Workflow debugging

Effective Debugging

This lab focuses on the tools to diagnose common problems.

Setup: download https://github.com/chrisnas/EffectiveDebugging/tree/master/SourceCode and open the .sln file. Ensure that the Debug configurations is set.

Presentation of the application

Tetris is a quick implementation of a well-known game. When starting a game, initialization is done in the `ButtonStart_Click` method. Then, `_timer` is used to call `UpdateGame` every 16 ms to update the position of the blocks. The rendering is done in `GameArea_Paint`.

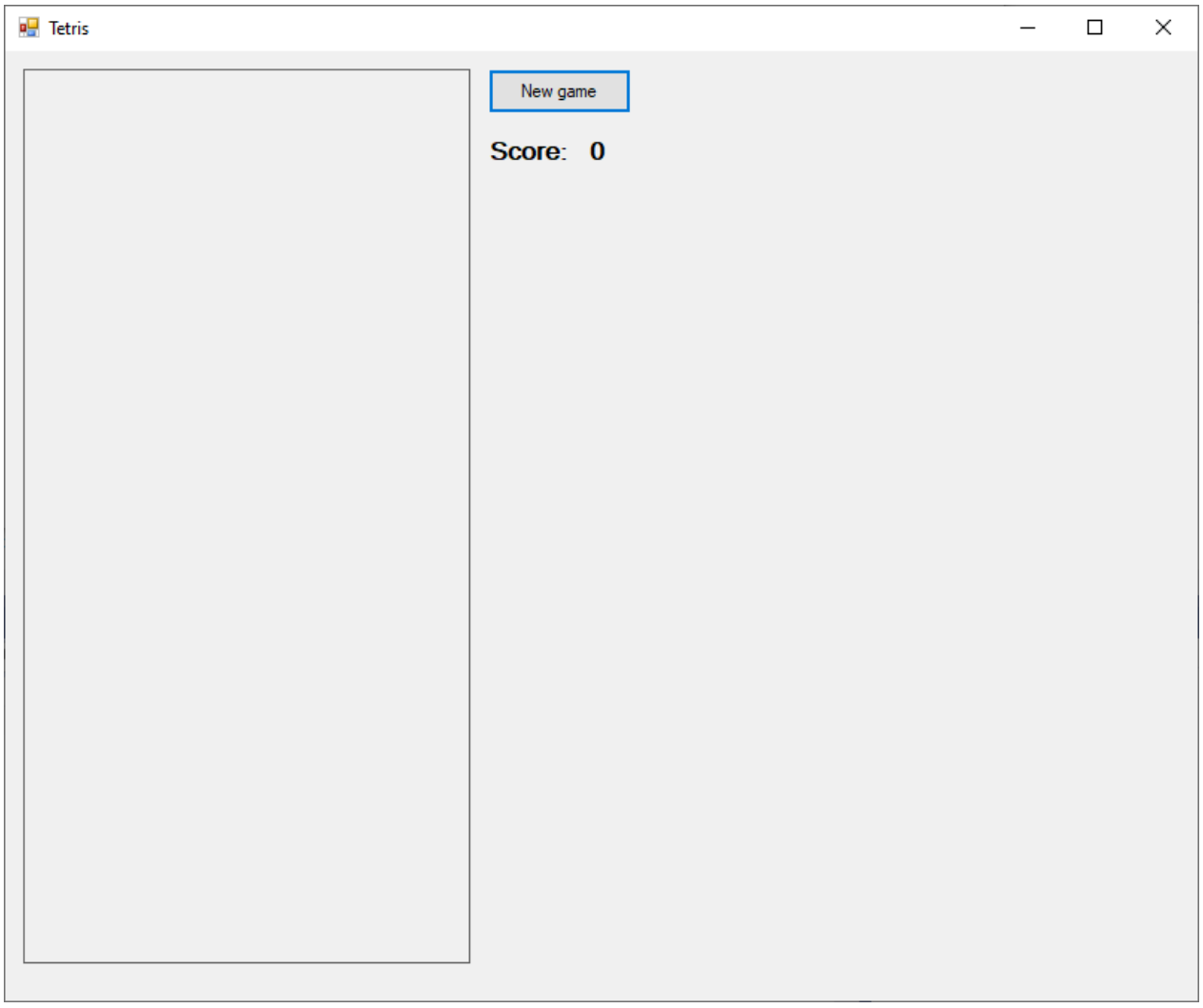
In `UpdateGame`, the following logic is executed:

- Check if a key is pressed and move blocks accordingly (in `MoveBlock` and `RotateBlock`). Moving and rotating can be respectively done once every 100 ms and once every 300 ms.
- Move the block down every 500 ms
- When moving the block down, check if it collides with something. If it does, then clear the lines if any, and spawn a new random block
- If anything has changed, invalidate the game area to force a refresh

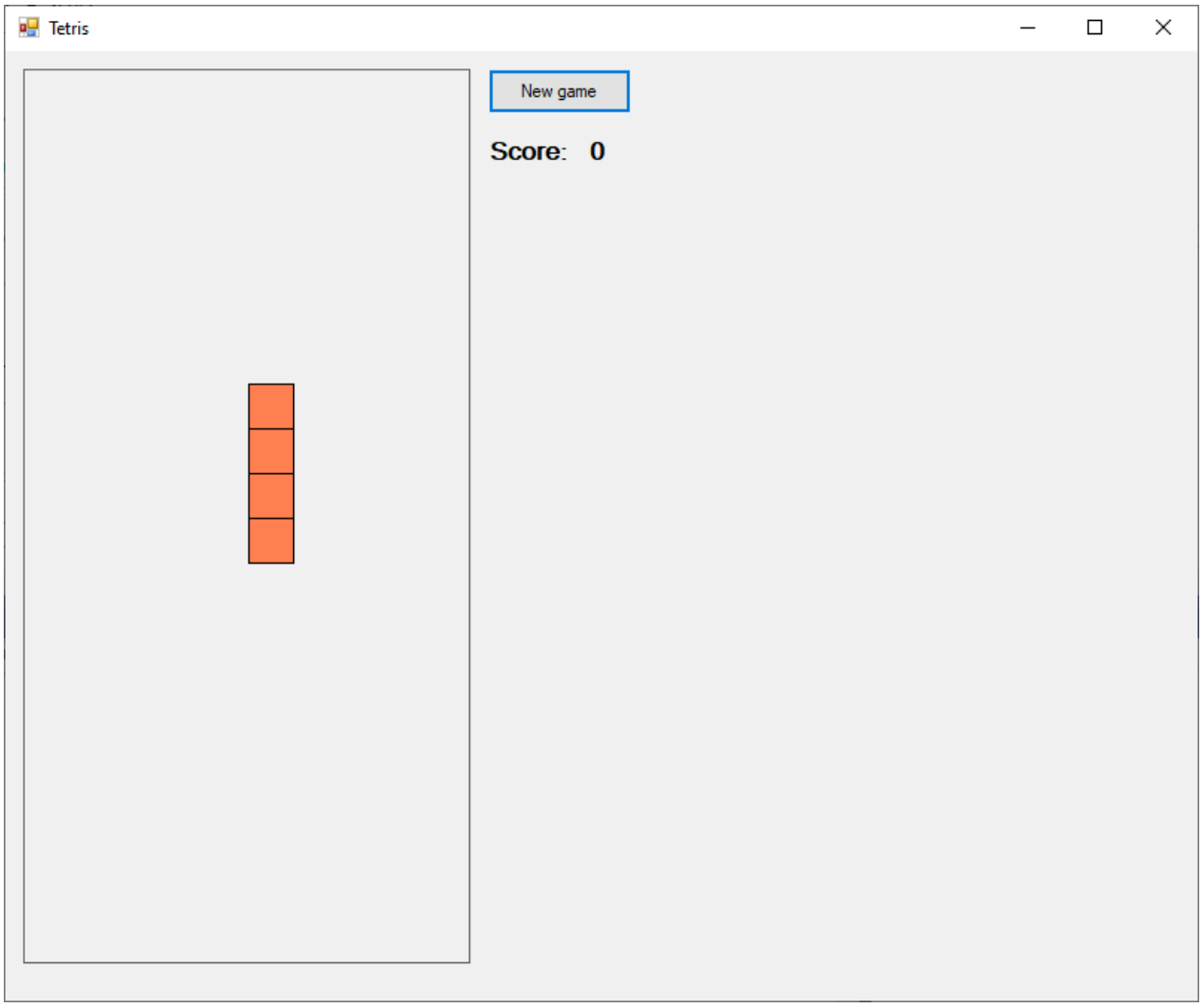
Using a 500 lines of code program for a debugging problem also aims to demonstrate how you don't need to fully understand the mechanics of an application to diagnose simple problems. Focus on the most suspicious piece of code, then work your way iteratively from there.

