

عنوان مضمون

# Visual Programming-I

توسط : صفری

خزان 1397

# Exception handling

# Error

- Errors in code are something that will always be with you. No matter how good a programmer is, problems will always slip through, and part of being a good programmer is realizing this and being prepared to deal with it.

# Type of Error

- syntax errors : *(fatal errors)*

some problems are minor such as a spelling mistake on a button, that prevent compilation.

Ex: When you forgot to type a semicolon (;)

- *semantic errors : (logic errors)*

Your code may compile and run without any Syntax Errors but the output of an operation may produce unwanted or unexpected results in response to user actions. These are generally the hardest type to fix.

EX1: `if(value/ 2==0)` (odd or even)

EX2: your application fails to add a record to a database because a requested field is missing,

EX3: adds a record with the wrong data in other restricted circumstances.

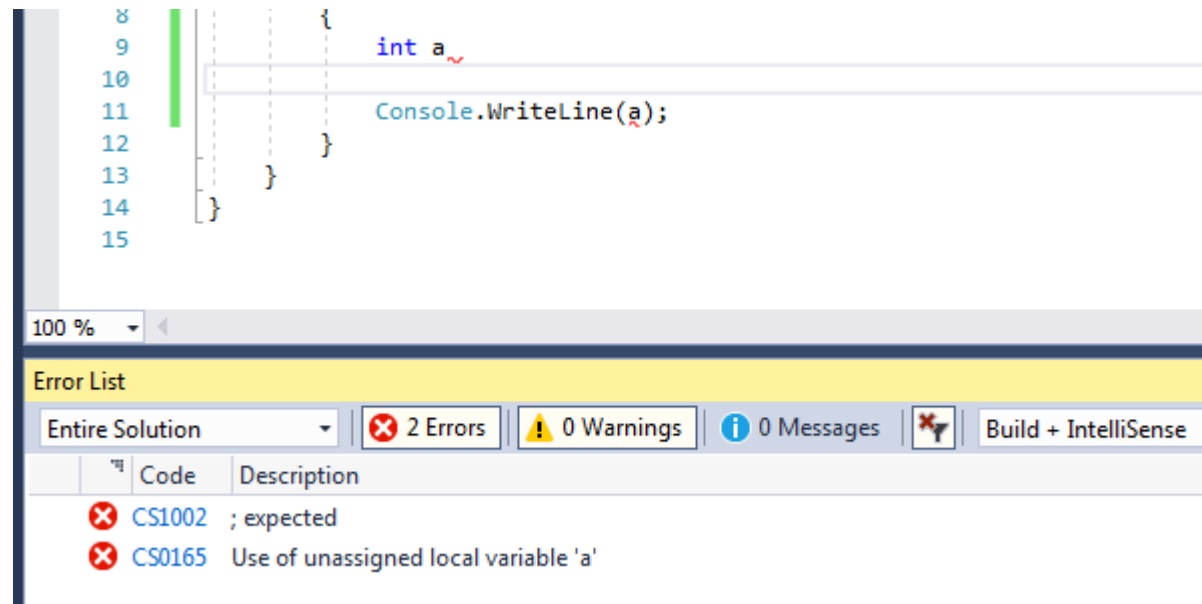
- *Run-time Errors:*

Run-time errors are those that appear only after you compile and run your code. It will occur when your program attempts an operation that is impossible to carry out.

EX: `int a = 5; int b = 0; int c = a /b;`

# Fix syntax errors

- When you compile your application in the development environment, the compiler would point out where the problem is so you can fix it instantly.



# *Fix semantic errors and Run-time Errors*

- **DEBUGGING :**

you can use to identify and fix areas of code that don't work as expected

- Earlier, you learned that you can execute applications in two ways:

- with debugging enabled :

By default, when you execute an application from Visual Studio (VS), it executes with debugging enabled. This happens, for example, when you press F5 or click the green Start arrow in the toolbar.

- without debugging enabled.

To execute an application without debugging enabled, choose Debug ⇨ Start Without Debugging, or press Ctrl+F5.

# with debugging enabled

- When you build an application in debug configuration and execute it in debug mode, more is going on than the execution of your code.
- Debug builds maintain ***symbolic information*** about your application, so that the IDE knows exactly what is happening as each line of code is executed.
- **Symbolic information** means keeping track of, for example, the names of variables used in uncompiled code,
- so they can be matched to the values in the compiled machine code application, **which won't contain such human-readable information.** This information is contained in **.pdb** files, which you may have seen in your computer's Debug directories.

# without debugging enabled

- In the release configuration, application code is optimized, and you cannot perform these operations.
- However, release builds **also run faster**; when you have finished developing an application, you will typically supply users with release builds because they won't require the symbolic information that debug builds include.



