

Handle Exceptions in Methods

Contents

Handling of exceptions	1
Catch Exceptions	2
Create a sample application.....	2
Add an action that generates exception.....	6
Capture specific types of exceptions	10
Catch all exceptions.....	15
Actions guaranteed to be executed	19

Handling of exceptions

Some actions may generate errors. For example, opening a file but giving an invalid file path.

Such action errors are called exceptions. An unhandled exception will crash the application.

To prevent such crashes, all UI level event handler methods, such as a button click event handler method, should handle all exceptions.

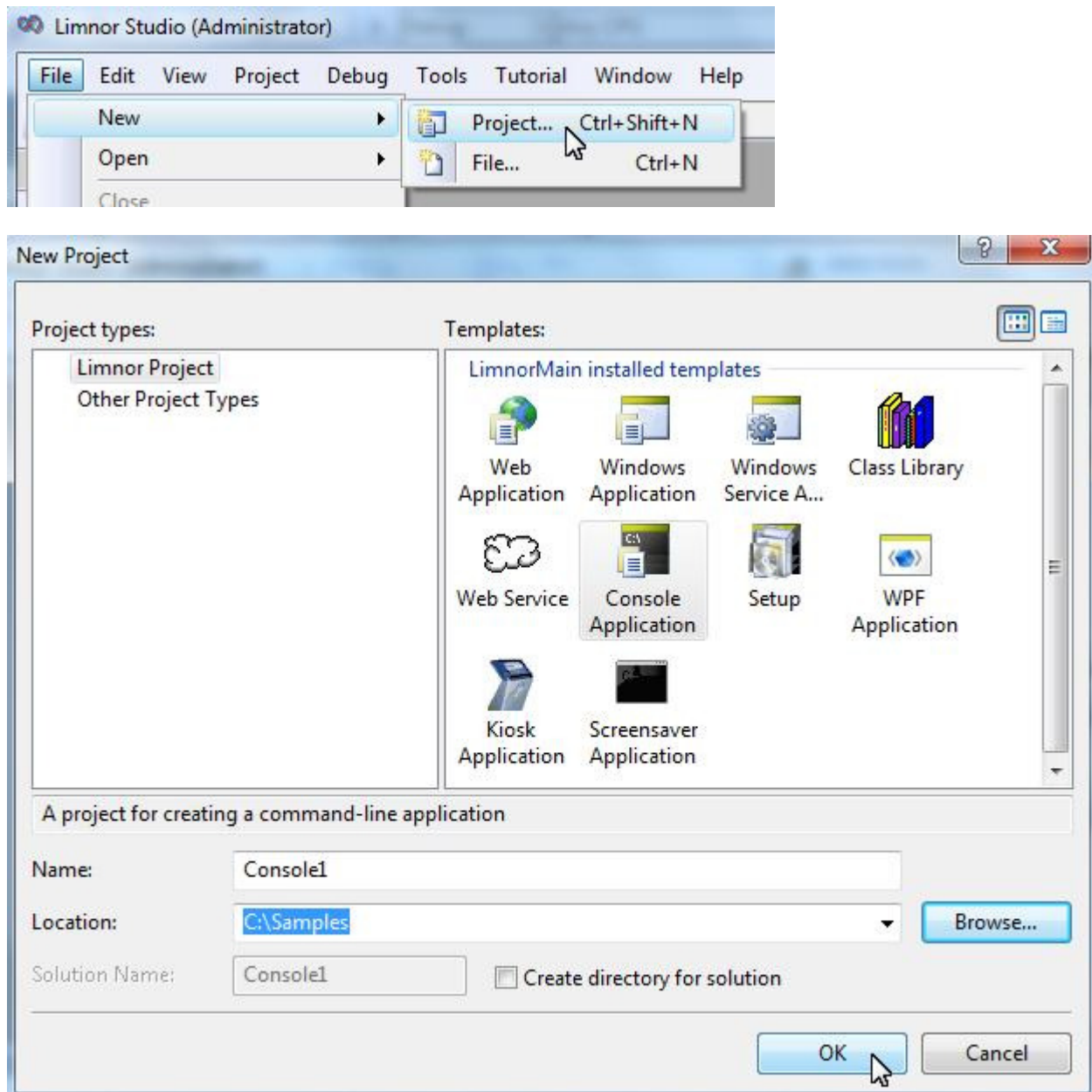
Other types of methods do not need to handle all exceptions. Unnecessarily handling of exceptions will hide the programming errors and make it very difficult to diagnose programming defects.

For example, suppose a method generates an exception and is handled by logging the exception. A user detects that the program works incorrectly and sees the logging of the exception. Thus it is known that it is the method causes the error. There may be hundreds of paths leading to the call of the method. Thus it will be difficult to diagnose. Suppose the method dose not handle the exception, the exception will pop up along the path of execution and be caught at the top level of execution, providing rich debugging information including a specific execution path.

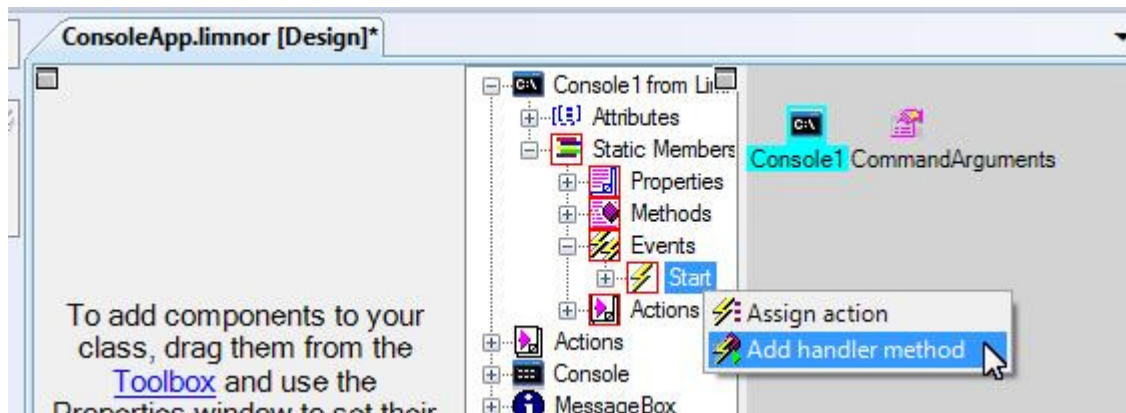
Catch Exceptions

Create a sample application

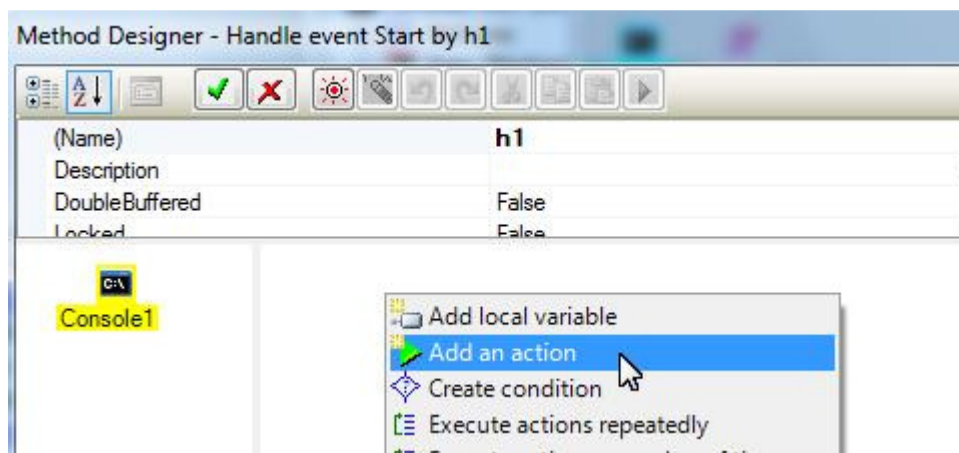
We create a console application to demonstrate the handling of exceptions.



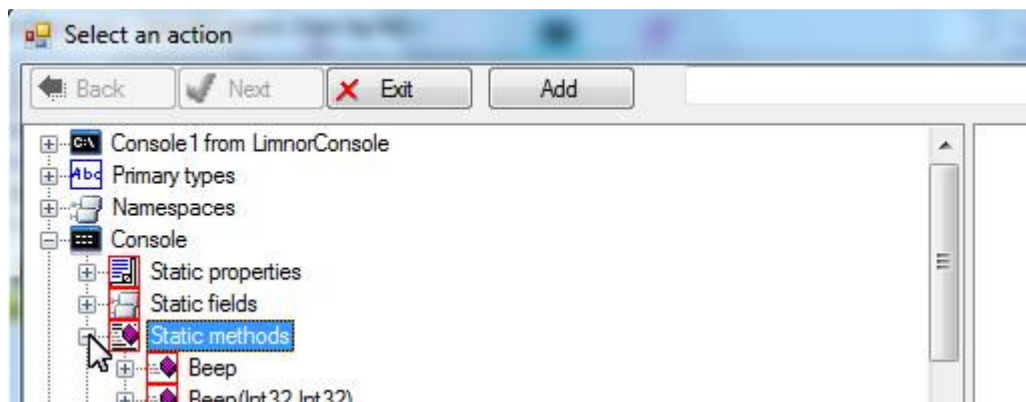
Create an event handler method for the Start event:

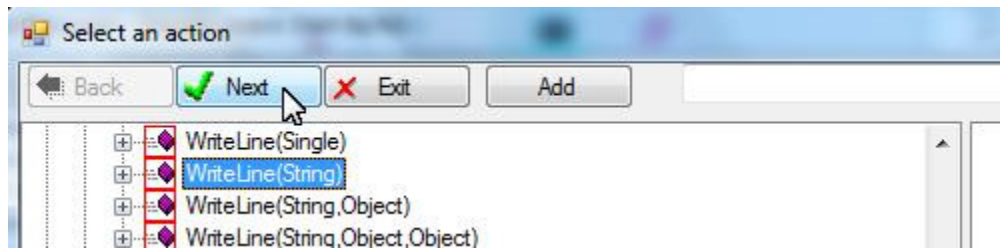


The Method Editor appears. Add a new action:

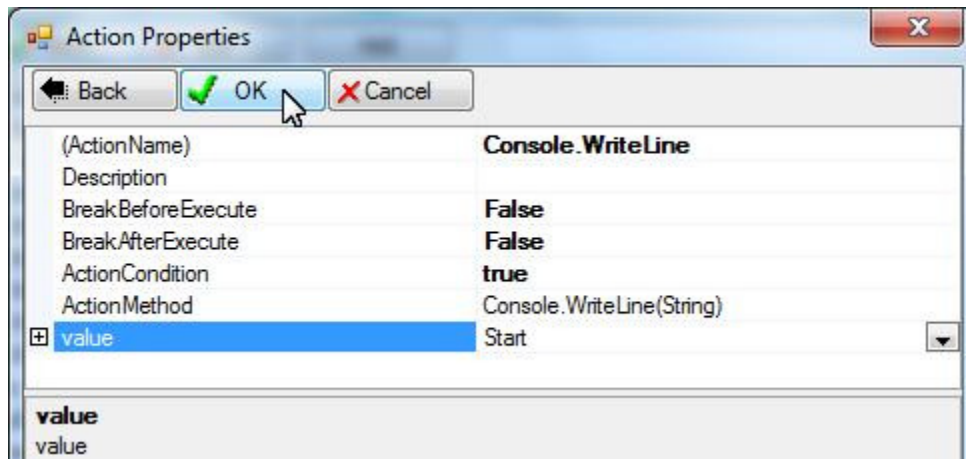


We want to use the Console class to write a line to the console:

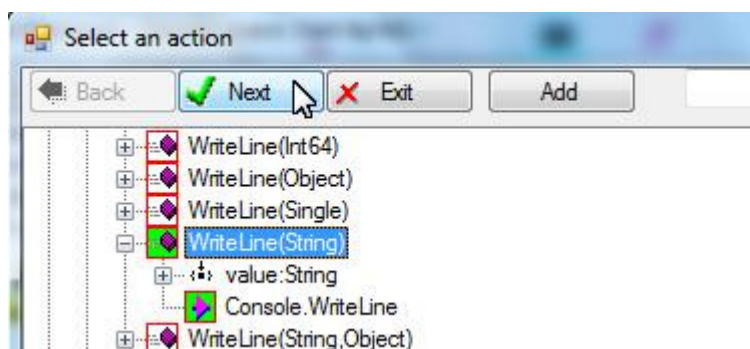
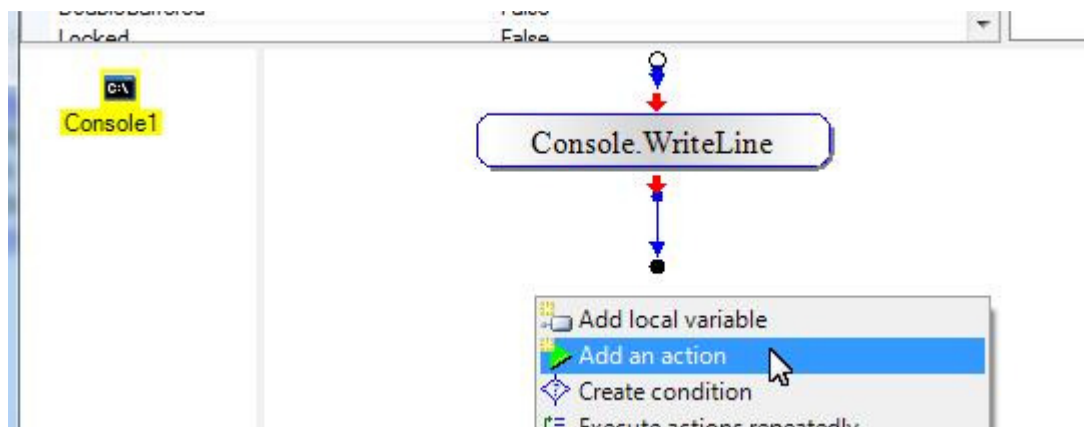




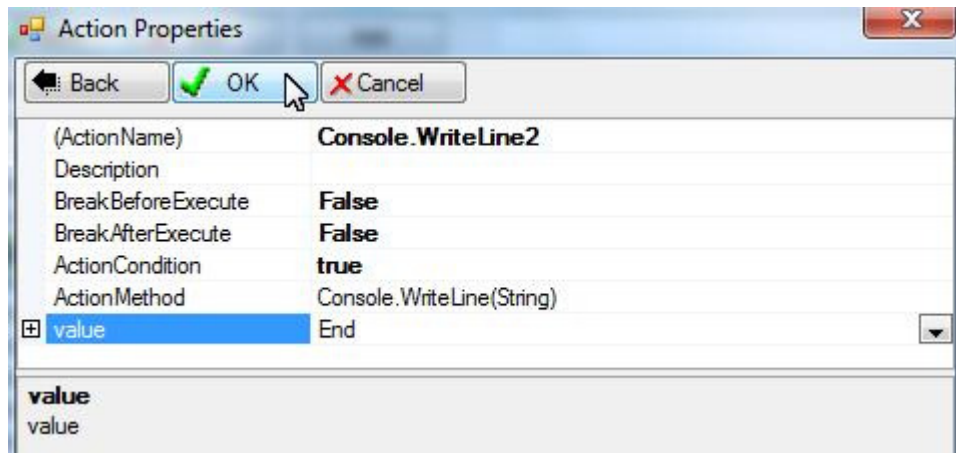
Type "Start" to the "value" parameter of this action:



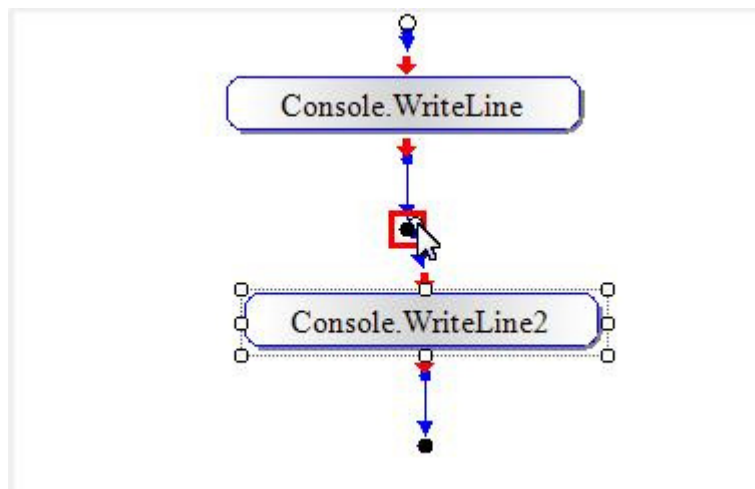
Add another WriteLine action:



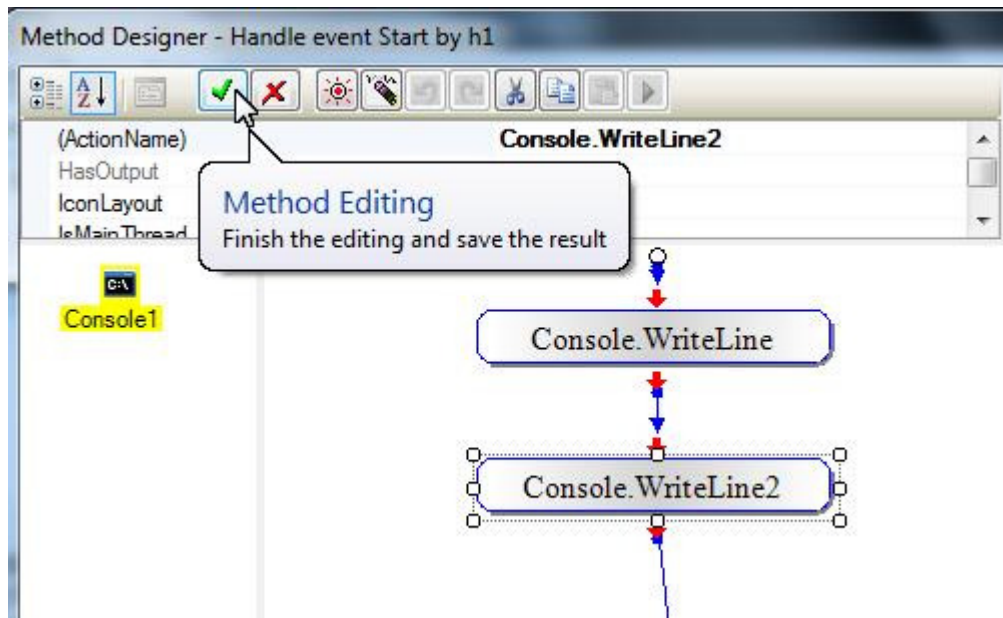
Type “End” to the “value” parameter of this action:



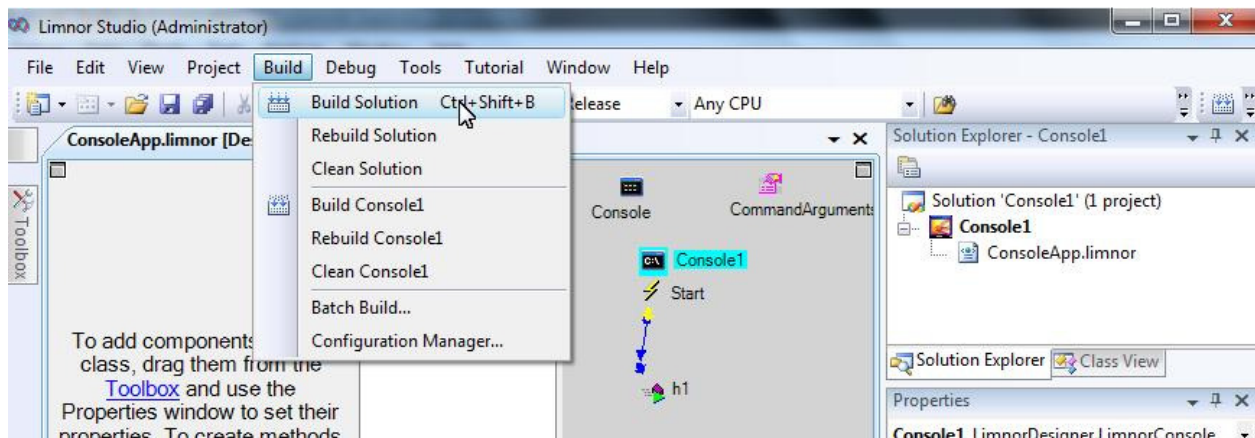
Link the two actions together:



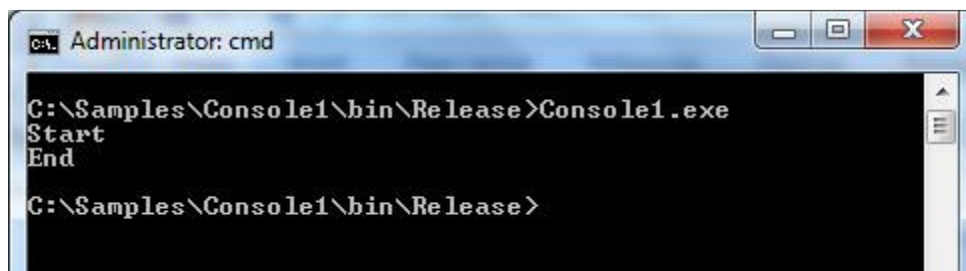
This is our sample event handler method:



Build the project:

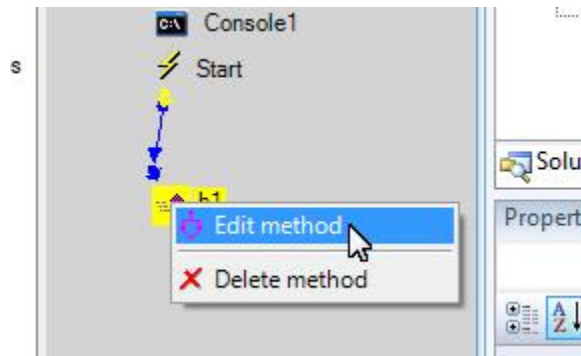


Run the program, Console1.exe, two lines, "Start" and "End", are written to the console.

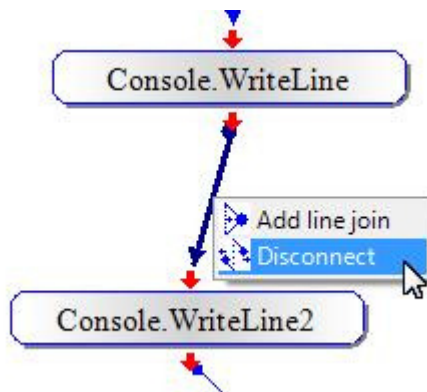


Add an action that generates exception

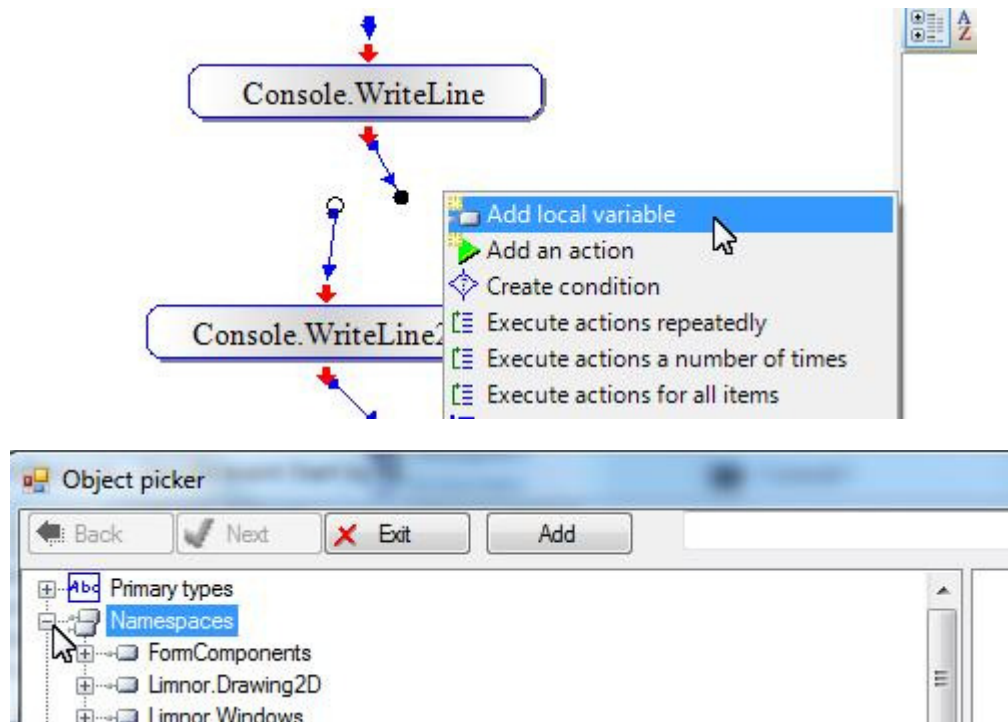
Edit the event handler method.

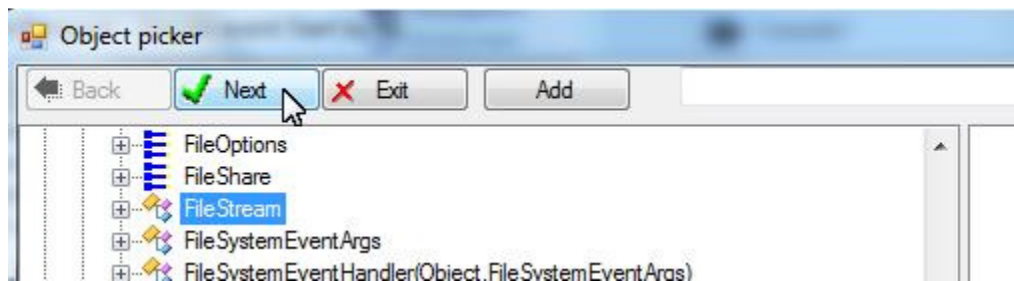
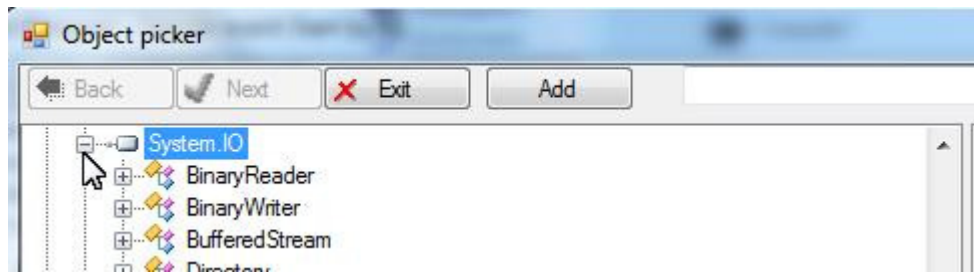


Break the action link so that we may insert a new action:

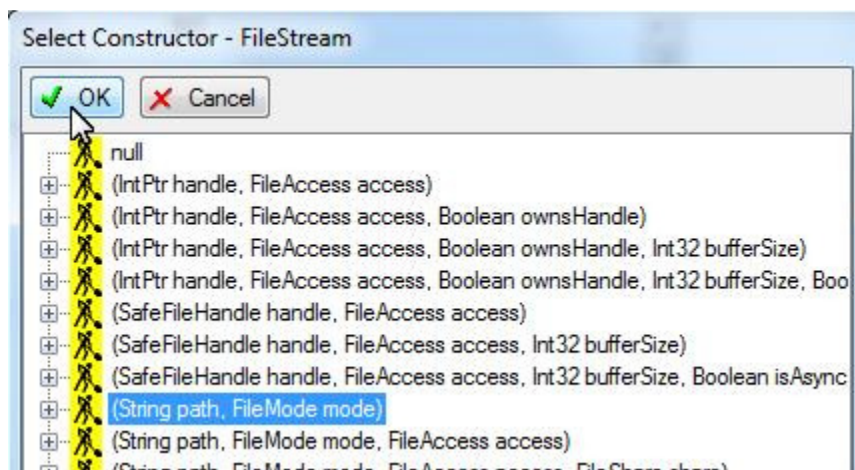


Add a FileStream variable to open a file:

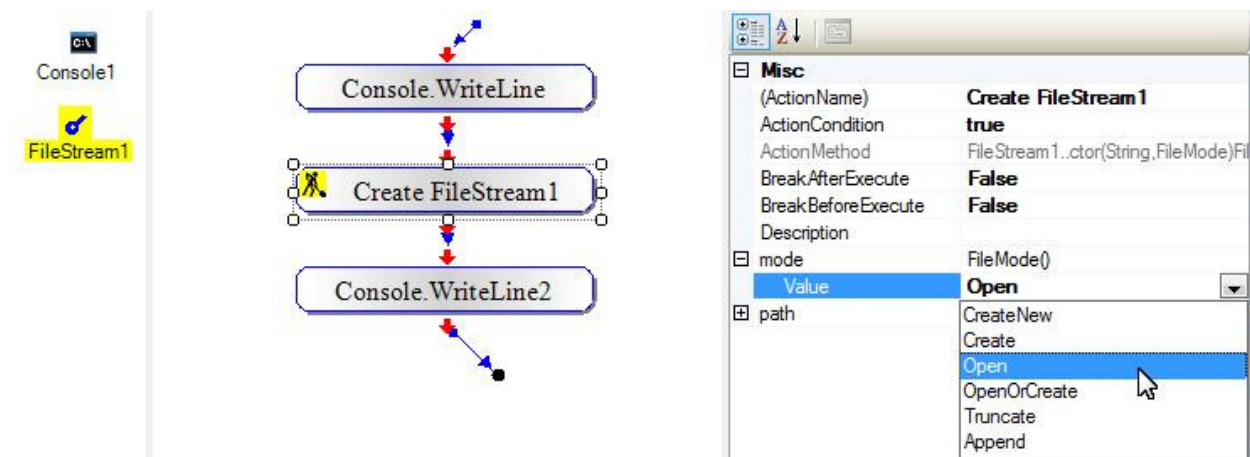




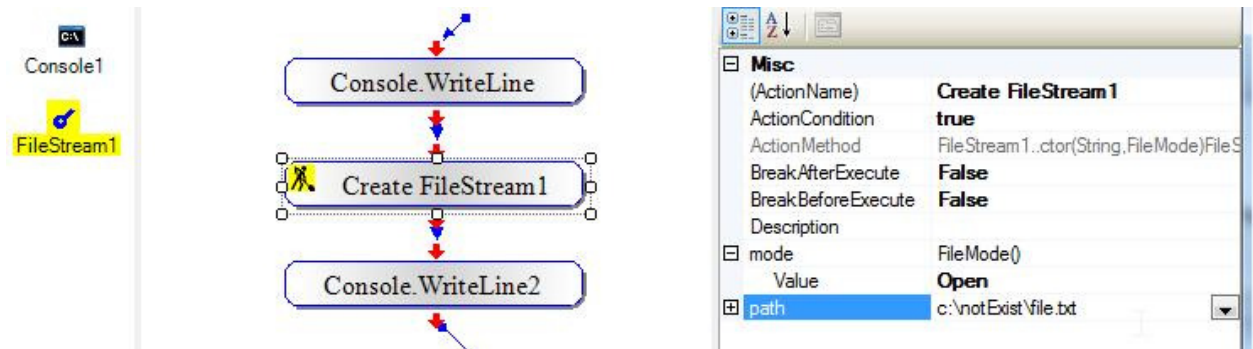
Select a constructor:



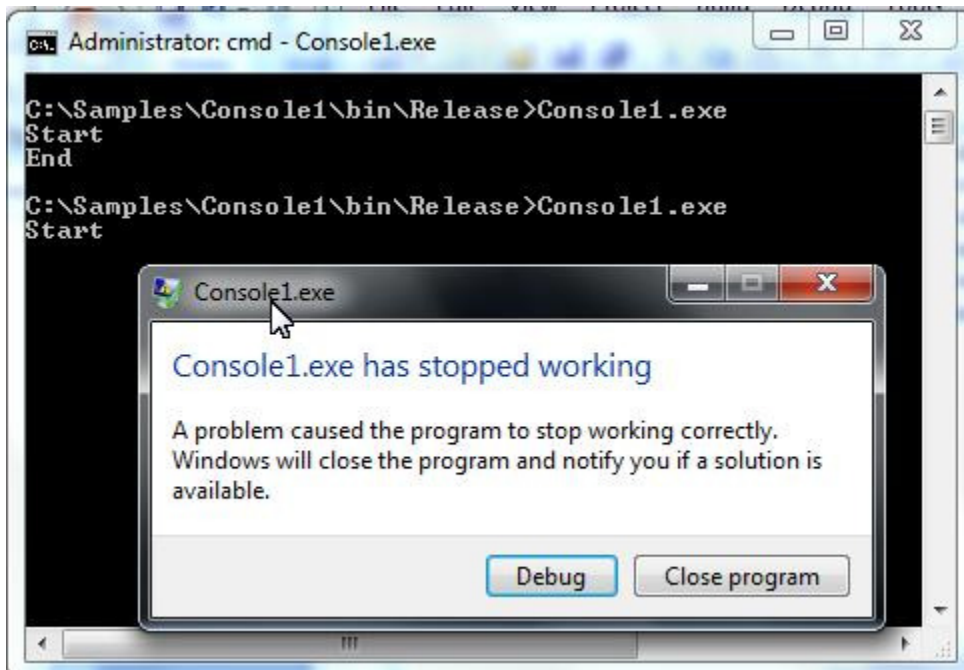
Insert the constructor action between the two actions. Set "mode" to Open for opening an existing file:



Give “path” an invalid file path:



Compile the project and run the program. Because the file does not exist, an exception occurs. The exception crashes the application.



We can see that the first action executes and show “Start” in the console. Then an exception occurs. The last action will not get executed and thus we will not see “End” in the console.

Click “Close program”, the exception is displayed in the console by the runtime engine.

```

Administrator: cmd

C:\Samples\Console1\bin\Release>Console1.exe
Start
End

C:\Samples\Console1\bin\Release>Console1.exe
Start


Unhandled Exception: System.IO.DirectoryNotFoundException
of the path 'c:\notExist\file.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String
   at System.IO.FileStream.Init(String path, FileMode mo
   nt32 rights, Boolean useRights, FileShare share, Int32 b
   ptions, SECURITY_ATTRIBUTES secAttrs, String msgPath, Bo
   at System.IO.FileStream..ctor(String path, FileMode m
   FileShare share, Int32 bufferSize, FileOptions options,
   bFromProxy)
   at System.IO.FileStream..ctor(String path, FileMode m
   at DefaultNamespace.Console1.Main() in c:\Samples\Con
   space.Console1.cs:line 37

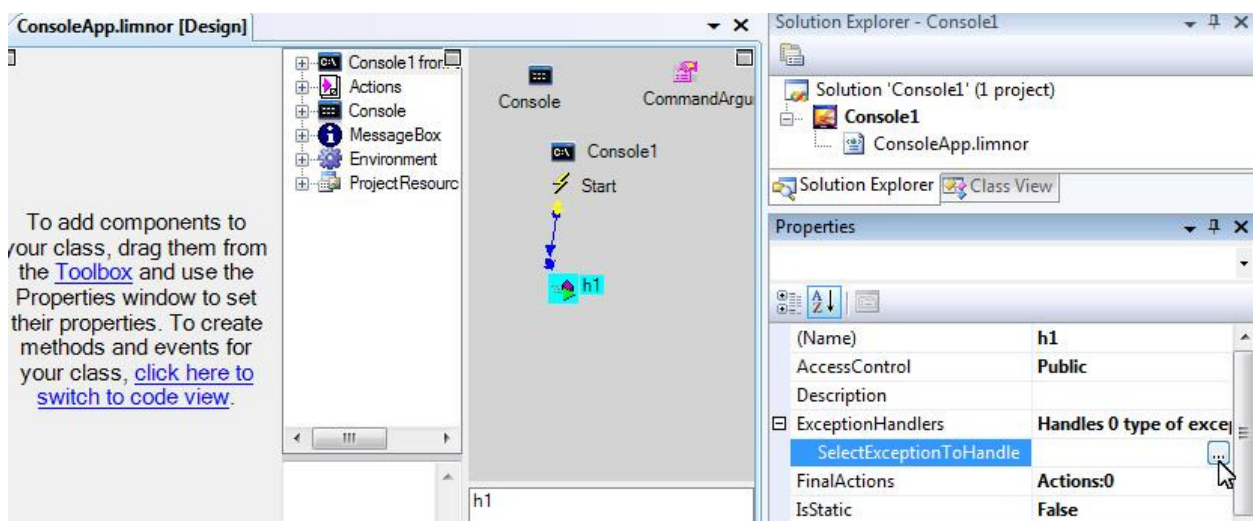
C:\Samples\Console1\bin\Release>
  
```

Capture specific types of exceptions

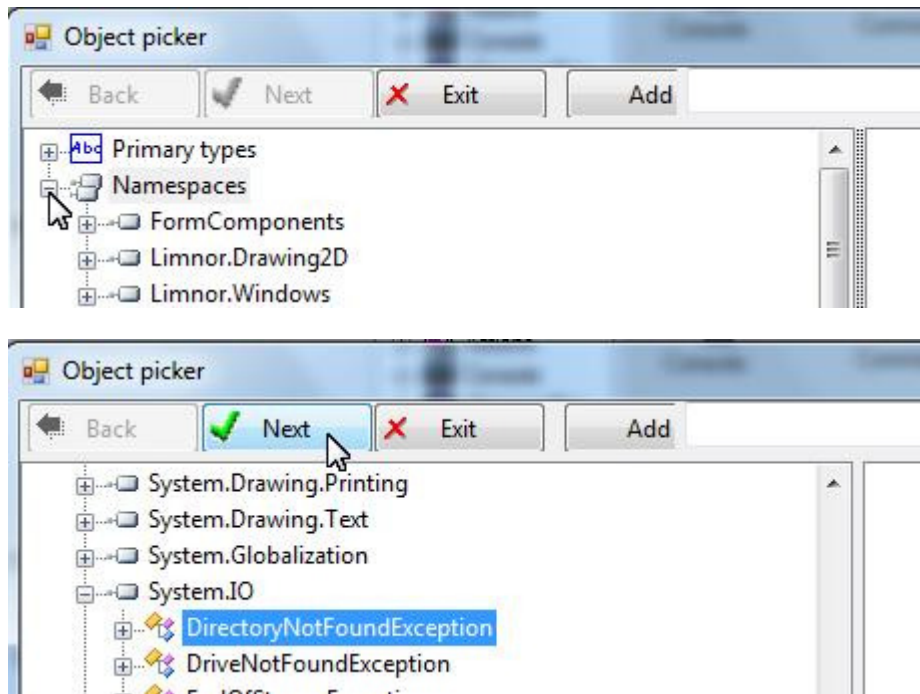
From the last run we can see that we got an exception of `System.IO.DirectoryNotFoundException` type.


If we want to capture this specific type of exceptions then we may set the property `ExceptionHandlers` of the method.