First problem – moving blocks

Start by debugging the Tetris application. Click on the "New game" button to start a new game.



While the first block moves down, try pressing left or right: nothing happens.



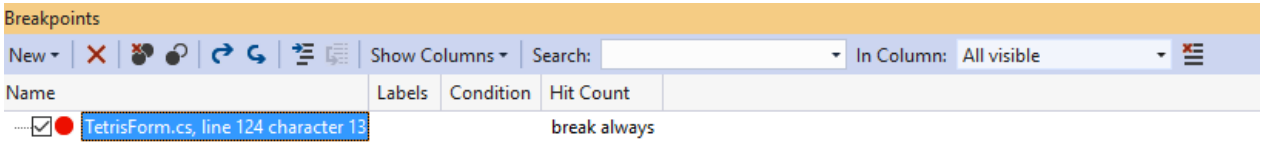
The logic to move blocks happens in the `MoveBlock` method (refer to the technical description in the introduction). Stop the application, then take a minute to locate the method and read the code to see if you spot something suspicious.

The code makes sure that the block is moved at most once every 100ms. Then it optimistically moves the block, and moves it back if the new coordinates intersect with something (another block or the border). At a glance, there are no obvious mistakes.

Let's try to set a breakpoint on the first line of the method: `if (offsetX == 0 && offsetY == 0)` (either by clicking on the margin next to the line, or by putting the cursor on the line and pressing **F9**)

```
1 reference
122     private void MoveBlock(int offsetX, int offsetY)
123     {
124         if (offsetX == 0 && offsetY == 0)
125         {
126             return;
127         }
128
```

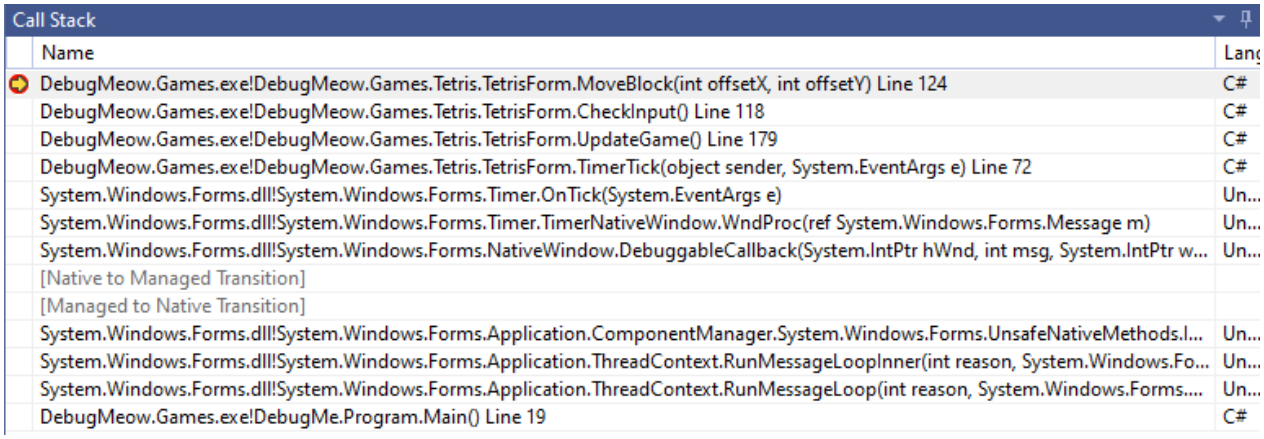
The line is highlighted in red, and a red dot in the margin confirms that the breakpoint is set. If you open the **Breakpoints** panel (**Debug | Windows | Breakpoints**), you will see all your breakpoints listed:



Now start the application again and begin a new game. The debugger immediately breaks the execution.

```
1 reference
122     private void MoveBlock(int offsetX, int offsetY)   offsetX = 0,  offsetY = 0
123     {
124         if (offsetX == 0 && offsetY == 0)   offsetX = 0,  offsetY = 0
125         {
126             return;
127         }
128
```

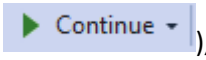
But we didn't actually have time to press a key, so why is `MoveBlock` called? To find out, open the **Call Stack** panel (**Debug | Windows | Call Stack**):



Here you can see that `MoveBlock` has been called by `CheckInput`. Double-click that line to go back to the caller.

```
1 reference
private void CheckInput()
{
    if (_currentBlock != null)
    {
        int offsetX = 0;   offsetX = 0
        int offsetY = 0;   offsetY = 0

        if (IsKeyDown(Keys.Left))
        {
            offsetX = -1;
        }
        else if (IsKeyDown(Keys.Right))
        {
            offsetX = 1;
        }
        else if (IsKeyDown(Keys.Down))
        {
            offsetY = 1;
        }
        else if (IsKeyDown(Keys.Up))
        {
            RotateBlock();
        }

        MoveBlock(offsetX, offsetY);   offsetX = 0,  offsetY = 0
    }
}
```

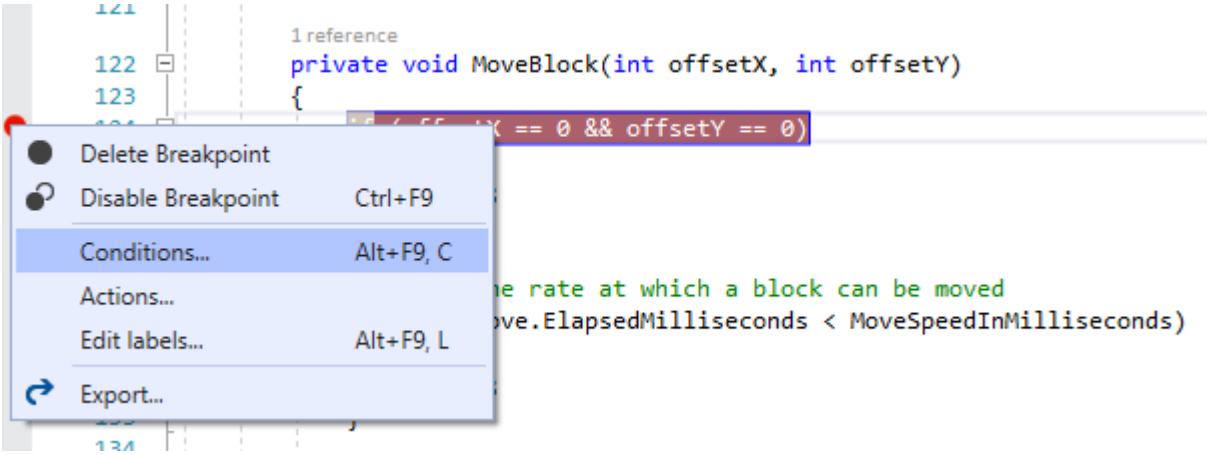
Here, we can see that `MoveBlock` is always called, hence the `if (offsetX == 0 && offsetY == 0)` check at the beginning of the method (to filter the case when the block shouldn't actually be moved). We are only interested in the case when `offsetX` or `offsetY` are different from 0.