# Debugging

The techniques are grouped into two sections according to how they are used.

- Debugging in Nonbreak (Normal) Mode

In general, debugging is performed either by interrupting program execution or by making notes for later analysis.

- Debugging in break mode

that is, normal execution is halted.

# Debugging in Nonbreak (Normal) Mode

1. One of the commands you've been using throughout this book is the `WriteLine()` function, which outputs text to the console. As you are developing applications, this function comes in handy for getting extra feedback about operations:

```
WriteLine("MyFunc() Function is about to be called.");  
MyFunc("Do something.");  
WriteLine("MyFunc() Function execution completed.");
```

Disadvantage:

- This code snippet shows how you can get extra information concerning a function called `MyFunc()`. This is all very well, but it can make your console output a bit **cluttered**;
- when you develop other types of applications, such as desktop applications, you **won't have a console to output information**.

# Debugging in Nonbreak (Normal) Mode

## 2. Outputting Debugging Information:

- Writing text to the Output window at runtime is easy. You simply replace calls to `WriteLine()` with the required call to write text where you want it. There are two commands you can use to do this:
  - `Debug.WriteLine()`  
works in debug builds only
  - `Trace.WriteLine()`  
works for release builds as well
- *Both methods are contained within the **System.Diagnostics** namespace.*

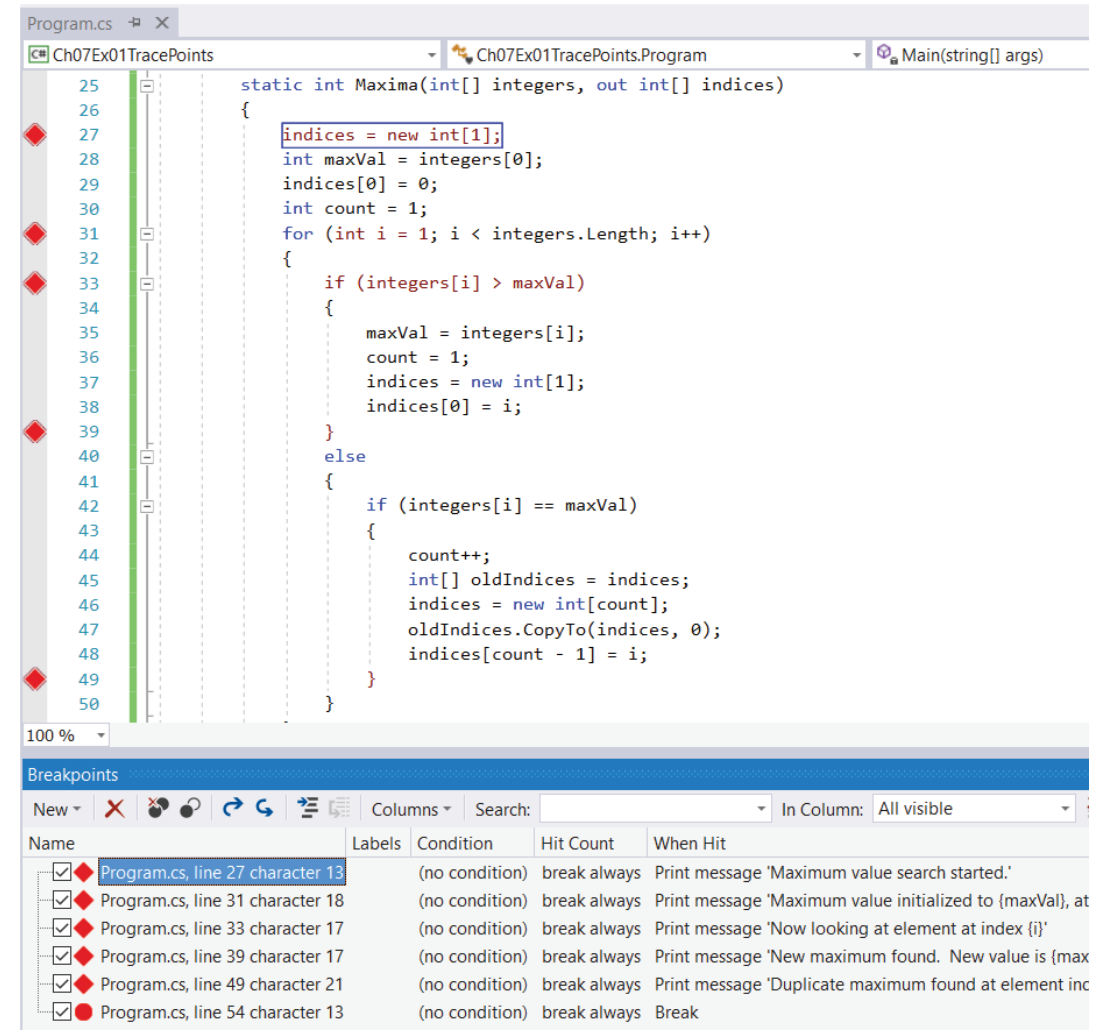
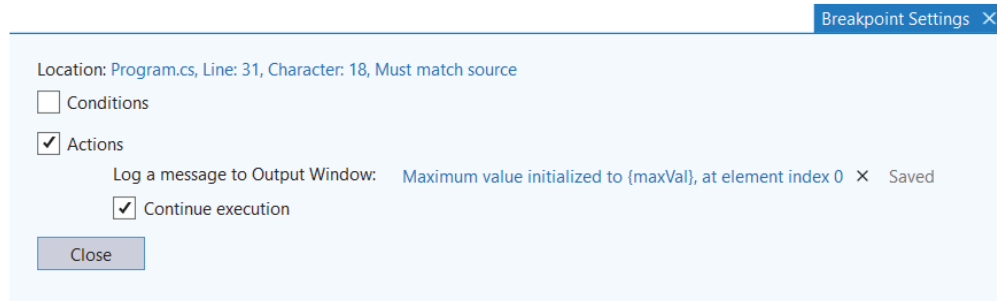
# Debugging in Nonbreak (Normal) Mode