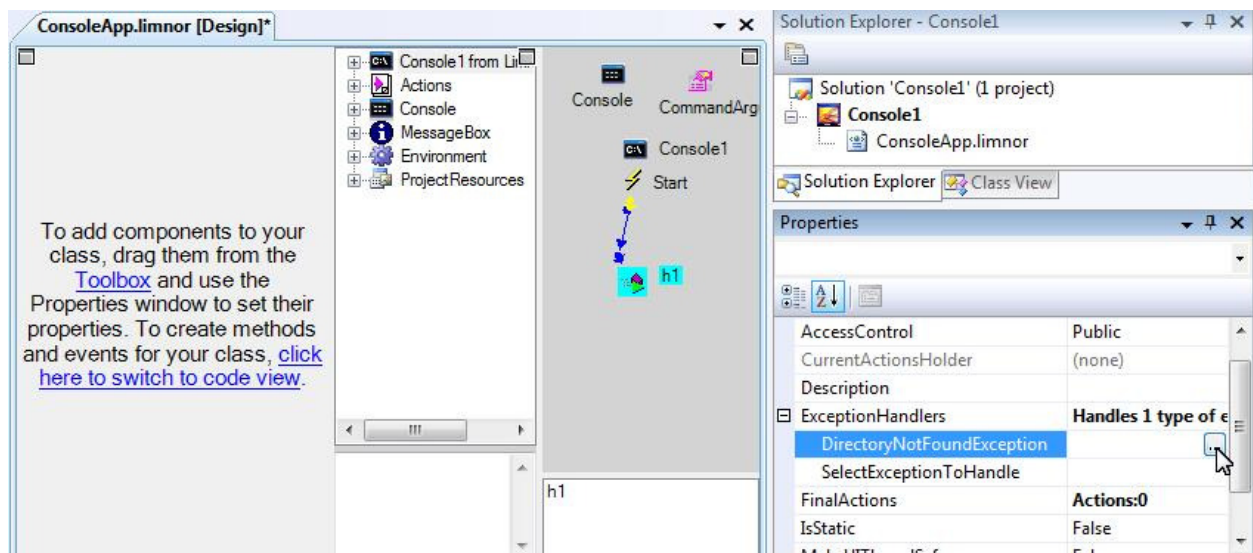
Select the method to display its properties in the Property Grid. Click  beside "SelectExceptionToHandle":



Select `DirectoryNotFoundException` under namespace `System.IO`:

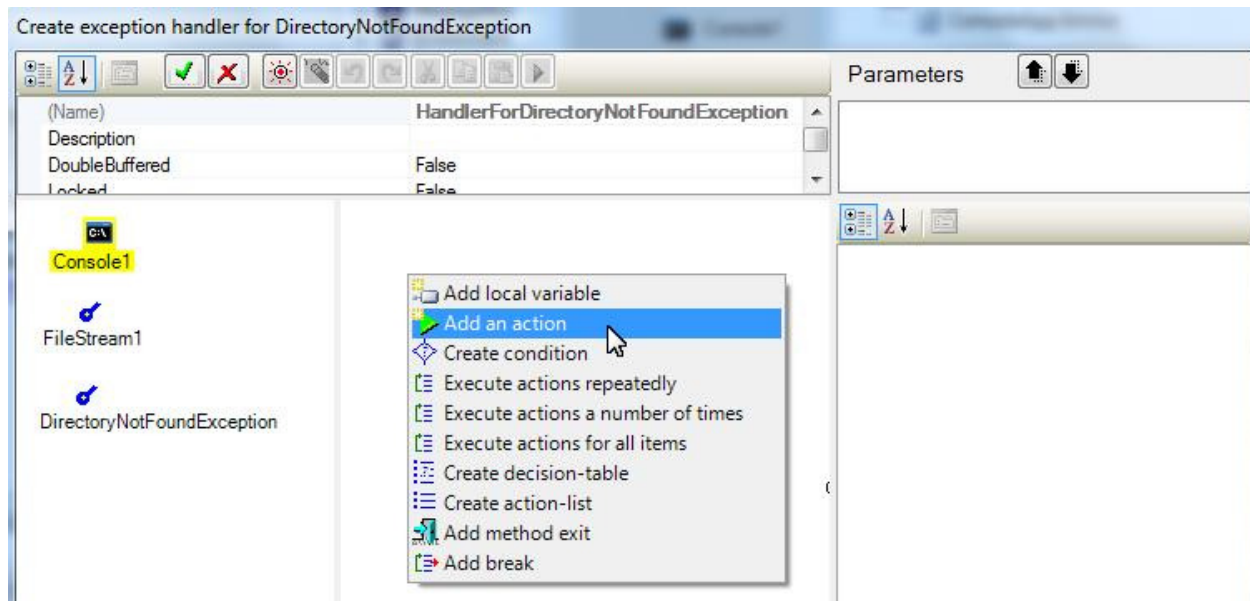


The selected exception type appears in the list. Click  to specify actions for handling this type of exceptions:

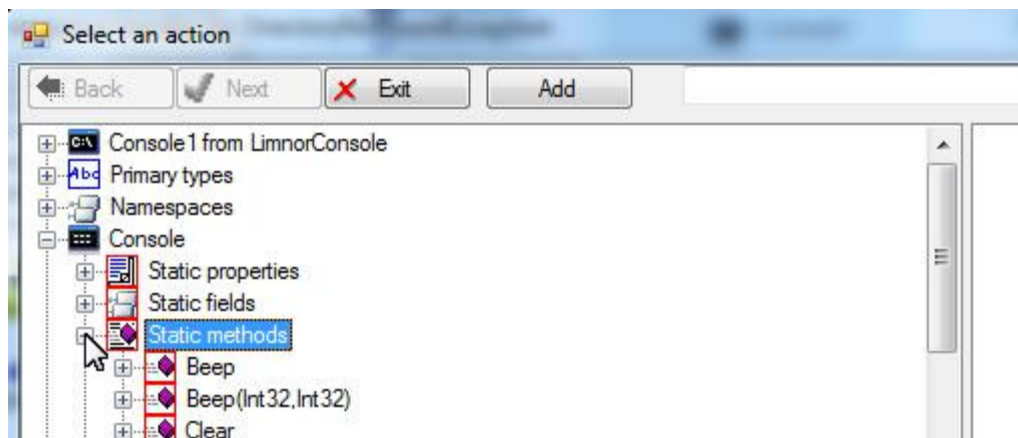


The Method Editor appears. For this sample, we simply output the exception message to the console.

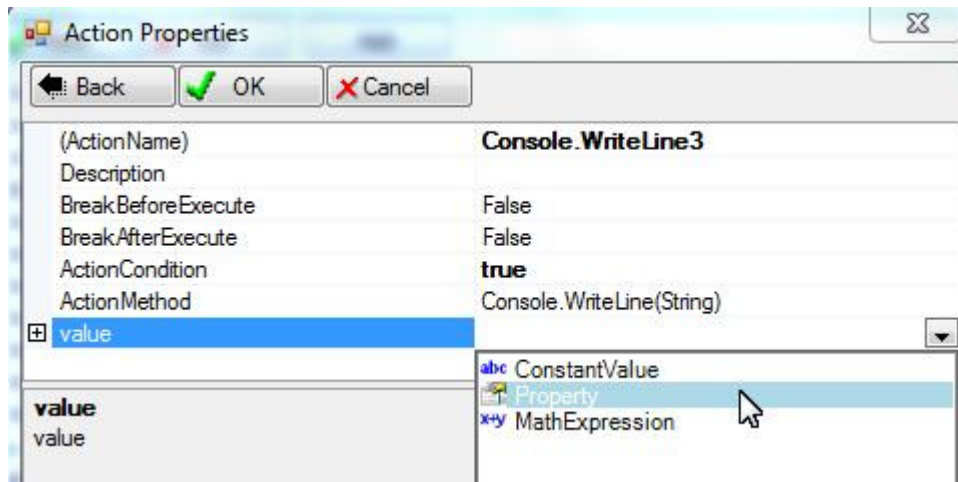
Right-click the Action Pane; choose “Add an action”:



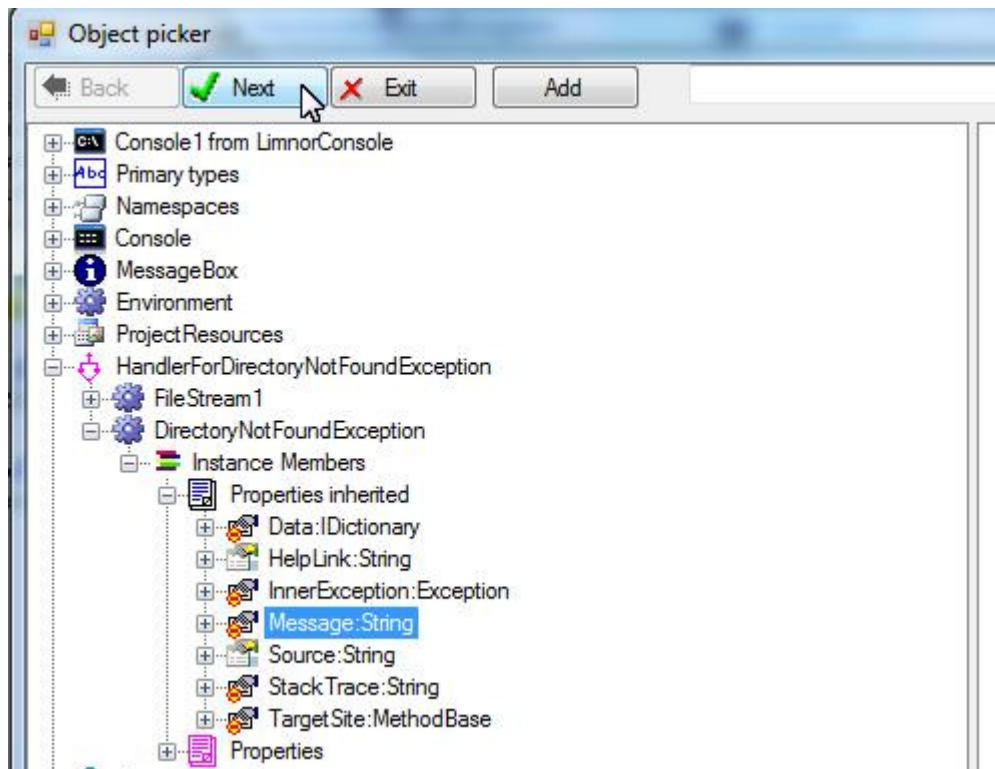
Select WriteLine method of the Console class:

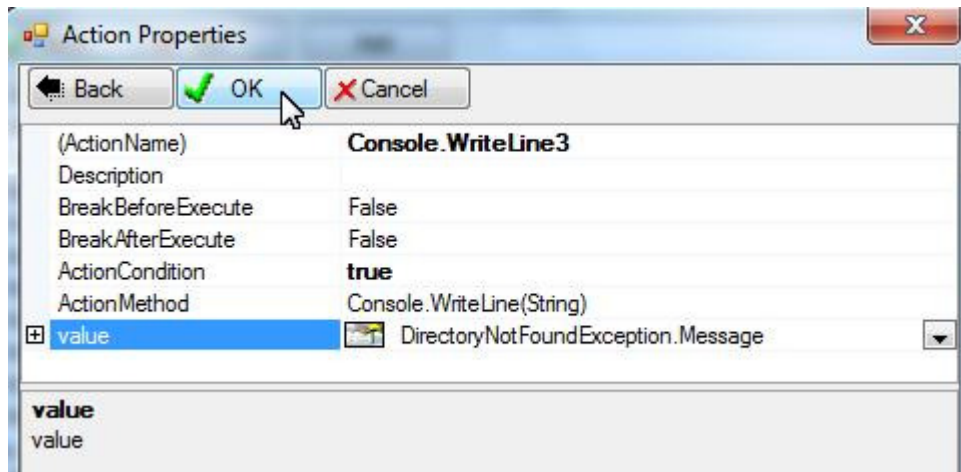


For the “value” of the action, choose “Property”:

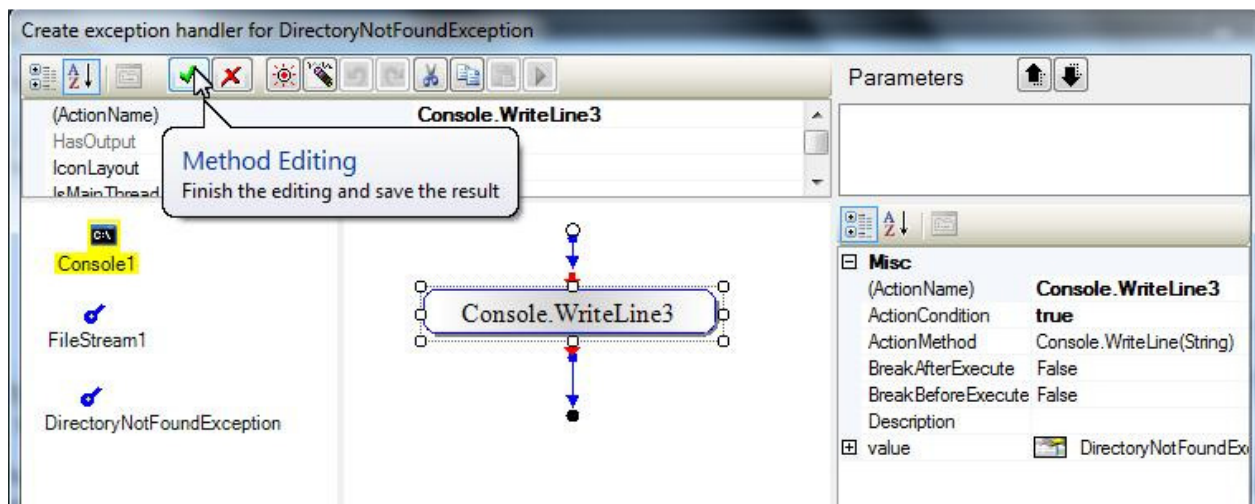


Select the Message property of the Exception object:

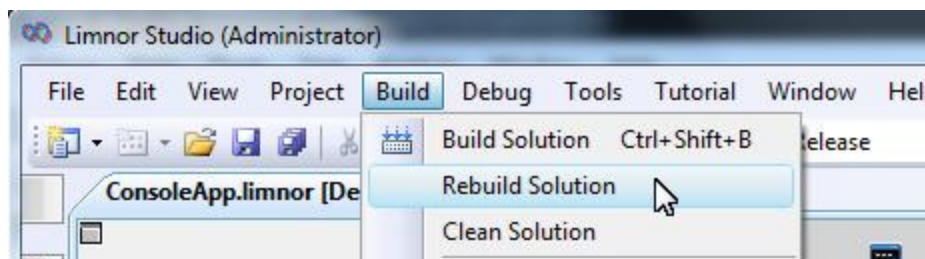




This is our simple exception handling:



Build the sample:



Run this sample. This time it will not crash, instead, it will display error message.

```

Administrator: cmd
C:\Samples\Console1\bin\Release>Console1.exe
Start
Could not find a part of the path 'c:\notExist\file.txt'.
C:\Samples\Console1\bin\Release>
    
```

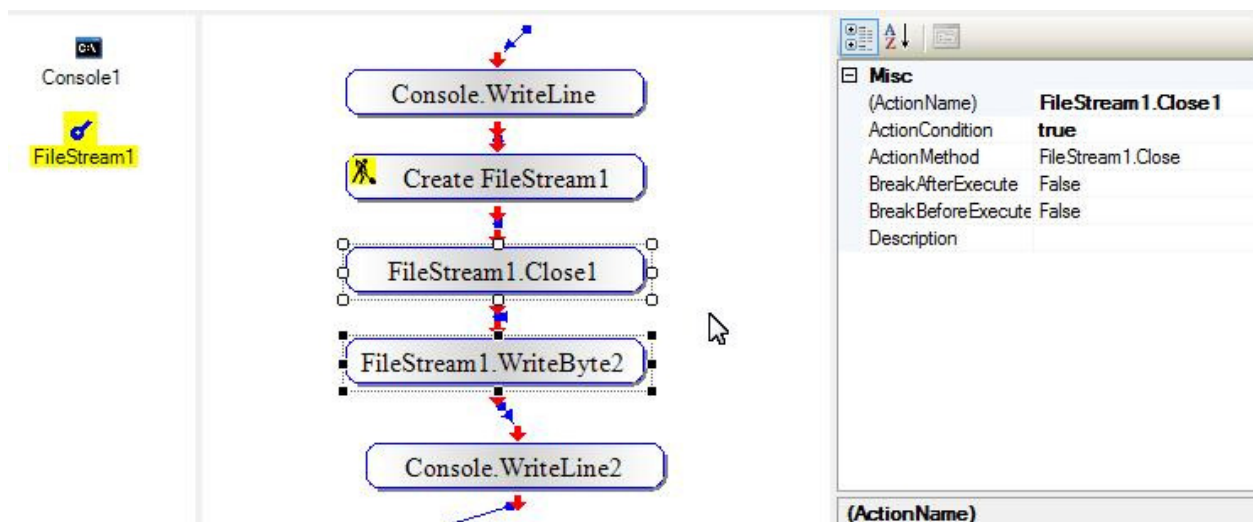
Catch all exceptions

Now let's purposely create a file "c:\notExist\file.txt". Run the sample again. This time no exception occurs. The application runs normally to the end:

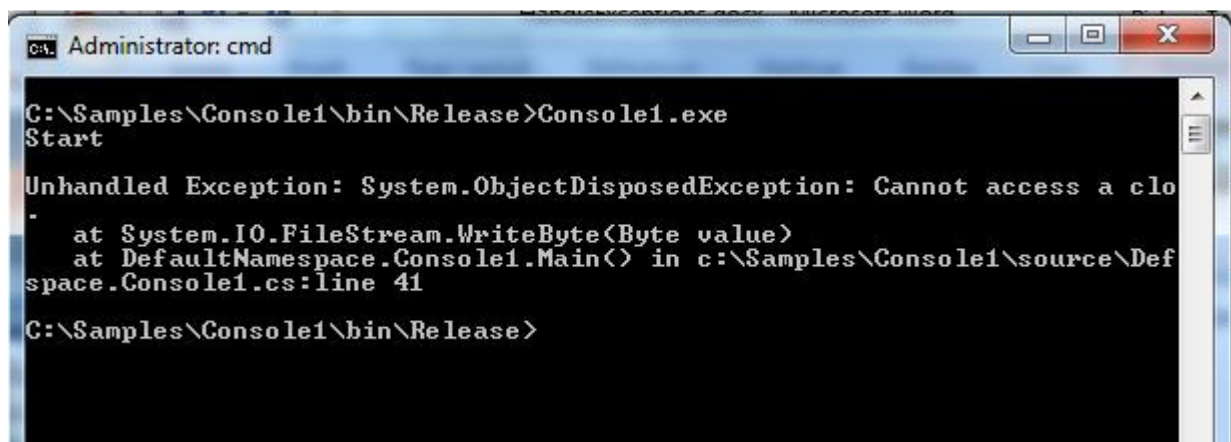
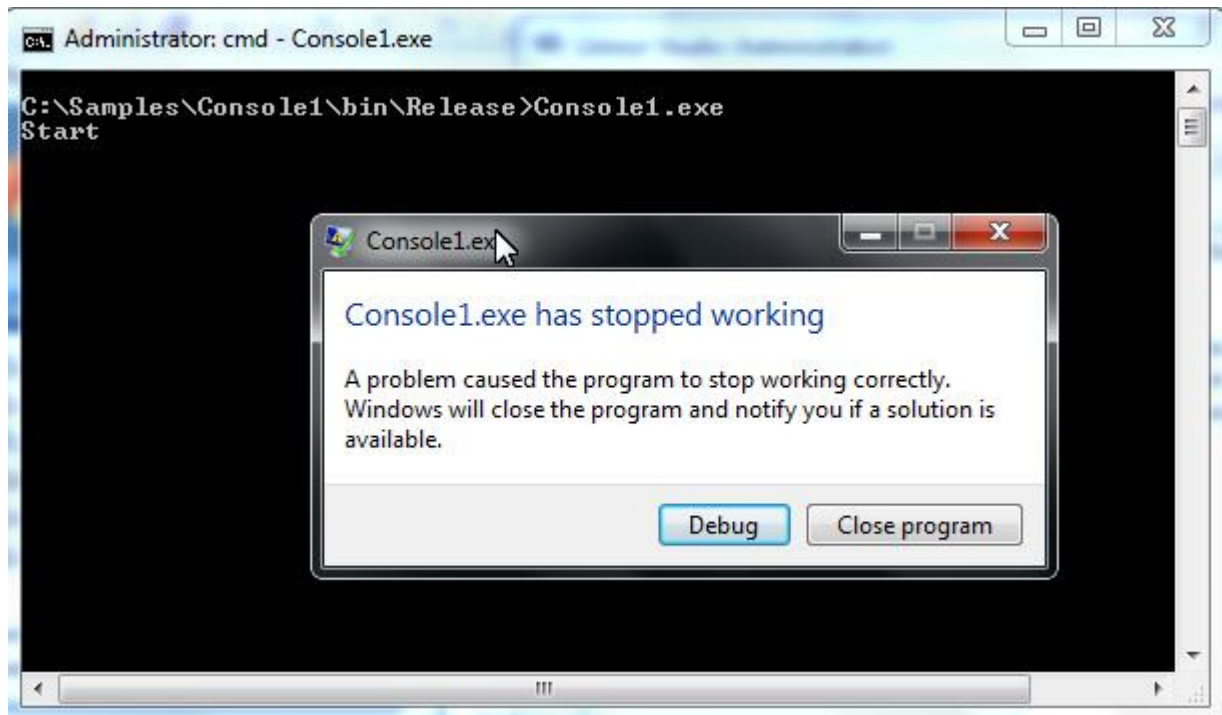
```

Administrator: cmd
C:\Samples\Console1\bin\Release>Console1.exe
Start
End
C:\Samples\Console1\bin\Release>
    
```