If you resume execution (by pressing **F5** of by clicking 

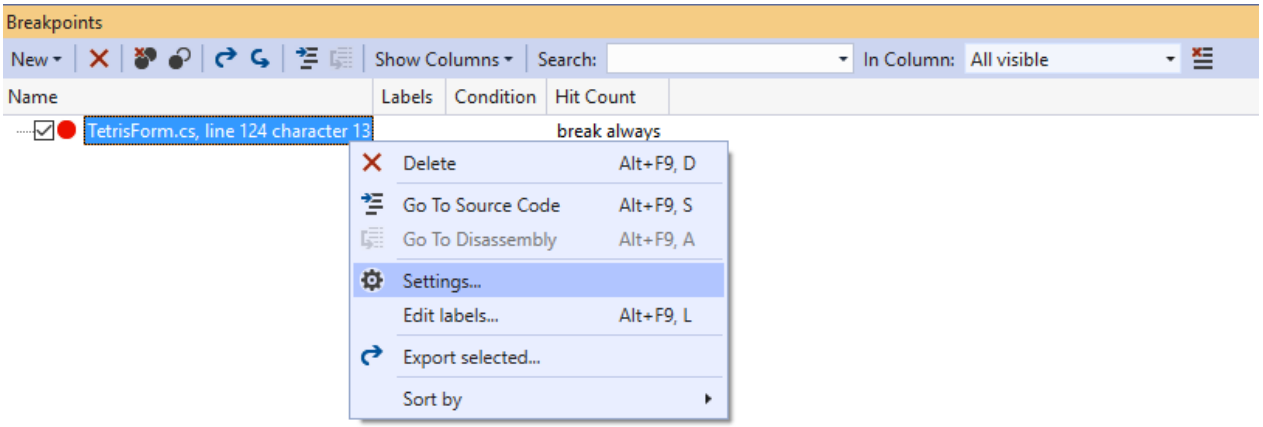
), the execution will immediately stop again on the breakpoint. This makes debugging much harder: since `CheckInput` is called at a very high rate (once every 16ms), there is no way that you could resume the execution and press a key before the breakpoint is hit again.

Of course, in this precise case you could set the breakpoint just after the `if (offsetX == 0 && offsetY == 0)` check, but for the sake of the exercise let's pretend this is not an option. How to activate the breakpoint only in the interesting cases? This is possible by converting the breakpoint into a conditional breakpoint.

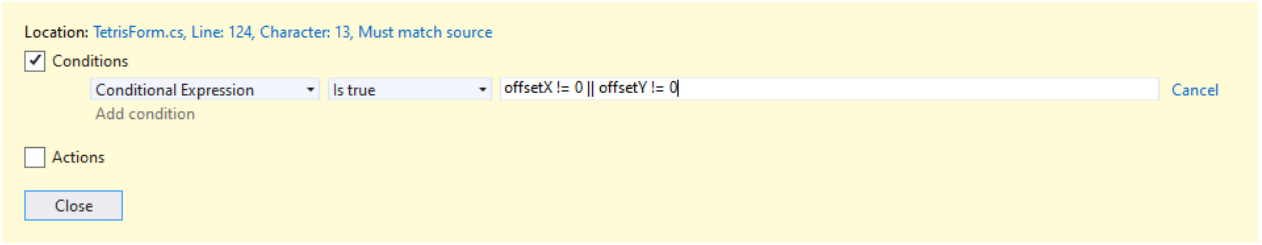
Right click the red dot in the margin and select **Conditions**.



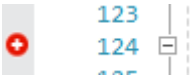
Another way to get there is to right-click the breakpoint in the **Breakpoints** panel and select **Settings**.



Make sure the “Conditions” box is checked, and type offsetX != 0 || offsetY != 0



After validating, you’ll see that the breakpoint icon in the margin has slightly changed with a white cross, to indicate that a condition is attached to the breakpoint:



Now start the application again and begin a new game. While the block goes down, the breakpoint isn't activated. Then as soon as you press the left or right arrow key, the breakpoint will be hit. This is a very powerful way to debug code based on conditions that are only hit occasionally.

Step through the code using **F10** or the step over button 

. You'll see that the first condition is ignored, but then the method is about to return after the second one:

```
// Limit the rate at which a block can be moved
if (_lastMove.ElapsedMilliseconds < MoveSpeedInMilliseconds)
{
    return false;  ≤1ms elapsed
}
```

What is the value of `_lastMove.ElapsedMilliseconds`? There are a few ways to find out. The most straightforward one is to hover the mouse on the property:

```
// Limit the rate at which a block can be moved
if (_lastMove.ElapsedMilliseconds < MoveSpeedInMilliseconds)
{
    return false;  ≤1ms elapsed
}
```

_lastMove.ElapsedMilliseconds 0

You can also inspect the **Autos** panel (**Debug | Windows | Autos**), which tries to guess what variables or properties you might be interested in, depending on the context:

Autos

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
MoveSpeedInMilliseconds	100	int
_lastMove	{System.Diagnostics.Stopwatch}	System.Diagnostics.S...
_lastMove.ElapsedMilliseconds	0	long
this	{DebugMeow.Games.Tetris.TetrisForm, Text: Tetris}	DebugMeow.Games....

You can also type the name of the property in one of the **Watch** panels (**Debug | Windows | Watch | Watch 1 through 4**):

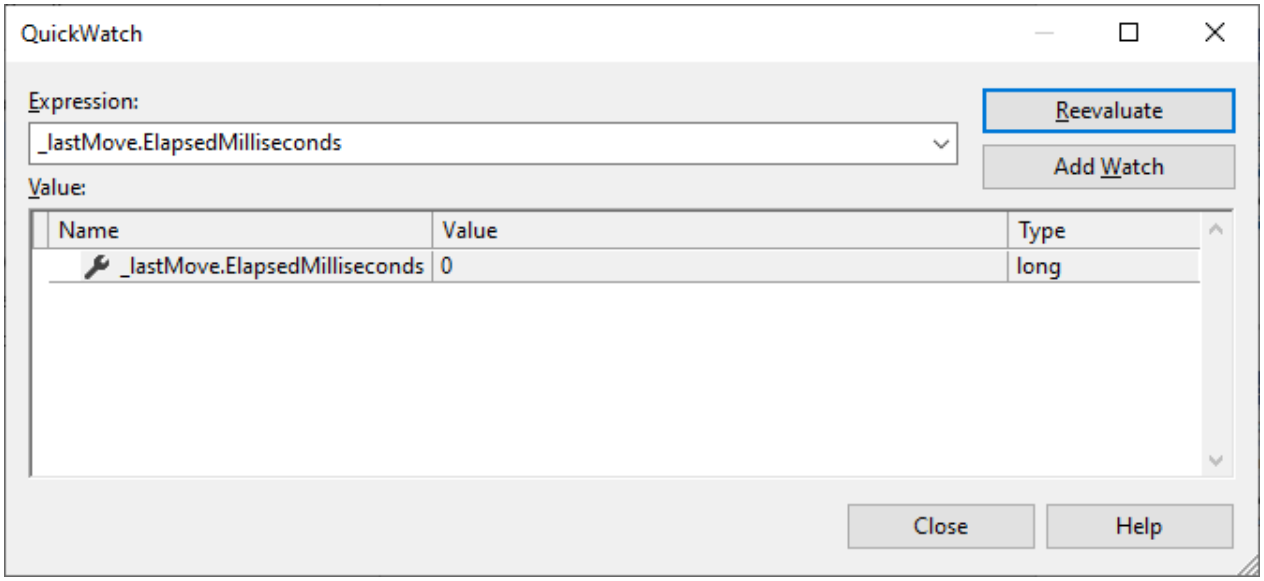
Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
_lastMove.ElapsedMilliseconds	0	long
Add item to watch		

Last but not least, you can select the text in the editor, right-click and select **Quickwatch**

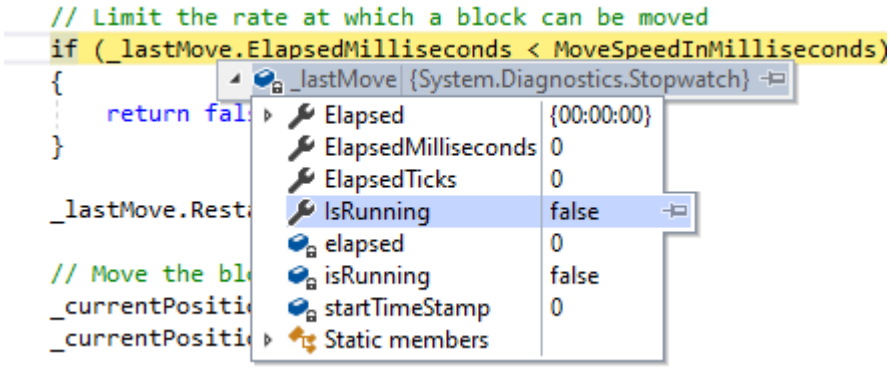
(or directly type **shift + F9**):



`_lastMove.ElapsedMilliseconds` is 0, which is lower than `MoveSpeedInMilliseconds` (100). The rate limiter is triggered, and the method exits. It could be a one-time thing, so press **F5** to resume the execution, and press left or right to break again.