## 3. Tracepoints:

Essentially, they enable you to output debugging information **without modifying your code**.

- The process for adding a tracepoint is as follows:
  1. Position the cursor at the line where you want the tracepoint to be inserted  
The tracepoint will be processed *before* this line of code is executed.
  2. To the left of the line number, click the side bar and a red circle appears. Hover your mouse pointer over the red circle placed next to the line of code and select the Settings menu item.
  3. Check the Actions checkbox and type the string to be output in the Message text box in the Log a message section. If you want to output variable values, enclose the variable name in curly braces.
  4. Click OK. The red circle changes into a red diamond to the left of the line of code containing a tracepoint, and the highlighting of the line of code itself changes from red to white.

# Debugging in Nonbreak (Normal) Mode



# Diagnostics Output Versus Tracepoints

- Now that you have seen two methods of outputting essentially the same information, consider the pros and cons of each.
  1. First, tracepoints have no equivalent to the Trace commands;  
there is no way to output information in a release build using tracepoints.
  2. Deleting a tracepoint is as simple as clicking on the red diamond indicating its position, which can be annoying if you are outputting a complicated string of information.
  3. One bonus of tracepoints, though, is the additional information that can be easily added,  
such as `$FUNCTION` which adds the current function name to the output message. Although this information is available to code written using Debug and Trace commands, it is trickier to obtain.

# Debugging in Break Mode

- This mode can be entered in several ways, all of which result in the program pausing in some way.
  1. Entering Break Mode(Pause button )
  2. Breakpoints
  3. when an assertion is generated
  4. when an unhandled exception is thrown.

# Debugging in Break Mode

- This mode can be entered in several ways, all of which result in the program pausing in some way.

## 1. Entering Break Mode:

- The simplest way to enter break mode is to click the Pause button in the IDE while an application is running.



- Pausing the application is perhaps the simplest way to enter break mode, but it doesn't give you fine grained control over exactly where to stop.  
You might also be able to enter break mode during a lengthy operation, or a long loop, but the exact stop point is likely **to be fairly random**.