Let's insert two new actions: one to close the file the other to write a byte to the file. This is incorrect because we cannot write to a closed file.

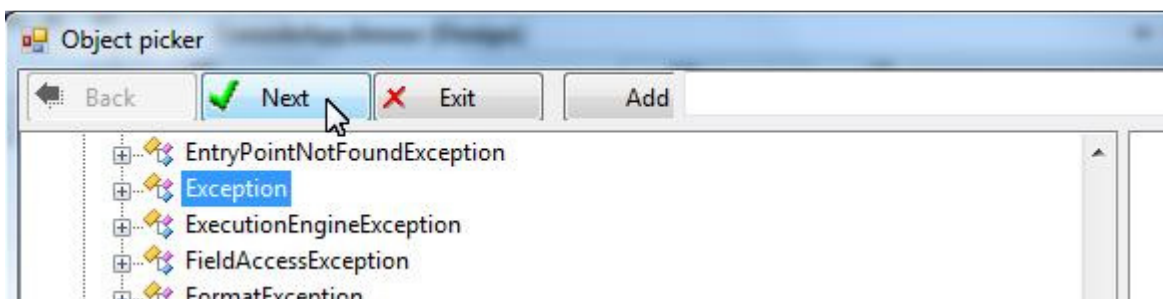
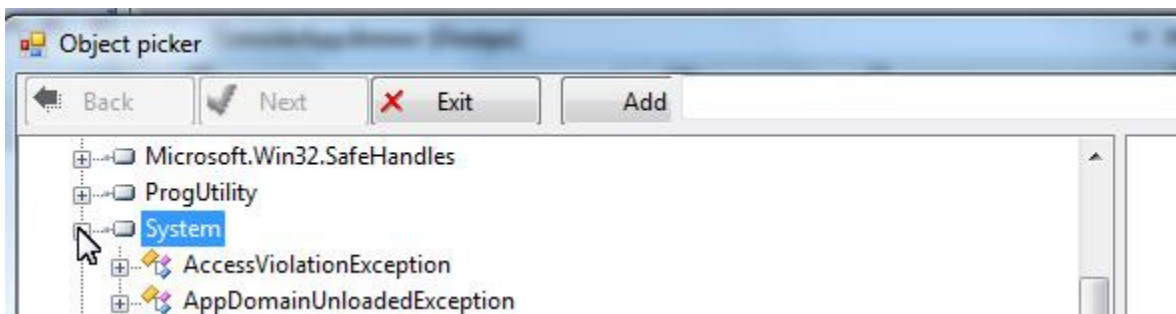
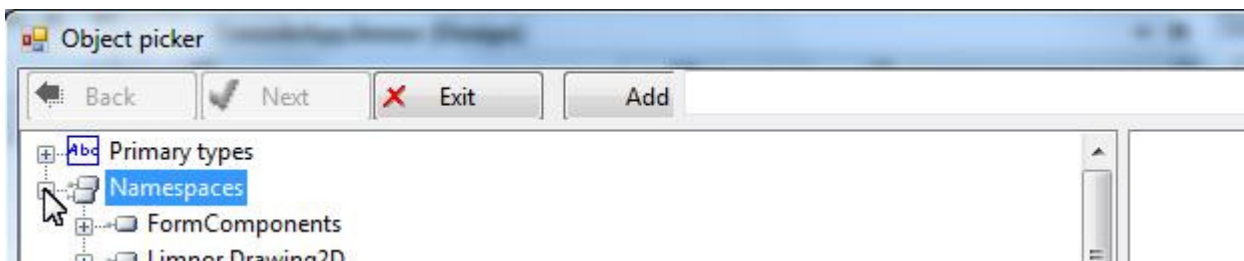
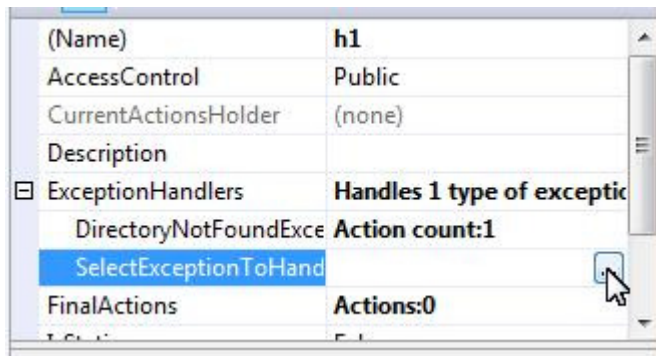


Compile the project. Run it again. We get an exception again:

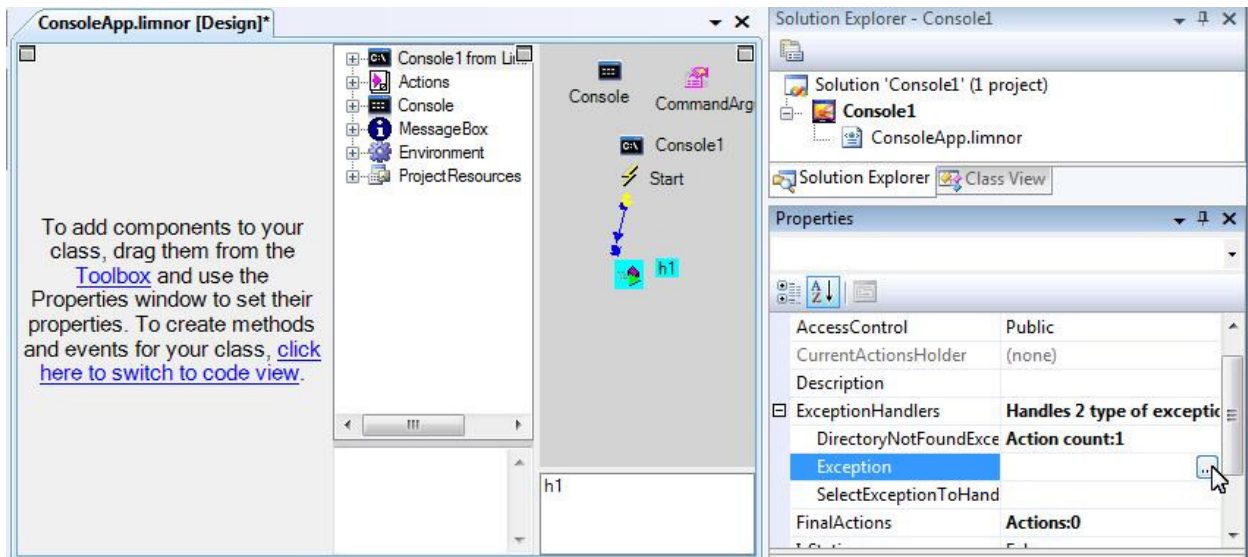


This time we got a `System.ObjectDisposedException`. Because we do not handle this type of exceptions, the program crashes.

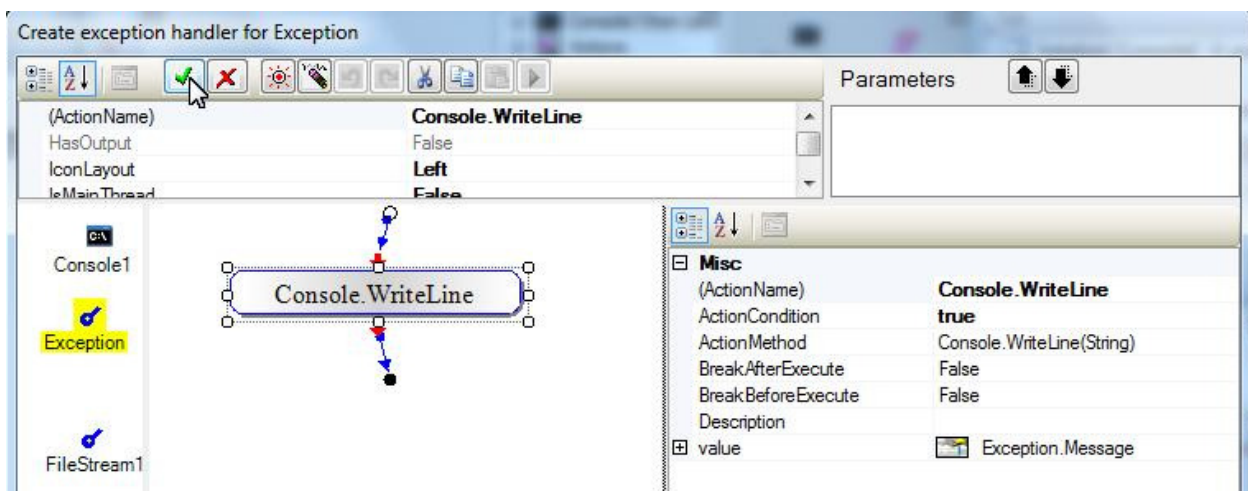
To handle all kinds of exceptions we may handle exceptions of type `System.Exception`. All other types of exceptions are derived from `System.Exception`.