`_lastMove.ElapsedMilliseconds` is still 0, so there's something wrong going on.

If you inspect the property of the object (by using any of the tools above), you will see that the `IsRunning` property is false.



The stopwatch is supposed to be started when starting a new game, in `ButtonStart_Click`. If you inspect the method, you'll see that `_lastMove.Reset();` is called instead of `_lastMove.Restart();`

```
1 reference
private void ButtonStart_Click(object sender, EventArgs e)
{
    ClearGrid();
    UpdateScore(0);
    _lastMove.Reset();
    _lastFrame.Restart();
    _lastUpdate.Restart();
    _lastRotate.Restart();
    _currentBlock = null;
    UpdateGame();
    _timer.Start();
}
```

Change that line by `_lastMove.Restart();` and you'll be able to move blocks properly.

Second problem – collisions

There's another bug that you may already have noticed while debugging the first one. If you start a game and wait until the block reaches the bottom of the game area, it won't stop and will disappear. Let's try to figure out what's happening.

This logic is in the `UpdateGame` method.

```
2 references
private void UpdateGame()
{
    // Check input
    CheckInput();

    // Move the block down
    if (_lastUpdate.ElapsedMilliseconds > UpdateSpeedInMilliseconds)
    {
        if (_currentBlock == null)
        {
            _currentBlock = _blocks[_rnd.Next(0, _blocks.Length)];
            _currentPosition = _startPosition - (0, _currentBlock.MaxHeight);
        }
        else
        {
            _currentPosition.Y++;

            // Check for collisions
            foreach (var coordinate in _currentBlock.GetCoordinates())
            {
                if (IntersectsWithSomething(coordinate))
                {
                    if (!DropBlock(_currentPosition + (0, -1), _currentBlock))
                    {
                        GameOver();
                        return;
                    }

                    _currentBlock = null;
                    break;
                }
            }
        }

        _lastUpdate.Restart();
    }

    GameArea.Invalidate();
    UpdateStats();
}
```

If the block hasn't moved down for more than `UpdateSpeedInMilliseconds`, the code increases the Y coordinate of the block (`_currentPosition`). If after that the block intersects with something (`IntersectsWithSomething`), the `DropBlock` method is called, which effectively freezes the block in place.

That's a lot of new code, so let's try to narrow-down the area to search. Start by setting a breakpoint inside of the `DropBlock` method then start a new game and wait for the block to reach the bottom:

```
1 reference
238 private bool DropBlock(Point position, Block block)
239 {
240     foreach (var point in block.GetCoordinates())
241     {
242         if (point.Y + position.Y < 0)
243         {
244             // Not enough room
245             return false;
246         }
247
248         _grid[point.X + position.X, point.Y + position.Y] = block.Brush;
249     }
```

You'll see that the breakpoint is never hit. So the issue isn't in the `DropBlock` method but in whatever is supposed to call it. That's good, that's 40 lines of code we won't have to understand.

So let's focus on the loop in `UpdateGame`. We know that the block moves down, so the `_currentPosition.Y++;` line is correctly executed. The issue is likely in the collision logic that follows:

```
// Check for collisions
foreach (var coordinate in _currentBlock.GetCoordinates())
{
    if (IntersectsWithSomething(coordinate))
    {
        if (!DropBlock(_currentPosition + (0, -1), _currentBlock))
        {
            GameOver();
            return;
        }

        _currentBlock = null;
        break;
    }
}
```

If you execute that code step-by-step, you'll notice that it's tedious to inspect the value of some of the variables, such as `coordinate`:

```
// Check for collisions
foreach (var coordinate in _currentBlock.GetCoordinates())   coordinate = {Point}
{
    coordinate {DebugMeow.Games.Tetris.Point}
    if (IntersectsWithSomething(coordinate))   ≤1ms elapsed   coordinate = {Point}
    {
```

Every time, you have to drill down into the fields of the objects to check the values. Let's take a minute to make our life easier.

There is a way to instruct the debugger on how to display the value of the objects, using the `DebuggerDisplay` attribute (in the **System.Diagnostics** namespace). Navigate to the `Point` class, and decorate it with the following attribute:

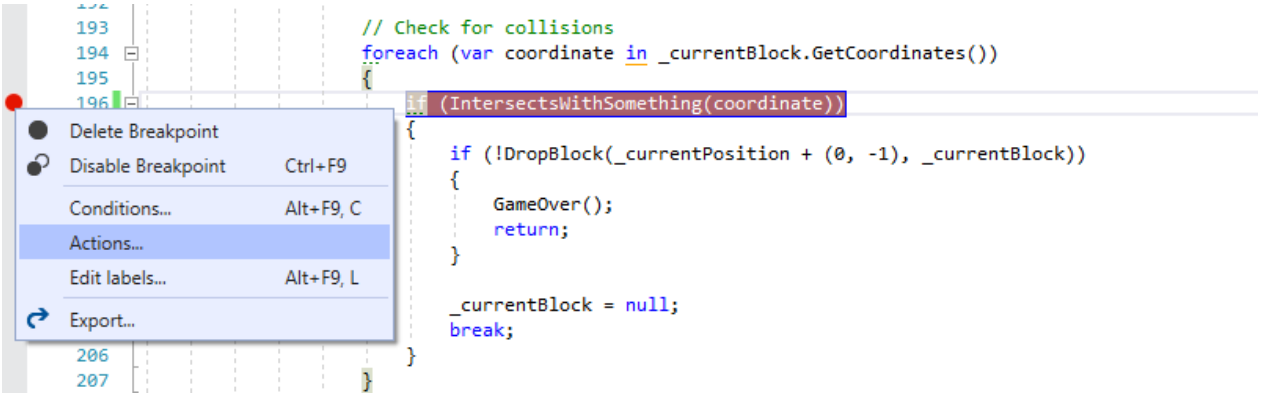
```
[DebuggerDisplay("{X},{Y}")]
```

```
[DebuggerDisplay("{X},{Y}")]
23 references
public struct Point
{
    public int X;
    public int Y;
```

The syntax is a lot like `String.Format`, using `{}` to indicate which properties to display. After doing that, restart the application and break again in the `UpdateGame` method. The coordinates are now automatically displayed the way you asked, which makes the code much easier to follow:

```
// Check for collisions
foreach (var coordinate in _currentBlock.GetCoordinates())   coordinate = 0,0
{   ≤1ms elapsed
    if (IntersectsWithSomething(coordinate))
    {
        if (!DropBlock(_currentPosition + (0, -1),  currentBlock))
        {                                  _currentPosition 5,-2
            GameOver();
            return;
        }
```

Even then, loops are still hard to debug. It becomes quickly apparent that `IntersectsWithSomething` never returns true. Rather than trying to understand the whole logic, it would be helpful to see how the value of the coordinates evolve over time. This is a good use-case for tracepoints. Start by setting a breakpoint on `if (IntersectsWithSomething(coordinate))`, then right-click the red dot in the margin and select **Actions**.

