Object Oriented Programming with Applications
Lecture 4

David Šiška & Witold Gawlikowicz

School of Mathematics, University of Edinburgh

2018-19[1]

[1] Last updated 10th October 2018

Lecture 4

- Enumeration types
- Value vs Reference types
- Exception handling
- Debugging

Read:
Wright, P. - Beginning Visual C# 2005 Express Edition. Chapter 5.
Duffy, D. J. and Germani, A. - C# for Financial Markets, Chapter 3, and Chapter 4 Sections 4.1–4.5.

Enumeration types

A data type that allows you to give names to numbers.

One can try to remember that e.g. '0' indicates call option, '1' indicates put, '2' indicates Bermudan put option etc.

This works but makes the code un-readable and leads to mistakes. Example

```
public enum BasicOptionTypes { Call, Put, BermudanPut,
                               AmericanPut, Digital };
```

Stack vs Heap

- Virtual memory available to a .NET program is divided between Stack and Heap
- Stack
  - Used for static memory allocation (compile-time)
  - Value type variables stored here
  - Memory freed when variable no longer in scope
  - Very fast
- Heap
  - Used for dynamic memory allocation (run-time)
  - Reference type variables stored here
  - Memory managed by Garbage Collector
  - Slower than stack

Value types and reference types

value type variable is created on the stack and it is popped off the stack when it goes out of scope. The variable contains the value. Variables of this type are passed by value (in other words, a copy of the variable is made) and it is copied when it is assigned to other variables. Examples of value types are intrinsic (built-in) types.

reference types variable data is created on the heap. Thus, reference type variables are not copied and it is possible to define several variables that reference the same object in memory. Objects, strings and arrays are reference data types and we create variables of these types in combination with the keyword 'new'.

Reference type variables

```
Date d1 = CreateDate(1,1,2016); // this is a valid date
                                //and 'new' keyword is used
Date d2 = d1;

d1.SetDayOfMonth(2); // still a valid date
WriteLine("Day of month in d1: {0}, d1.GetDayOfMonth());
```

So far so good. But what is the output here?

```
WriteLine("Day of month in d2: {0}, d2.GetDayOfMonth());
```

Answer: `Day of month in d2: 2`.

Reference type variables are not copied and it is possible to define several variables that reference the same object in memory.

Methods: Pass by value vs. pass by reference

Pass by value:

```
static void TestPassByValue(int a)
{
     a = 0;
}
static void Main(string[] args)
{
    int a = 1;
    Console.WriteLine("{0}",a);
    TestPassByValue(a);
    Console.WriteLine("{0}",a);
}
```

What is the output?

Pass by reference:

```
static void TestPassByValue(int[] a)
{
     a[0] = 0;
}
static void Main(string[] args)
{
    int[] a = new int[2] {1,2};
    Console.WriteLine("{0}",a[0]);
    TestPassByValue(a);
    Console.WriteLine("{0}",a[0]);
}
```

What is the output?

Methods: Force pass by reference and copy

Pass a value type by reference:

```
static void PassByRef(ref int a)
{
     a = 0;
}
static void Main(string[] args)
{
    int a = 1;
    Console.WriteLine("{0}",a);
    TestPassByValue(a);
    Console.WriteLine("{0}",a);
}
```

What is the output?

Pass by reference and create copy inside method:

```
static void TestPassByValue(int[] a)
{
     int[] myCopy = new int[a.Length];
     a.CopyTo(myCopy, 0);
     myCopy[0] = 0;
}
static void Main(string[] args)
{
    int[] a = new int[2] {1,2};
    Console.WriteLine("{0}",a[0]);
    TestPassByValue(a);
    Console.WriteLine("{0}",a[0]);
}
```

What is the output?

Reference type variables: copy constructor

Change the code to make a copy.

```
Date d1 = CreateDate(1,1,2016); // this is a valid date
                                //and 'new' keyword is used
Date d2 = new Date(d1);

d1.SetDayOfMonth(2); // still a valid date
WriteLine("Day of month in d1: {0}, d1.GetDayOfMonth());
```

So far so good. What is the output now?

```
WriteLine("Day of month in d2: {0}, d2.GetDayOfMonth());
```

Answer: `Day of month in d2:  1`.

`null` keyword

Used to indicate that a reference variable is not pointing to any object.

```
static double NewtonSolver(Func<double, double> f,
                           Func<double, double> fPrime,
                           double x0, double maxError, int maxIter)
{
    double fPrimeOfx0;
    if(fPrime == null) {
        fPrimeOfx0 = (1.0 / 2*delta)* (f(x0+delta) - f(x0 - delta));
    }
    else {
        fPrimeOfx0 = fPrime(x0); // use directly
    }
}
```

Can be called as

```
NewtonSolver(xSquaredMinusFour, xSquaredMinusFourPrime, 4, 1e-10,10));
NewtonSolver(xSquaredMinusFour, null, 4, 1e-10,10));
```

Exception handling

Exceptions are a technique for dealing with errors.