# Debugging in Break Mode

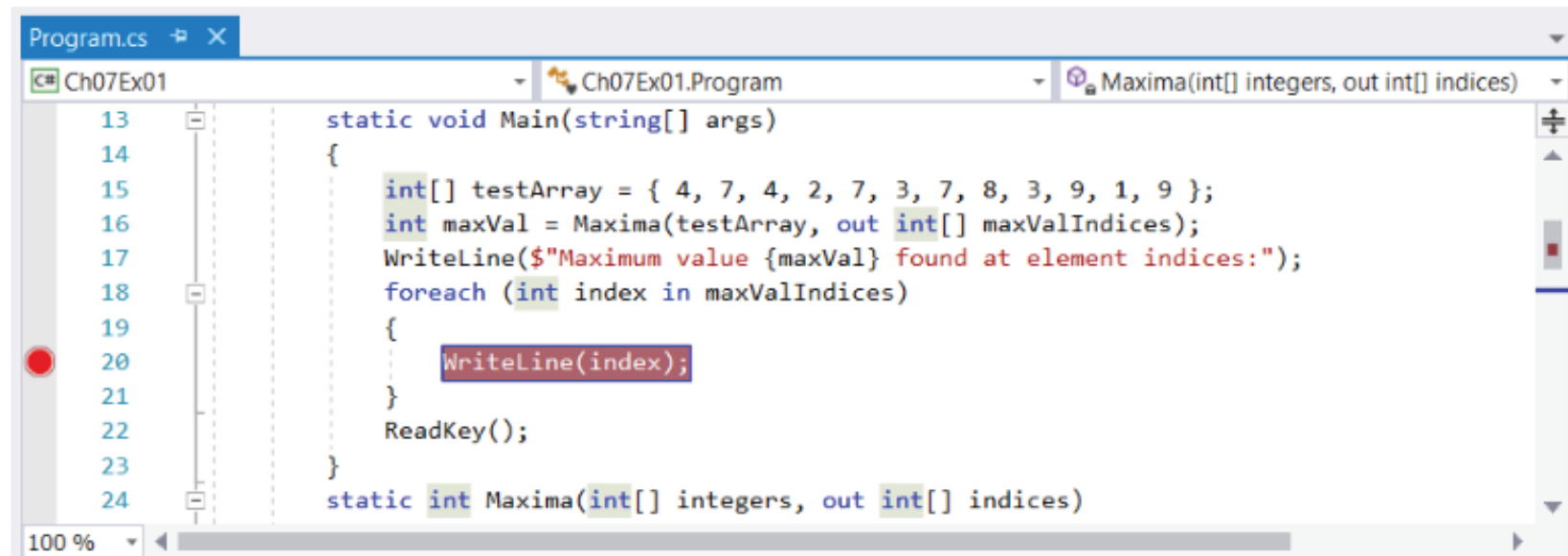
## 2. Breakpoints

A *breakpoint* is a marker in your source code that triggers automatic entry into break mode.

- Breakpoints can be configured to do the following:
  - Enter break mode immediately when the breakpoint is reached.
  - Enter break mode when the breakpoint is reached if a Boolean expression evaluates to true.
  - Enter break mode once the breakpoint is reached a set number of times.
  - Enter break mode once the breakpoint is reached and a variable value has changed since the last time the breakpoint was reached.

# Debugging in Break Mode

- Enter break mode immediately when the breakpoint is reached.
  - ☐ just left-click on the far left of the line of code.
  - ☐ you can select the menu item Debug ⇨ Toggle Breakpoint from the menu, or press F9 and the breakpoint is placed on the line of code which has focus.



# Debugging in Break Mode

- You can see information about a file's breakpoints using the **Breakpoints window**:
  - ✓ You can use the Breakpoints window to disable breakpoints,
  - ✓ to delete breakpoints,
  - ✓ to edit the properties of breakpoints.
  - ✓ You can also add labels to breakpoints, which is a handy way to group selected breakpoints.

Notice that by removing the tick to the left of a description, a disabled breakpoint shows up as an unfilled red circle. You can see labels in the Labels column and filter the items shown in this window by label.

# Debugging in Break Mode

- The other columns shown in this window, **Condition** and **Hit Count**, are only two of the available ones, but they are the most useful. You can edit these by right-clicking a breakpoint and selecting Conditions.... Expanding the drop-down box displays the following options:
  - Conditional Expression
  - Hit Count
  - Filter

# Debugging in Break Mode

- Selecting Conditions...

opens a dialog box in which you can type any Boolean expression, which may involve any variables in scope at the breakpoint.

For example, you could configure a breakpoint that triggers when it is reached and the value of `maxVal` is greater than 4 by entering the expression "`maxVal > 4`" and selecting the `Is true` option. You can also check whether the value of this expression has changed and only trigger the breakpoint then (you might trigger it if `maxVal` changed from 2 to 6 between breakpoint encounters, for example).

# Debugging in Break Mode

- Selecting Hit Count

from the drop-down list opens a dialog box in which you can specify how many times a breakpoint needs to be hit before it is triggered. A drop-down list offers the following options:

- Break always (default value)
  - Break when the hit count is equal to
  - Break when the hit count is a multiple of
  - Break when the hit count is greater than or equal to
- The option you choose, combined with the value entered in the text box next to the options, determines the behavior of the breakpoint.
  - The hit count is useful in long loops, when you might want to break after, say, the first 5,000 cycles. It would be a pain to break and restart 5,000 times if you couldn't do this!

# Debugging in Break Mode



Variable out of bounds.  
Please contact vendor with the error code KCW001.  
at AssertionDemo.Program.Main(String[] args) in  
C:\BeginningCSharp7\Chapter07\AssertionDemo\Program.cs:line 15

Abort

Retry

Ignore

## 3. Assertion :

*Assertions* are instructions that can interrupt application execution with a user-defined message.