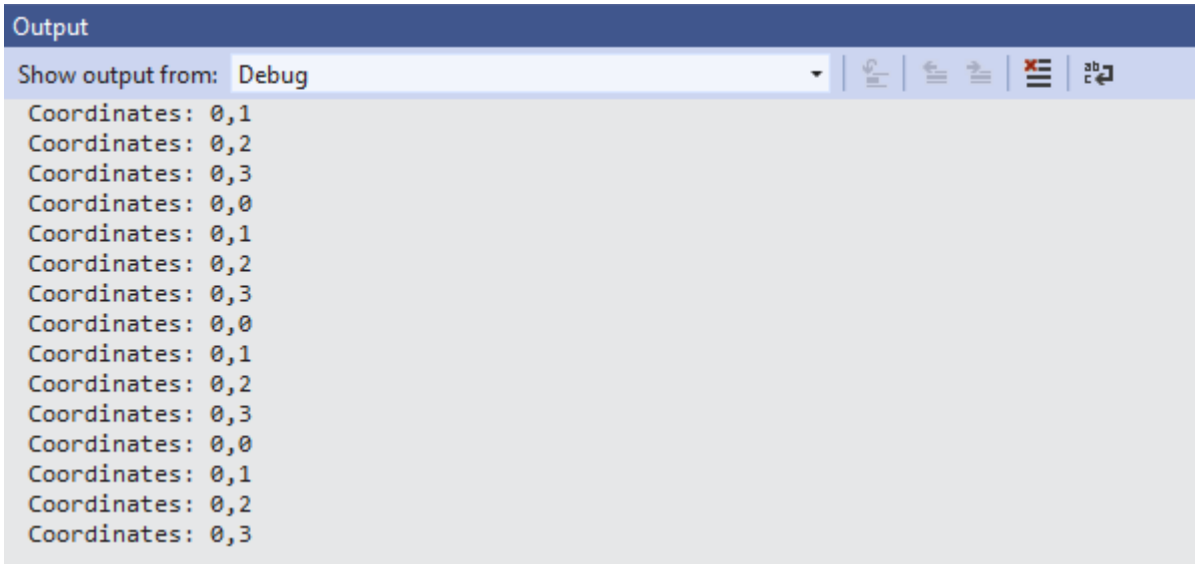


Make sure that “Continue code execution” is checked, and in “Show a message in the Output Window” type:

Coordinates: {coordinate}

This tells the debugger to display that message every time the breakpoint is hit, instead of stopping the execution. The syntax is the same as for the `DebuggerDisplay` attribute.

Resume the execution, and you’ll see the value of the coordinates displayed in the **Output** panel (**Debug | Windows | Output**).



It looks like the coordinates are always the same values repeating again and again, even though the block is supposed to be moving!

In light of this revelation, the issue becomes more apparent in the code. It’s calling `IntersectsWithSomething` with `coordinate` (which is the shape of the block) instead of `coordinate + _currentPosition`.

Fix the line and restart the application:

```
if (IntersectsWithSomething(coordinate + _currentPosition))
```

Now you’ll see that the block is correctly dropped when reaching the bottom.

Third problem – rotating blocks

The third and last issue looks very similar to the first one. Start a new game and try pressing the up arrow to rotate a block. You'll see that nothing happens. Let's figure out what the problem is.

Just like moving a block happened in `MoveBlock`, rotating a block happens in `RotateBlock`. Like previously, it's hard to spot an error at a glance:

```
1 reference
private void RotateBlock()
{
    if (_lastRotate.ElapsedMilliseconds < RotateSpeedInMilliseconds)
    {
        return;
    }

    _lastRotate.Restart();

    int newRotation = _currentBlock.Rotation + 1 % 4;

    foreach (var point in _currentBlock.GetCoordinates(newRotation))
    {
        if (IntersectsWithSomething(_currentPosition + point))
        {
            return;
        }
    }

    _currentBlock.Rotation = newRotation;
}
```

The logic is very similar to `MoveBlock`: first a rate limiter, then the code checks if the block can be rotated without colliding with anything, and finally applies the rotation if everything is fine.

Set a breakpoint on the first line of the method, then start the game and press the up-arrow key. This causes the debugger to break in the method.

```
1 reference
private void RotateBlock()
{
    if (_lastRotate.ElapsedMilliseconds < RotateSpeedInMilliseconds)  _lastRotate = {Stopwatch}, RotateSpeedInMilliseconds = 300
    {
        return;
    }

    _lastRotate.Restart();

    int newRotation = _currentBlock.Rotation + 1 % 4;

    foreach (var point in _currentBlock.GetCoordinates(newRotation))
    {
        if (IntersectsWithSomething(_currentPosition + point))
        {
            return;
        }
    }

    _currentBlock.Rotation = newRotation;
}
```

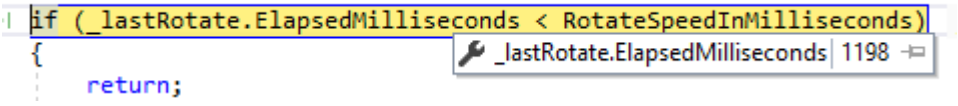
Use **F10** or the step-over icon to execute the code step-by-step (like with problem 1), and...

```
1 reference
private void RotateBlock()
{
    if (_lastRotate.ElapsedMilliseconds < RotateSpeedInMilliseconds)
    {
        return;
    }

    _lastRotate.Restart();

    int newRotation = _currentBlock.Rotation + 1 % 4;  newRotation = 1

    foreach (var point in _currentBlock.GetCoordinates(newRotation))
    {
        if (IntersectsWithSomething(_currentPosition + point))
        {
            return;
        }
    }

    _currentBlock.Rotation = newRotation;
}  ≤1ms elapsed
```

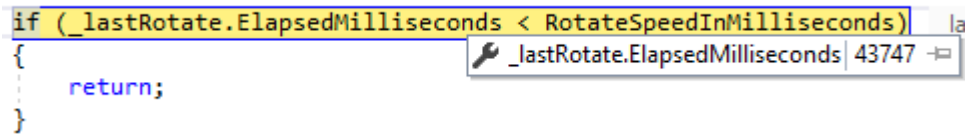
You'll reach the bottom of the method just fine! Resume the execution of the application, and you'll see that the block rotated. The pattern can be repeated: if you run the code step by step, it works. If you run the code without breaking into the debugger, the block won't rotate.

It means that the state of the application is impacted by the debugging. It's usually one of two possibilities: either you're inspecting the value of a property that has side-effects, or it's a timing issue. The former is unlikely to happen if you don't manually set watches, as Visual Studio is very cautious not to evaluate properties with side-effects unless asked to. So let's assume it's a timing issue.

The first line of the method is `if (_lastRotate.ElapsedMilliseconds < RotateSpeedInMilliseconds)`. It does look like it would be impacted by timing. If you break the execution on that line and inspect `_lastRotate.ElapsedMilliseconds`, you'll see one value:

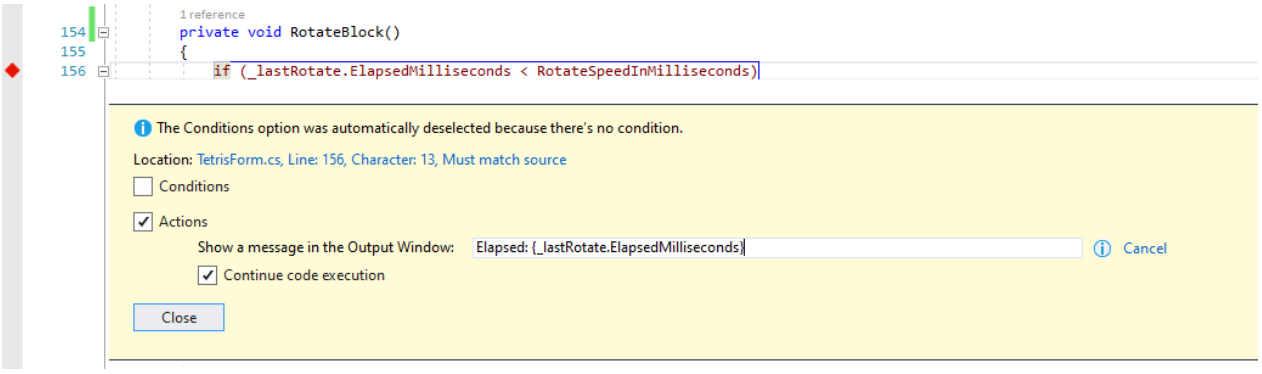


Then reevaluate it once more (by moving the mouse away then back on the property) and you'll see another value:



Just because we're spending time in the debugger, we're causing the value of `_lastRotate.ElapsedMilliseconds` to change, which may change the outcome of the `if (_lastRotate.ElapsedMilliseconds < RotateSpeedInMilliseconds)` condition. You can confirm that by setting a breakpoint on `_lastRotate.Restart();` and trying to rotate a block: the breakpoint won't be hit.