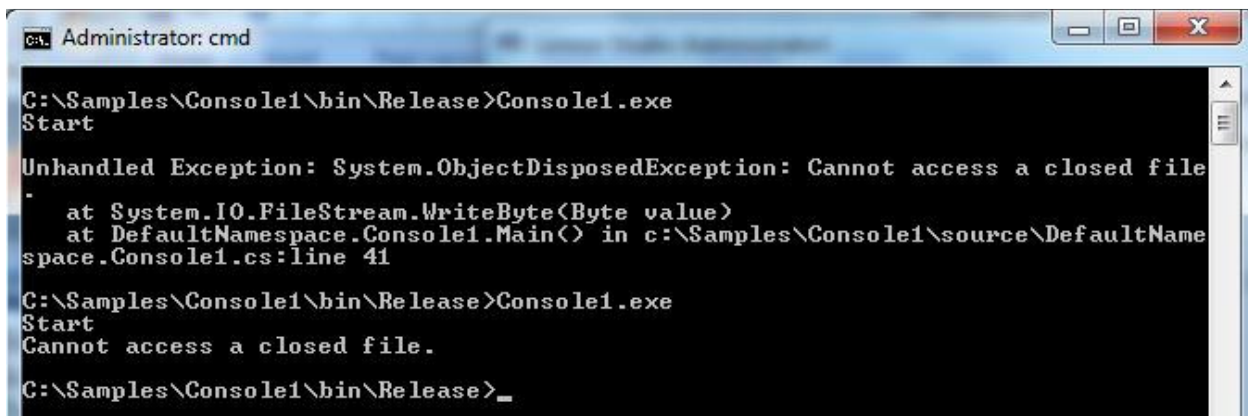
Add actions to handle this generic type of exceptions:



Simply write the exception to the console:



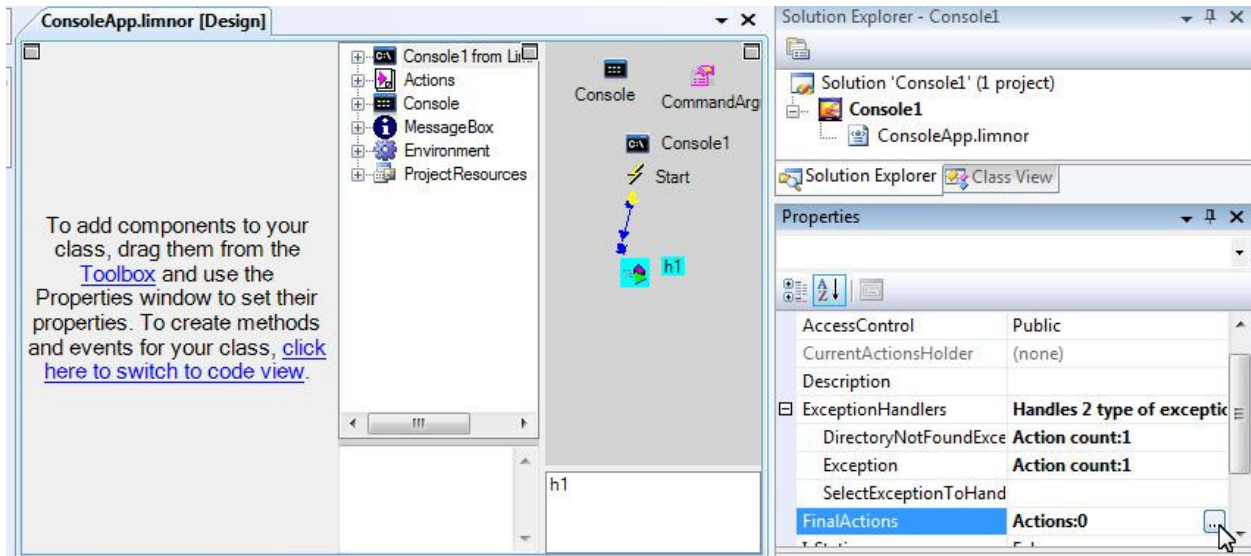
Compile it and run it again. This time it will not crash.



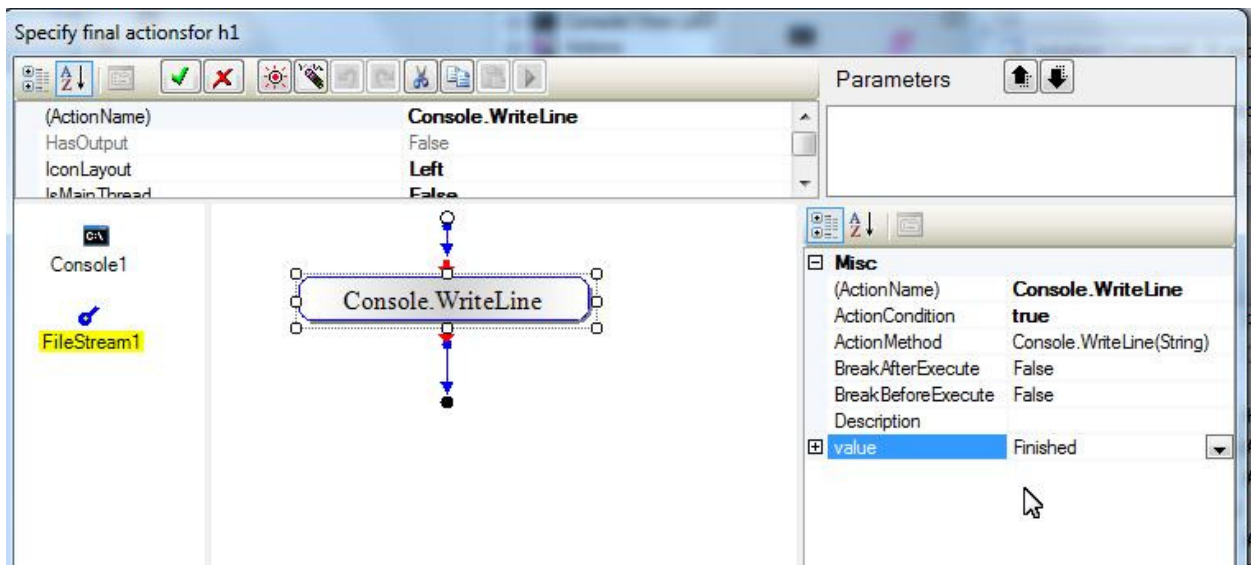
Actions guaranteed to be executed

You may notice that when an exception occur the actions following the action that causes the exception will not get executed. In our sample, the action writing “End” to the console will not be executed.

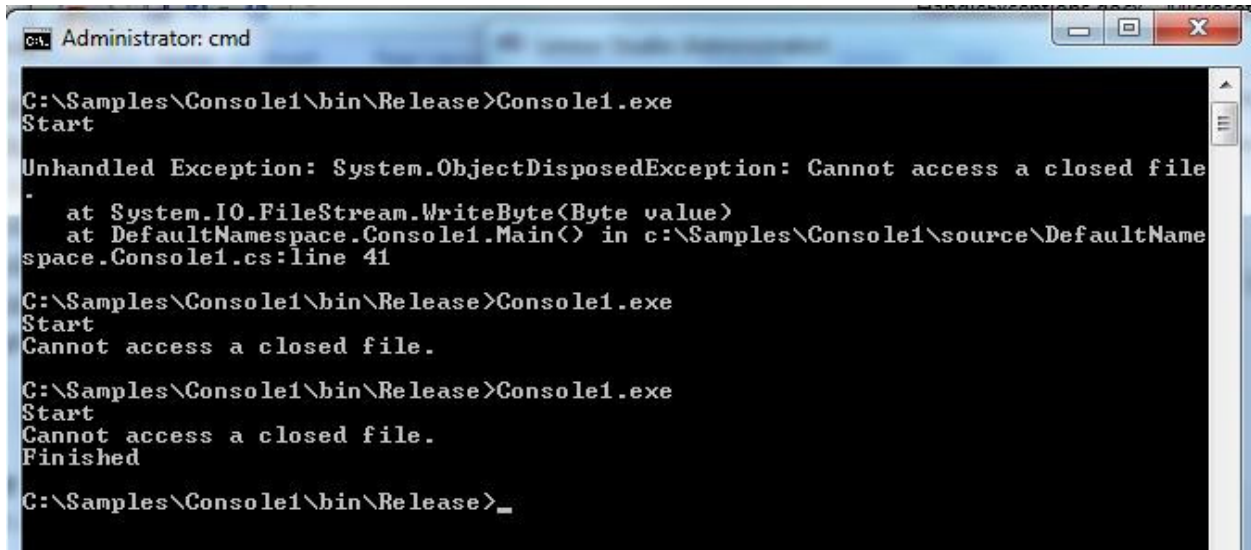
In some cases we want some actions to be executed no matter an exception occurs or not. Such actions can be inserted in the FinalActions property of a method.



For this sample, we add one action which writes a line “Finished” to the console:



Compile and run it. We can see that the line “Finished” appears in the console:



```
Administrator: cmd
C:\Samples\Console1\bin\Release>Console1.exe
Start
Unhandled Exception: System.ObjectDisposedException: Cannot access a closed file
-
   at System.IO.FileStream.WriteByte(Byte value)
   at DefaultNamespace.Console1.Main() in c:\Samples\Console1\source\DefaultName
space.Console1.cs:line 41
C:\Samples\Console1\bin\Release>Console1.exe
Start
Cannot access a closed file.
C:\Samples\Console1\bin\Release>Console1.exe
Start
Cannot access a closed file.
Finished
C:\Samples\Console1\bin\Release>_
```

Note that actions in the `FinalActions` property will be executed no matter how the method is terminated. For example, a method may terminate in one of the following cases:

- An exception occurs.
- All actions finish without exceptions.
- A method-exit action is executed.