- Exceptions are objects derived[2] from `System.Exception`.
- To raise an error i.e. "throw an exception" use e.g.

```
throw new System.ArgumentException ("Incorrect input argument.");
```

- Handle exception

```
Date d3;
try
{
    d3 = Date.CreateDate(32,1,2015);
    Console.WriteLine("Day of month in d3: {0}", d3.GetDayOfMonth());
}
catch (System.ArgumentException e)
{
    Console.WriteLine ("Something bad happened: "+e.Message);
    return 1;
}
```

[2] We will discuss inheritance in more detail.

More exception handling

- Can separately catch exceptions of different types:

```
try  {
    // Some risky code
}
catch (System.ArgumentException e)  {
    // Some action for this type of exception
}
catch (System.AccessViolationException e) {
    // Some other action for this type of exception
}
finally {
    // this will be run whether exception was caught or not.
}
```

- You can re-throw a caught exception

```
try  {
    // Some risky code
}
catch (System.ArgumentException e)  {
    // Do some partial error handling
    throw; // but leave the rest to the level above.
}
```

What are bugs?

Bugs are a catch all term for mistakes made while coding that can cause the program to:

- Crash.
- Produce incorrect output e.g. from a calculation.
- Mismanage resources e.g. keep opening network connections, leak memory etc.

If your code won't compile that's not a bug. That is the compiler trying to help you write correct code.

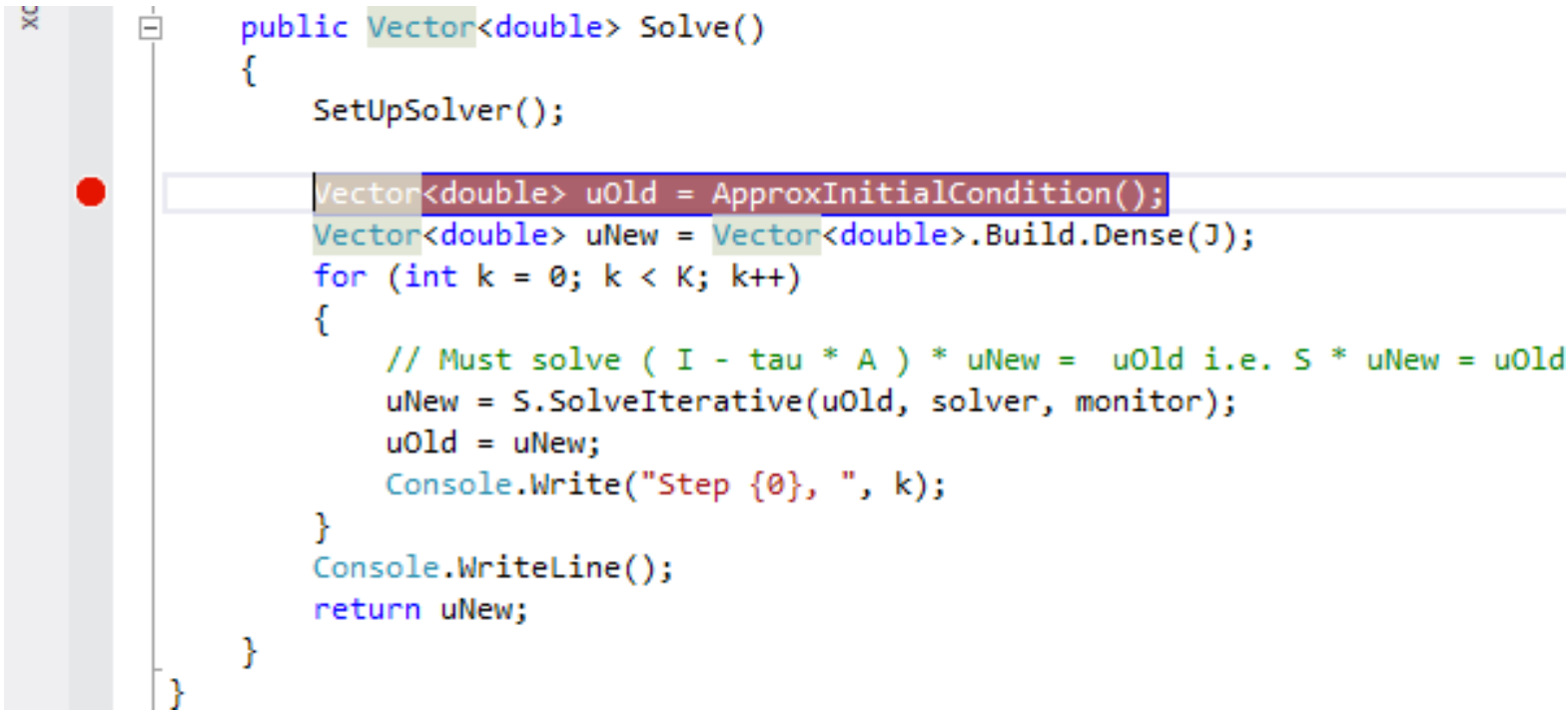
What is debugging

Process of inspecting the program behaviour, while it is running with, the aim to find bugs.