How to observe the behavior of the application without impacting the timings too much? It's another good use-case for tracepoint. Just like in exercise 2, set a breakpoint on `if (_lastRotate.ElapsedMilliseconds < RotateSpeedInMilliseconds)`, then configure it to display the value of `_lastRotate.ElapsedMilliseconds` when hit:



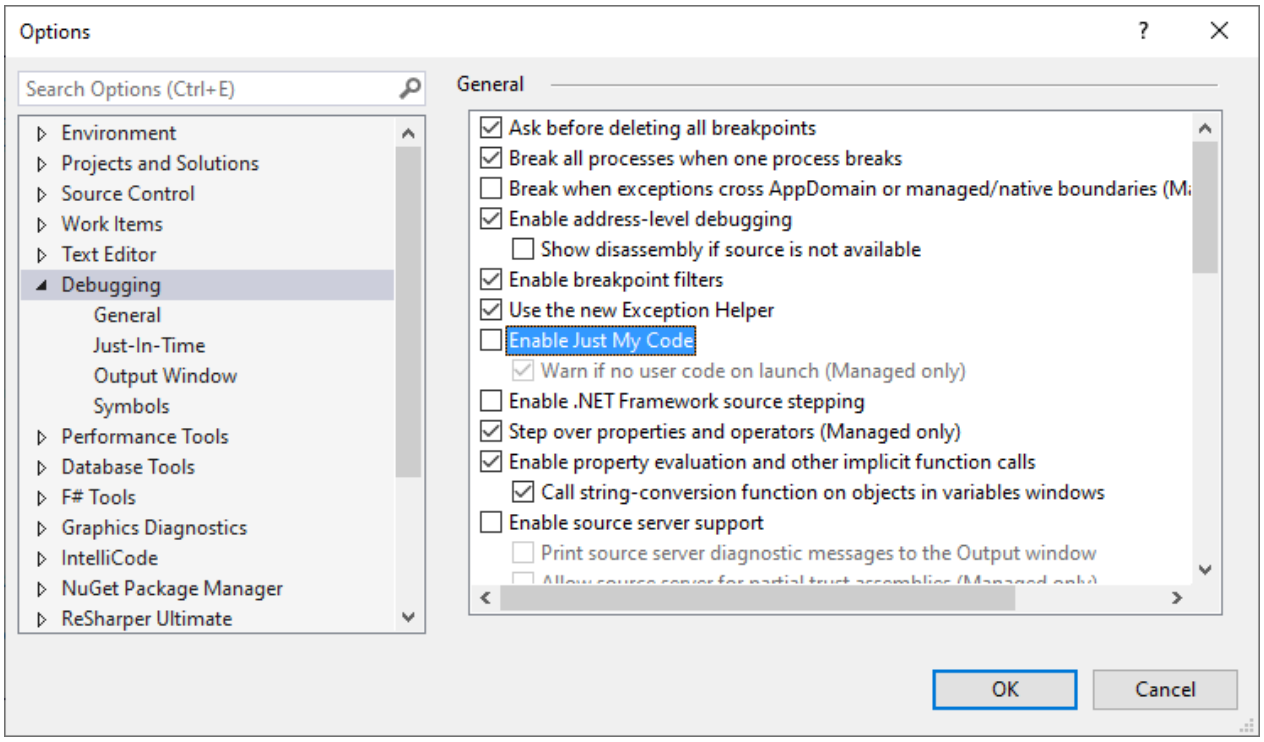
Resume the execution and try rotating the block. You'll see that the values change but are never bigger than `RotateSpeedInMilliseconds` (300).

```
Elapsed: 41
Elapsed: 69
Elapsed: 97
Elapsed: 123
Elapsed: 92
Elapsed: 119
Elapsed: 146
Elapsed: 171
Elapsed: 17
Elapsed: 45
Elapsed: 40
Elapsed: 65
Elapsed: 106
Elapsed: 135
```

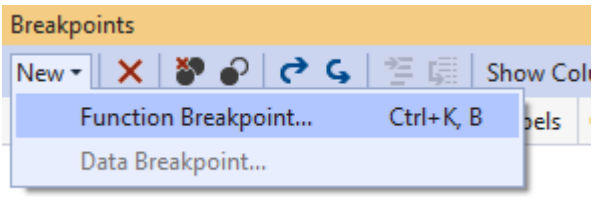
Furthermore, it looks like the value sometimes gets lower, even though it's supposed to measure an elapsed time. It likely means that something is resetting the `_lastRotate` stopwatch.

How to find the culprit? If you search for `_lastRotate.Restart();` in the project, you'll find `ButtonStart_Click` (which is called only when starting the game), and `RotateBlock` (but we know we never reach that line). Is there a more exhaustive way to find who calls Stopwatch.Restart?

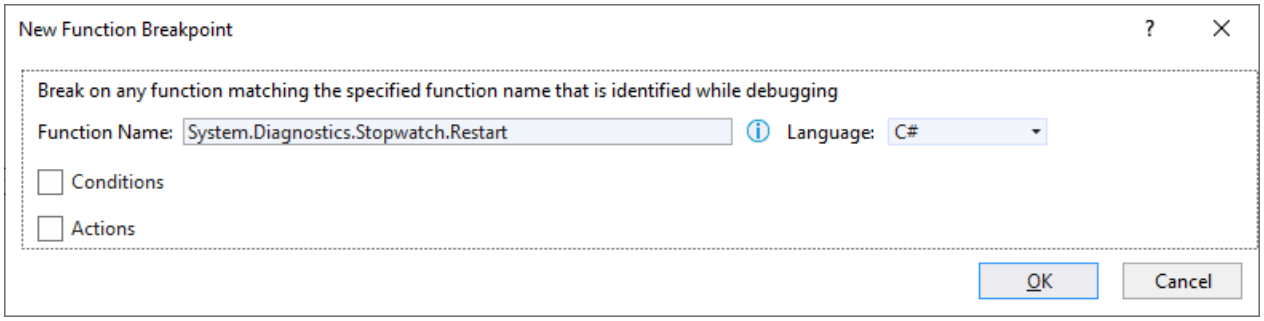
Just like you can set a breakpoint on any method in your code, you can set a breakpoint on any method of any library. But you need a bit of preparation first. Go in the **Tools | Options** menu, select **Debugging**, and make sure "Enable Just My Code" is unchecked.



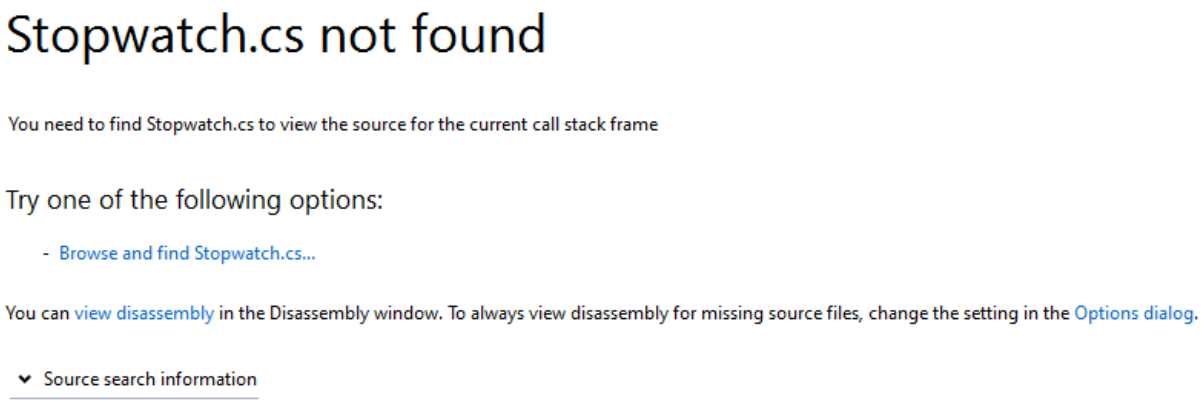
Now open the **Breakpoints** panel (**Debug | Windows | Breakpoints**), click **New** and select **Function Breakpoint**