They are often used during application development to test whether things are going smoothly. For example, at some point in your application you might require a given variable to have a value less than 10. You can use an assertion to confirm that this is true, interrupting the program if it isn't.

- When the assertion occurs you have the option to :

- ☐ Abort, which terminates the application;

- ☐ Retry, which causes break mode to be entered;

- ☐ Ignore, which causes the application to continue as normal.

# Debugging in Break Mode

- As with the debug output functions shown earlier, there are two versions of the assertion function:

- `Debug.Assert()`

- `Trace.Assert()`

The `Debug` class is only compiled into debug builds, while `Trace` exists in release builds.

- These functions take three parameters:

- 1-The first is a Boolean value, whereby a value of false causes the assertion to trigger.

- 2,3- The second and third are string parameters to write information both to a pop-up dialog box and the Output window.

- The preceding example would need a function call such as the following:

- ```
Debug.Assert(myVar < 10, "myVar is 10 or greater.", "Assertion occurred in Main().");
```



# Monitoring Variable Content

- ❑ The easiest way to check the value of a variable is to hover the mouse over its name in the source code while in break mode.
- You may have noticed that when you run an application, the layout of the various windows in the IDE changes.
- ❑ The new window that appears in the bottom-left corner is particularly useful for debugging. It enables you to keep tabs on the values of variables in your application when in break mode:
  - **Autos**—Variables in use in the current and previous statements (Ctrl+D, A)
  - **Locals**—All variables in scope (Ctrl+D, L)
  - **Watch N**—Customizable variable and expression display (where N is 1 to 4, found on Debug ⇌ Windows ⇌ Watch)
- You can also edit the content of variables from this view.
  - To do this, simply type a new value into the Value column for the variable you want to edit.

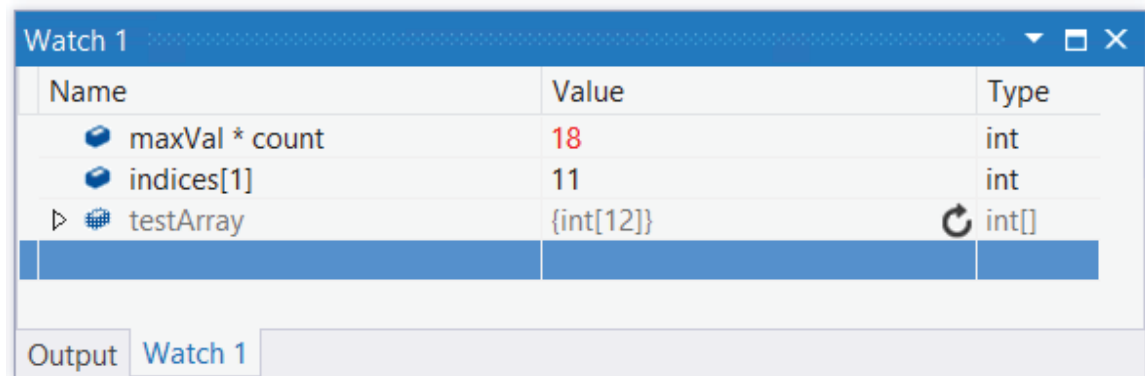
# Monitoring Variable Content





- The **Watch window** enables you to monitor specific variables, or expressions involving specific variables.

To use this window, type the name of a variable or expression into the Name column and view the results.

Note that not all variables in an application are in scope all the time, and are labeled as such in a Watch window.

