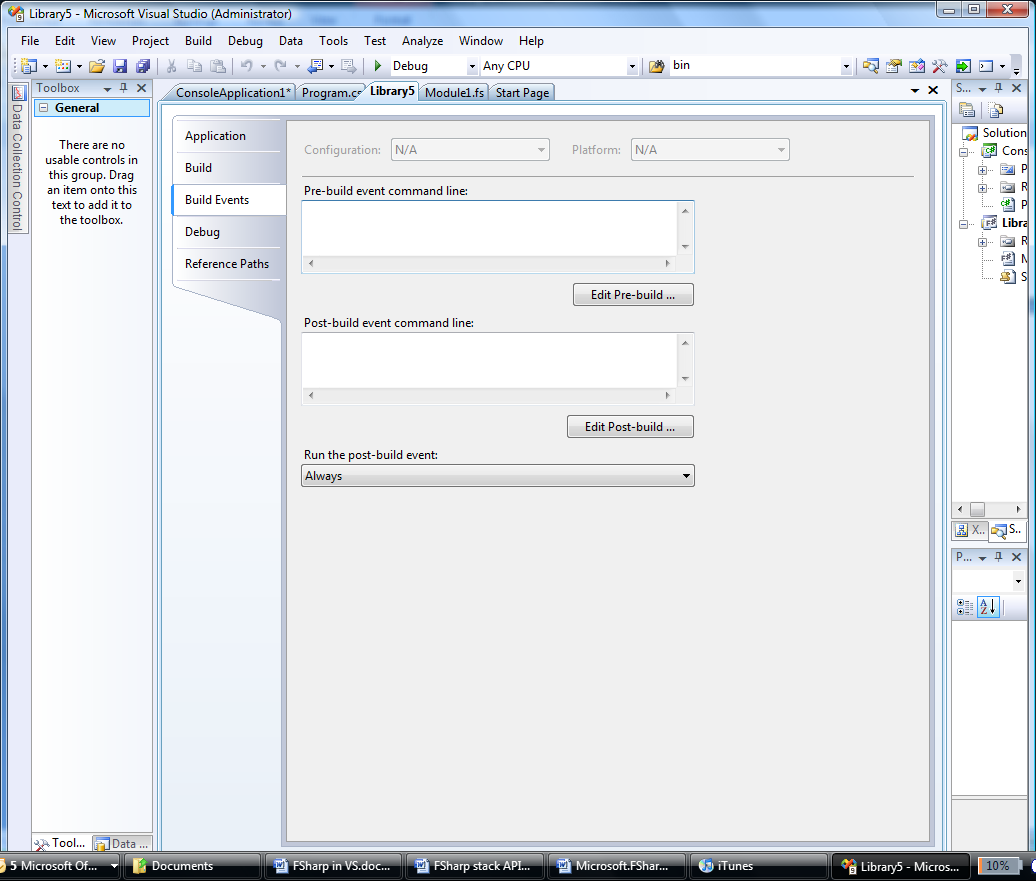
### Project Properties

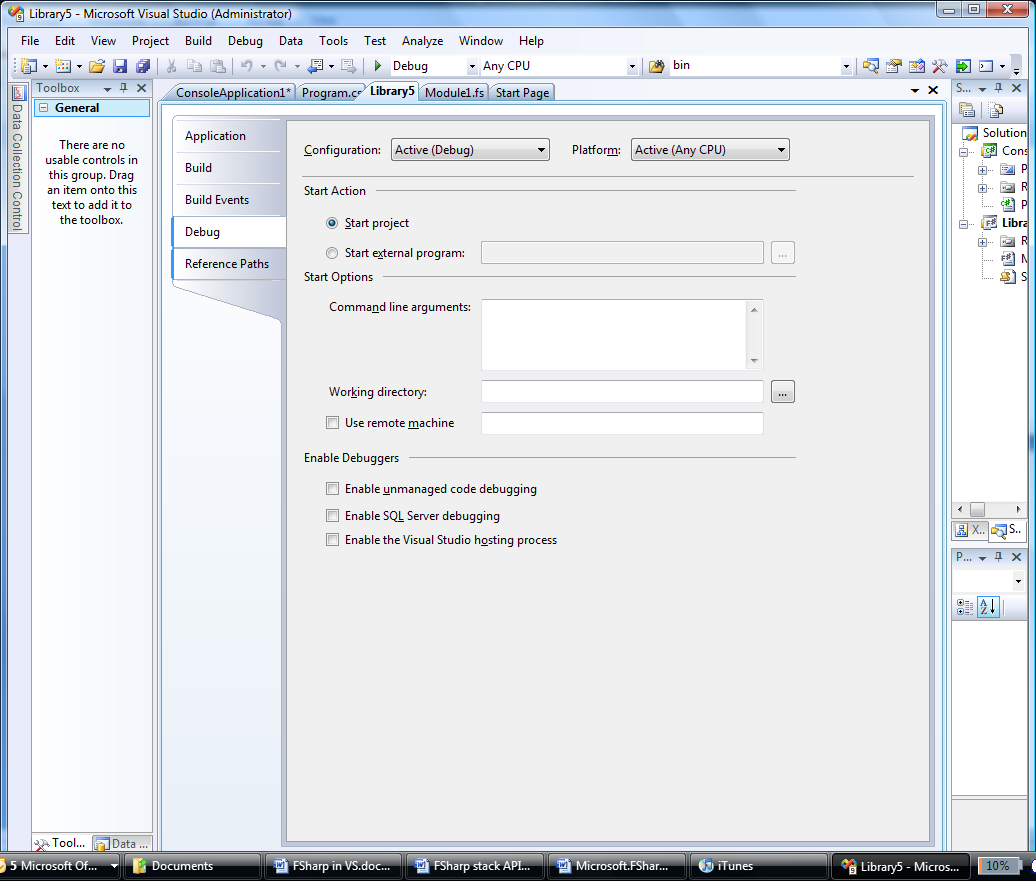
F# project properties can be manipulated using the 5 Project Properties pages shown below. The settings on these pages modify information which will be persisted in the .fsproj file. The behavior of the options on these pages is the same as for C#, expect:

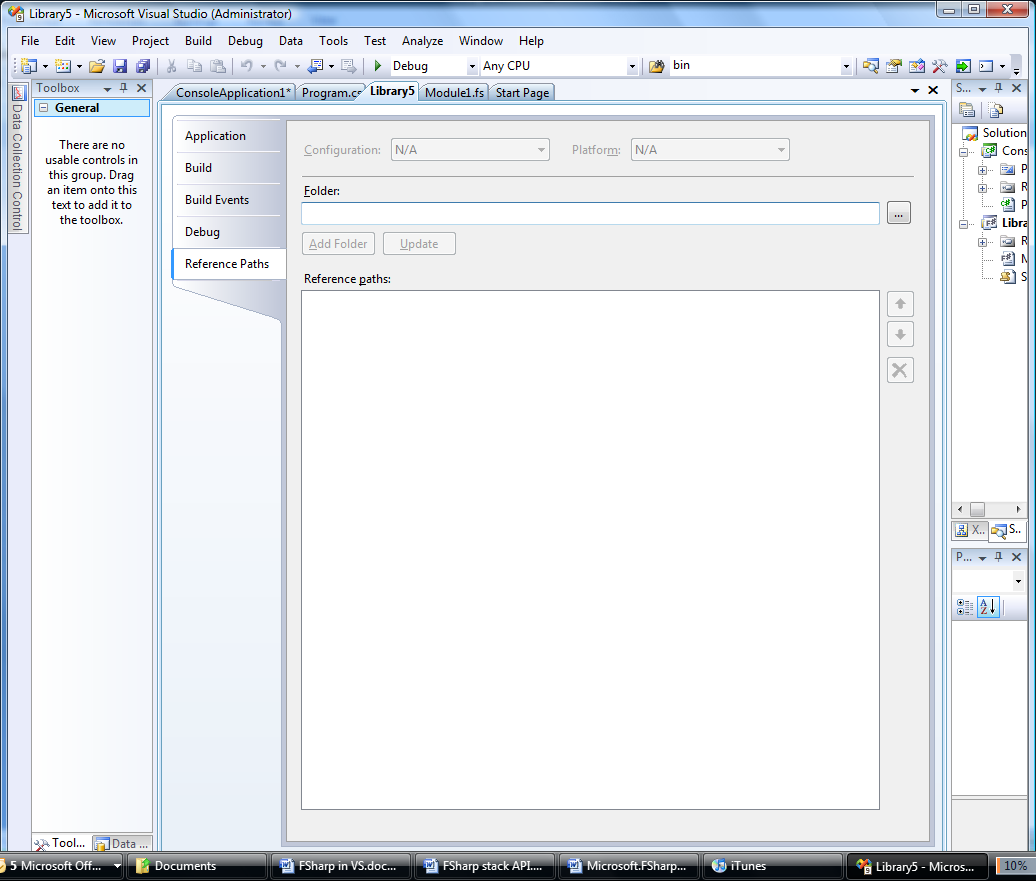
**Other Flags**: This setting allows additional command line flags to be passed to the F# compiler.











**Not available:**

Pages:

* Resources Page
* Services Page
* Settings Page
* Signing Page
* Security Page
* Publish Page
* Code Analysis Page

On the Application page:

* Default Namespace
* Client-framework only
* Startup object
* Assembly Information
* Icon and Manifest embedding

On the Build page:

* Allow unsafe code
* Register for COM interop
* Generate serialization assembly
* Advanced

On the Debug page:

* Start browser with URL

## 2.3 F# Interactive

The F# Interactive provides an explorative Read-Eval-Print-Loop environment for programming in F#. It is a standard Visual Studio toolwindow.

The F# Interactive

# Visual Studio Features which Do Not Work with F#

### Add Item Templates

The following list includes the templates available for C# applications. The items in green have equivalents available in F#. The items in yellow do not, but could be added easily. The items in red are intentionally missing to reduce designer surface area.

* Database Unit Test
* ADO.NET Entity Data Model
* Application Manifest File
* Bitmap File
* Class Diagram
* Component Class
* Custom Control
* Debugger Visualizer
* HTML Page
* lnstaller Class
* JScript File
* Local Database
* MDI Parent Form
* Report Wizard
* Service-based Database Style Sheet
* User Control
* WCF Service
* Windows Script Host
* XML File
* XSLT File
* About Box
* Assembly Information File
* Class
* Code File
* Cursor File
* Data Set
* Dynamic Data Field
* Icon File
* Interface
* LINQ to SQL Classes
* Local Database Cache
* Report
* Resources File
* Settings File
* Text File
* User Control (WPF)
* Windows Form
* Windows Service
* XML Schema