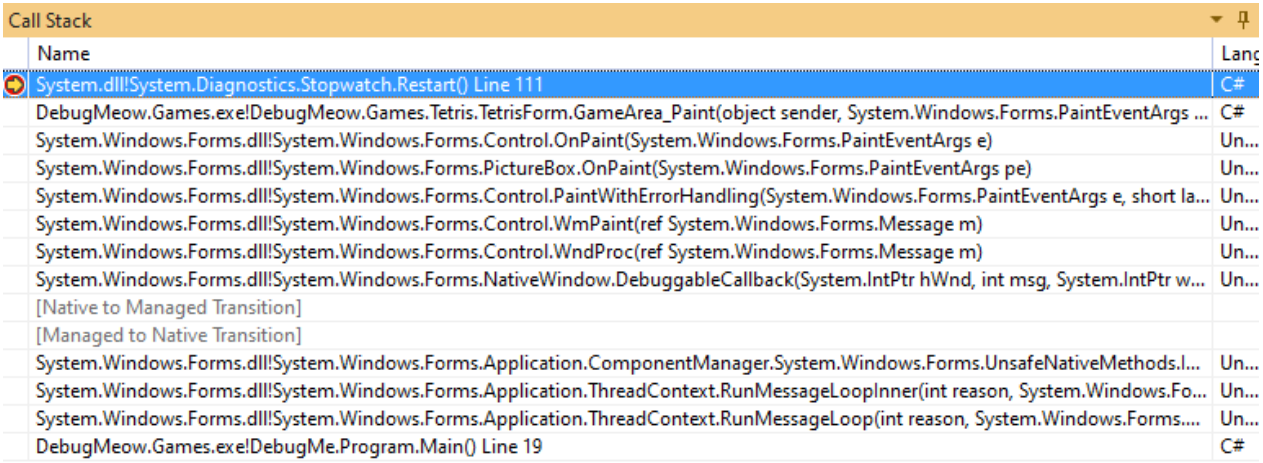
As “Function Name”, type System.Diagnostics.Stopwatch.Restart and click OK:



Start the application, and you’ll see that screen:



It means that the debugger hit the breakpoint but has no source code to show. Look at the **Call Stack** panel to see who called the method:

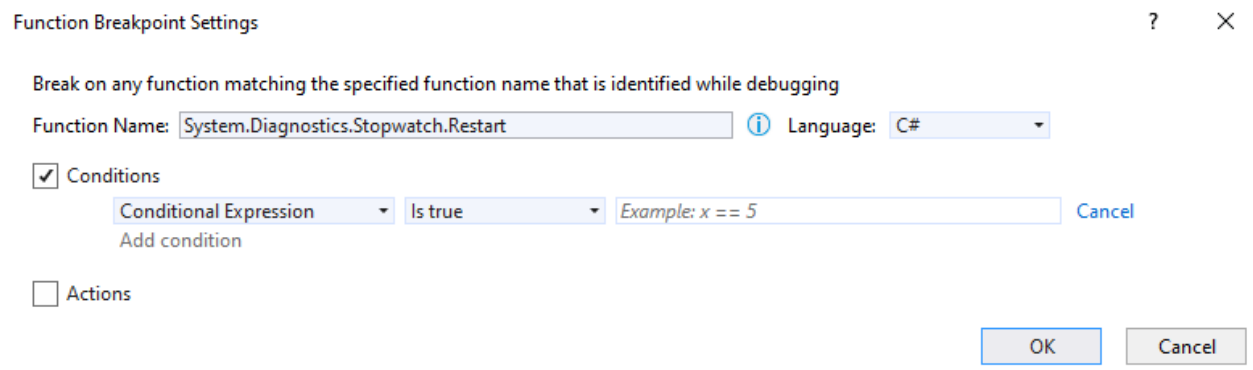


Then double-click the parent frame (here: **DebugMeow.Games.exe!DebugMeow.Games.Tetris.TetrisForm.GameArea_Paint**) to navigate to the code:

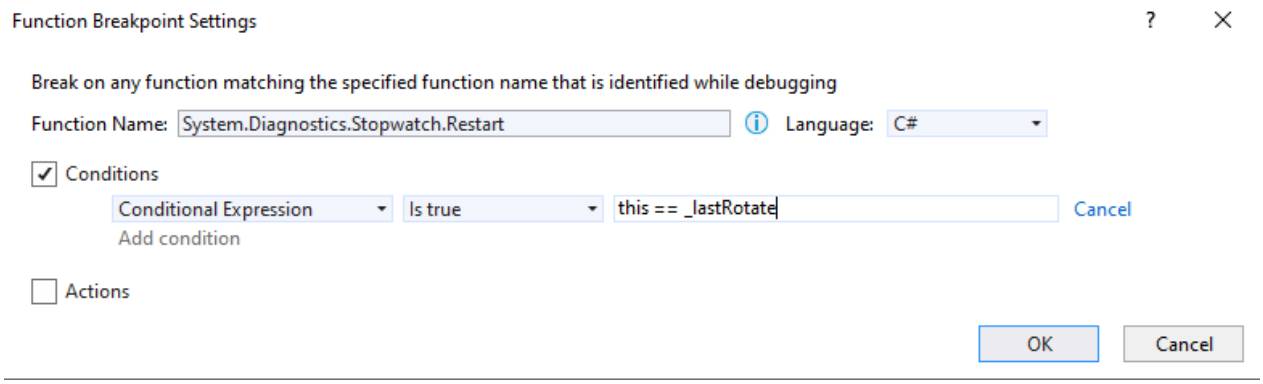
```
1 reference
private void GameArea_Paint(object sender, PaintEventArgs e)  sender = {Syst
{
    if (IsDisposed)
    {
        return;
    }

    _lastFrameDelays.Enqueue((int)_lastFrame.ElapsedMilliseconds);
    _lastFrame.Restart();   _lastFrame = {Stopwatch}
```

The breakpoint is working, but there is a Stopwatch that is reset every time the game is rendered, which is going to make the debugging very painful. How to configure the breakpoint to break only when Restart is called on `_lastRotate`?

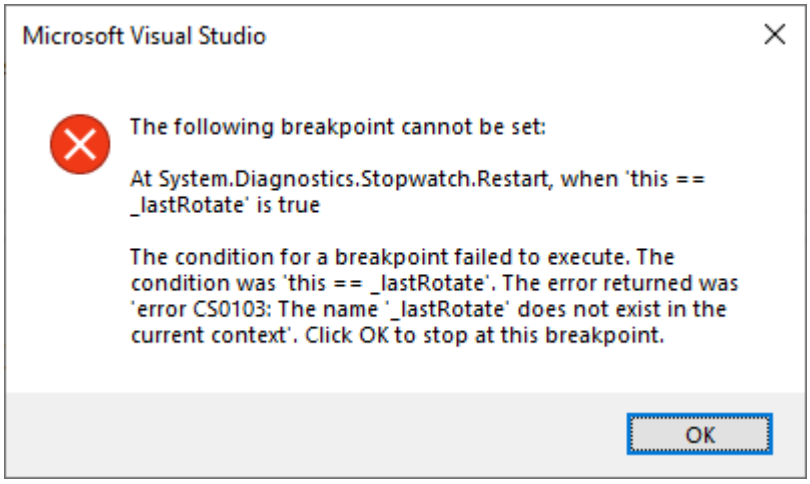
We could set a condition in the breakpoint, like we did for the first issue. For that, right-click the breakpoint in the **Breakpoints** panel, click **Settings**, then check “Conditions”:



The condition you set on a breakpoint is evaluated in the same context as the method on which you set the breakpoint. In other words, if you set a breakpoint on `Stopwatch.Restart`, you can only use all the variables that are reachable from within `Stopwatch.Restart`. So "this" would evaluate to the instance of the `Stopwatch`. Then, can we set `this == _lastRotate` as condition?