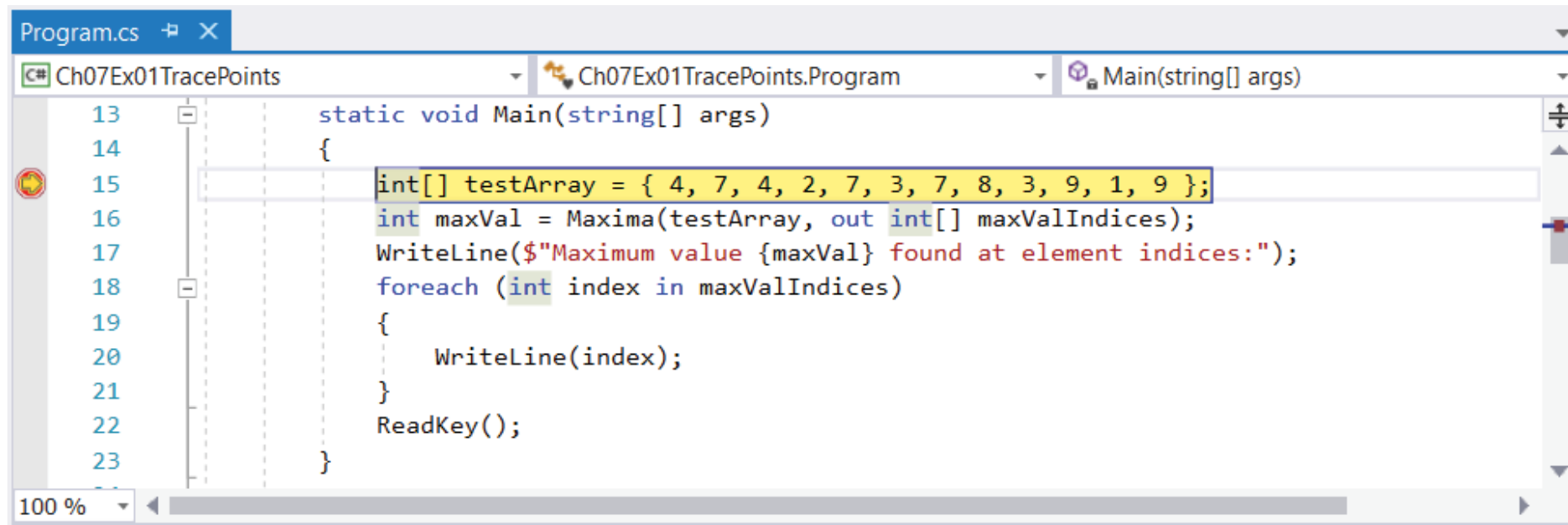
For example, Figure shows a Watch window with a few sample variables and expressions in it, obtained when a breakpoint just before the end of the `Maxima()` function is reached.



| Name                                                                                                 | Value     | Type                                                                                        |
|------------------------------------------------------------------------------------------------------|-----------|---------------------------------------------------------------------------------------------|
|  maxVal * count | 18        | int                                                                                         |
|  indices[1]     | 11        | int                                                                                         |
|  testArray      | {int[12]} |  int[] |

# Stepping through Code

- When Visual Studio enters break mode, a yellow arrow cursor appears to the left of the code view next to the line of code that is about to be executed, as shown in Figure



# Stepping through Code

- The sixth, seventh, and eighth icons control program flow in break mode. In order, they are as follows:



- **Step Into**—Execute and move to the next statement to execute.(F11)
- **Step Over**—Similar to Step Into, but won't enter nested blocks of code, including functions.(F10)
- **Step Out**—Run to the end of the code block and resume break mode at the statement that follows.(shift+F11)

# Stepping through Code

- In code that has semantic errors, these techniques may be the most useful ones at your disposal.
- You can step through code right up to the point where you expect problems to occur, and the errors will be generated as if you were running the program normally

# ERROR HANDLING

# ERROR HANDLING

- Sometimes, however, **you know that errors are likely to occur** and there is **no way to be 100 percent sure that they won't**.
- In those situations, it may be preferable to anticipate problems and **write code that is robust enough to deal** with these errors gracefully, without interrupting execution.
- ***Error handling*** is the term for all techniques of this nature, and this section looks at exceptions and how you can deal with them.
- An **exception** is an error generated either in your code or in a function called by your code that **occurs at runtime**.

```
int[] myArray = { 1, 2, 3, 4 };
```

```
int myElem = myArray[4];
```

- This outputs the following exception message and then terminates the application:

Index was outside the bounds of the array.

- Exceptions are defined in namespaces, and most have names that make their purpose clear. In this example, the exception generated is called `System.IndexOutOfRangeException`,



# try...catch...finally

- The C# language includes syntax for *structured exception handling* (SEH).
- Three keywords mark code as being able to handle exceptions, along with instructions specifying what to do when an exception occurs: **try**, **catch**, and **finally**
- Each of these has an associated code block and must be used in consecutive lines of code. The basic structure is as follows:

```
try
{
    ...
}
catch (<exceptionType> e)
{
    ...
}
finally
{
    ...
}
```

# try...catch...finally

- It is also possible, however, **to have a try block and a finally block with no catch block, or a try block with multiple catch blocks.**
- If one or more catch blocks exist, then the finally block is optional; otherwise, it is mandatory.

➤ try—Contains code that might throw exceptions (“throw” is the C# way of saying “generate” or “cause” when talking about exceptions).

➤ catch—Contains code to execute when exceptions are thrown.

catch blocks can respond only to specific exception types (such as `System.IndexOutOfRangeException`) using `<exceptionType>`, hence the ability to provide multiple catch blocks. It is also possible to omit this parameter entirely, to get a general catch block that responds to all exceptions.

# try...catch...finally

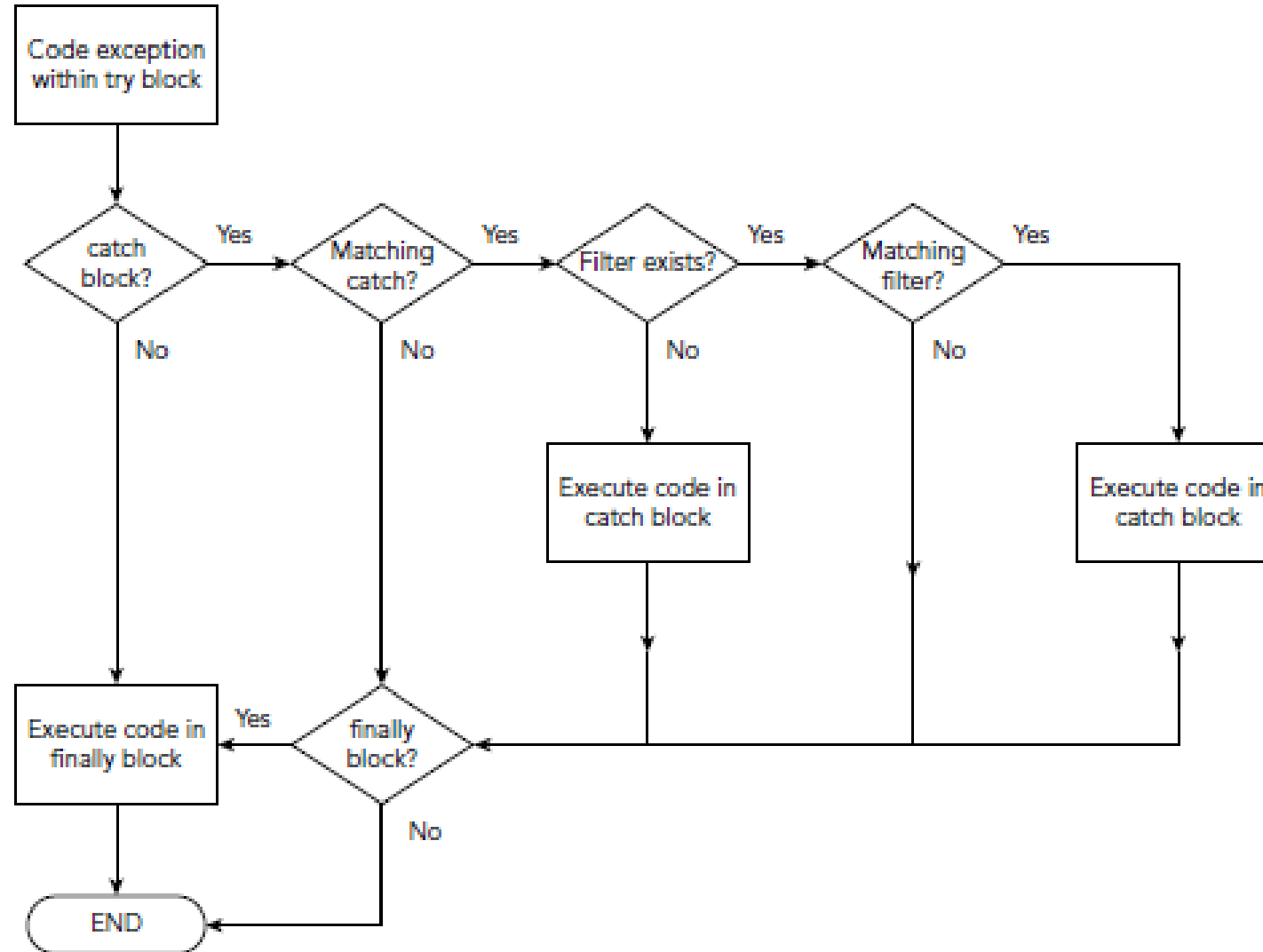
- ➤ finally — Contains code that is always executed,
  - either after the try block if no exception occurs,
  - after a catch block if an exception is handled,
  - or just before an unhandled exception moves “up the call stack.” This phrase means that SEH allows you to nest try...catch...finally blocks inside one another, either directly or because of a call to a function within a try block.

For example, if an exception isn't handled by any catch blocks in the called function, it might be handled by a catch block in the calling code. Eventually, if no catch blocks are matched, then the application will terminate.