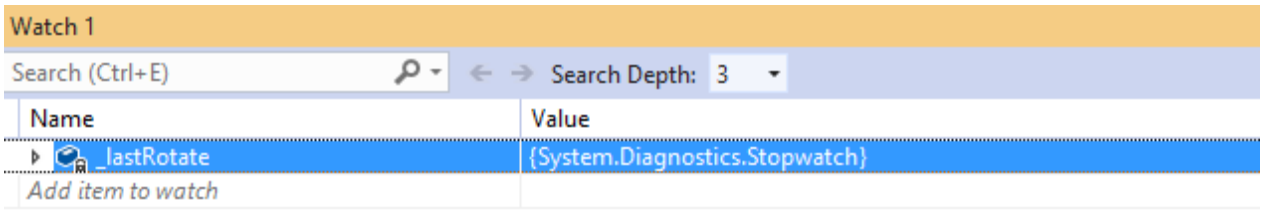
If you try that and resume the execution, you will see the following error message:



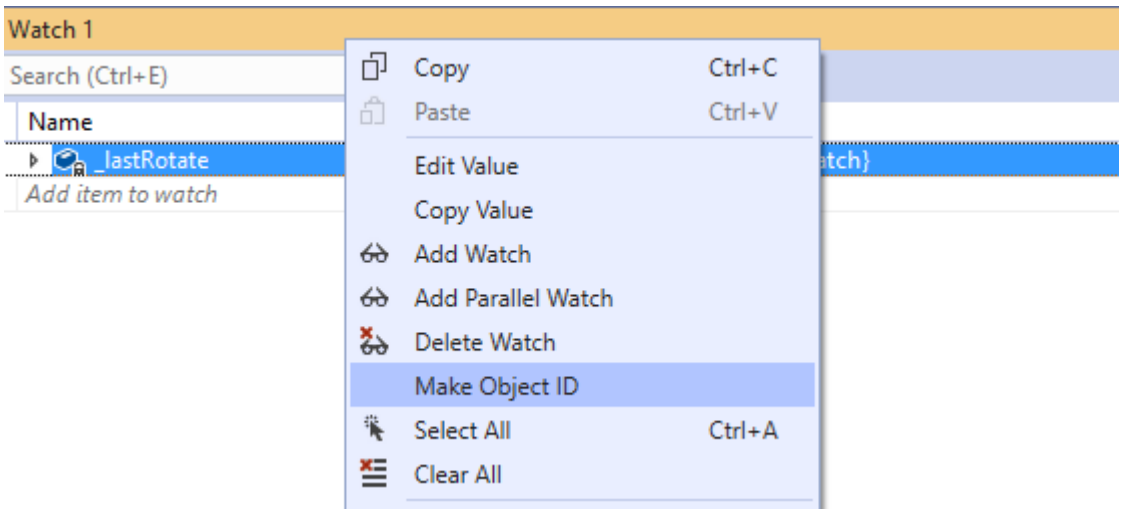
The statement "The condition you set on a breakpoint is evaluated in the same context as the method on which you set the breakpoint" works both ways: it means you can evaluate "this" to the instance of `Stopwatch`, but it also means that `_lastRotate` isn't reachable from within the `Stopwatch`!

For this to work, we would need a way to make `_lastRotate` globally reachable, like a static field. We can achieve this result by using the object ID feature of Visual Studio.

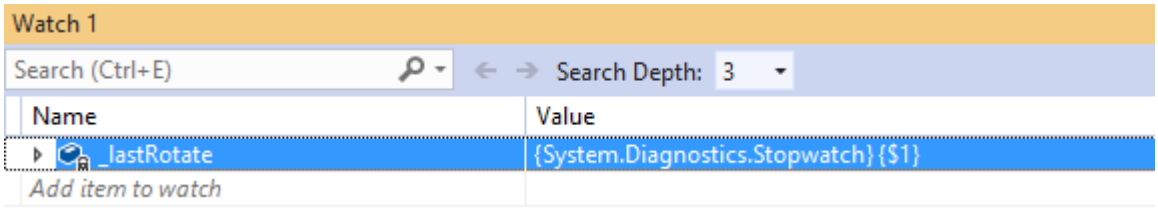
First, break the execution at a place where `_lastRotate` is reachable, and add it to the **Watch** panel:



Now right-click on it, and select Make Object ID:



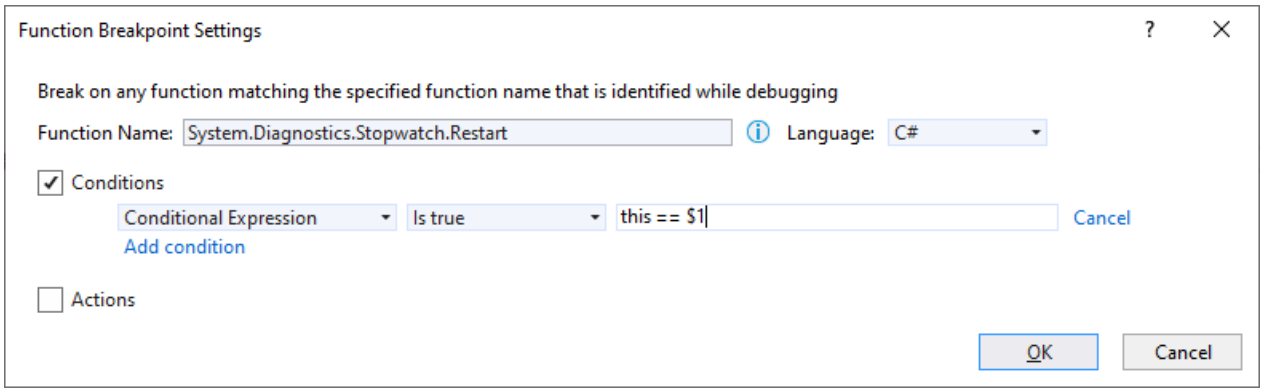
You now see a $1 next to the value:



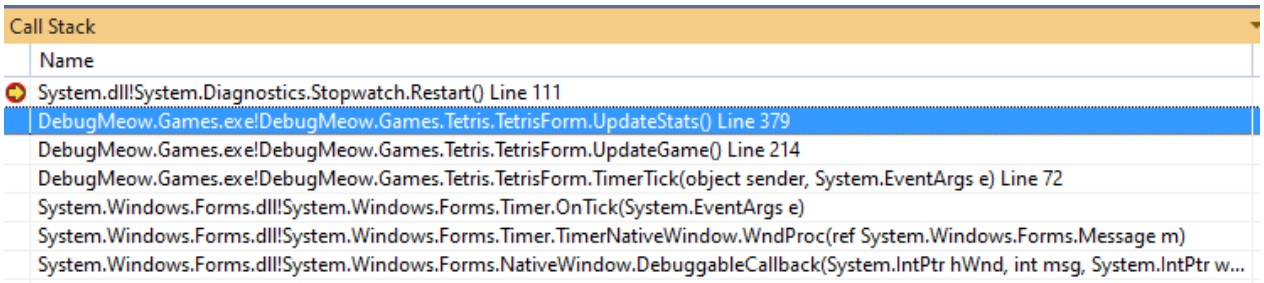
After doing that, you can reference this object from anywhere by evaluating $1.

Note: if you stop the execution after this point, you will have to do the same procedure again to create an object ID. Object IDs do not persist from a debugging session to another.

Go back to the breakpoint on `Stopwatch.Restart` and set the condition to `this == $1`



Then resume the execution. When the breakpoint is hit, switch to the caller using the **Call Stack** panel:



You can see that the `UpdateStats` method, called every frame, resets the `_lastRotate` Stopwatch:

```
1 reference
private void UpdateStats()
{
    if (_lastFrameDelays.Count > 5)
    {
        LabelDelay.Text = (int) _lastFrameDelays.Average() + "ms";
        _lastFrameDelays.Clear();
        var s = _lastRotate;   s = {Stopwatch}{$1}
        s.Restart();   s = {Stopwatch}{$1}
    }
}
```

Just remove the call to `Restart` and the bug will be fixed.