# try...catch...finally

- Here's the sequence of events that occurs after an exception occurs in code in a try block:
  - The try block terminates at the point where the exception occurred.
  - If a catch block exists, then a check is made to determine whether the block matches the type of exception that was thrown. If no catch block exists, then the finally block (which must be present if there are no catch blocks) executes.
  - If a catch block exists but there is no match, then a check is made for other catch blocks.
  - If a catch block matches the exception type, then the code it contains executes, and then the finally block executes if it is present.
  - If no catch blocks match the exception type, then the finally block of code executes if it is present

the sequence of events that occurs after an exception occurs in code in a try block



```
class Program
```

```
{    int result;

    Program ()
    {        result = 0;
    }

    public void division(int num1, int num2)
    {        try {
                result = num1 / num2;
            }
            catch
            {    Console.WriteLine("Something went wrong!");
            }
            finally
            {        Console.WriteLine("Result: {0}", result);
            }

    }

    static void Main(string[] args)
    {        Program d = new Program ();
            d.division(25, 0);
            Console.ReadKey();

    }

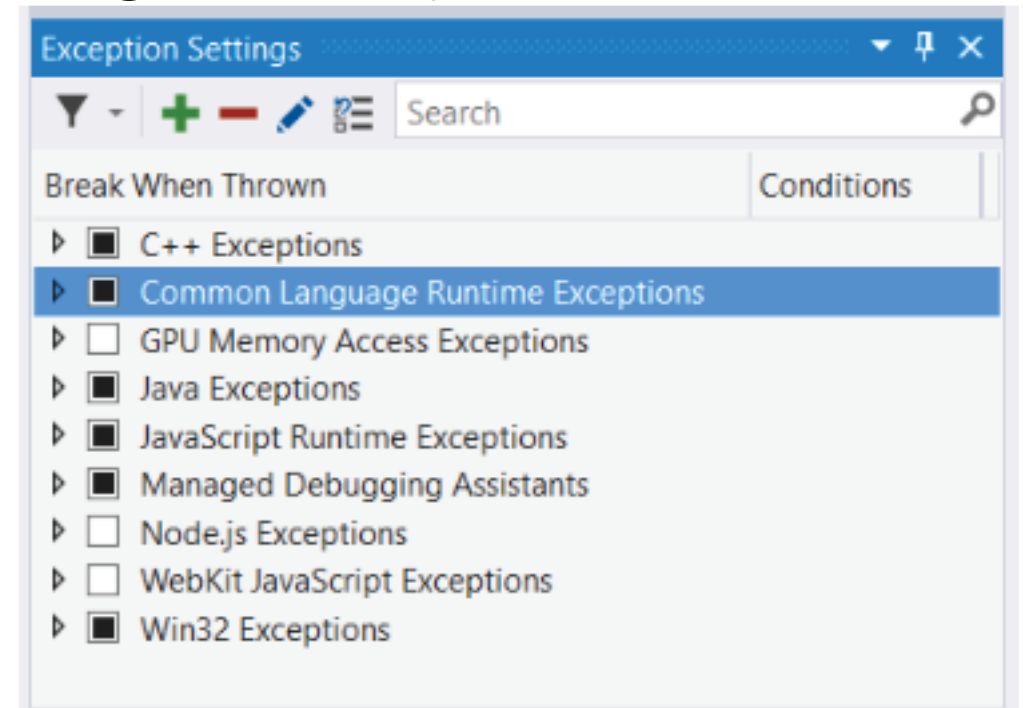
}
```

```
class program
```

```
{    int result;  
    program()  
    {        result = 0;  
    }  
    public void division(int num1, int num2)  
    {        try  
            {            result = num1 / num2;  
            }  
            catch(Exception ex)  
            {            Console.WriteLine("An error occurred: " + ex.Message);  
            }  
            finally  
            {            Console.WriteLine("Result: {0}", result);  
            }  
    }  
    static void Main(string[] args)  
    {        program d = new program();  
            d.division(25, 0);  
            Console.ReadKey();  
    }  
}
```

# Listing and Configuring Exceptions

- The .NET Framework contains a host of exception types, and you are free to throw and handle any of these in your own code.
- The IDE supplies a dialog box for examining and editing the available exceptions, which can be called up with the Debug ⇨ Windows ⇨ Exception Settings menu item (or by pressing Ctrl+D, E).





```
class program
```

```
{    int result;
    program()
    {    result = 0;
    }
    public void division(int num1, int num2)
    {    try
        {    result = num1 / num2;
        }
        catch (DivideByZeroException e)
        {    Console.WriteLine("Exception caught: {0}", e);
        }
        finally
        {    Console.WriteLine("Result: {0}", result);
        }
    }
    static void Main(string[] args)
    {    program d = new program();
        d.division(25, 0);
        Console.ReadKey();
    }
}
```