We will now discuss various aspects of the debugger.

Breakpoints

You can place a *breakpoint* at any line in your code.



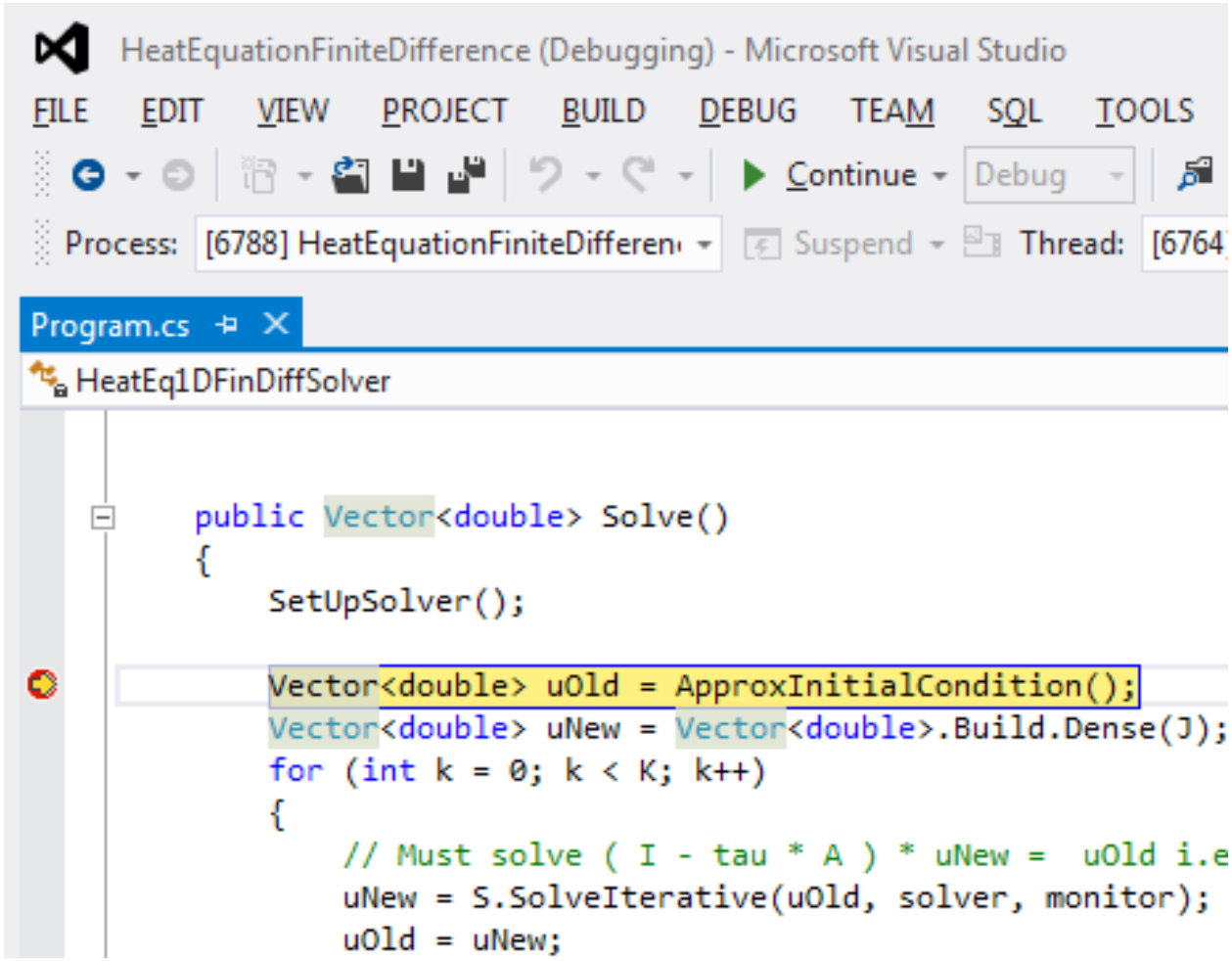
```
    public Vector<double> Solve()
    {
        SetUpSolver();

        Vector<double> uOld = ApproxInitialCondition();
        Vector<double> uNew = Vector<double>.Build.Dense(J);
        for (int k = 0; k < K; k++)
        {
            // Must solve ( I - tau * A ) * uNew =  uOld i.e. S * uNew = uOld
            uNew = S.SolveIterative(uOld, solver, monitor);
            uOld = uNew;
            Console.Write("Step {0}, ", k);
        }
        Console.WriteLine();
        return uNew;
    }
}
```

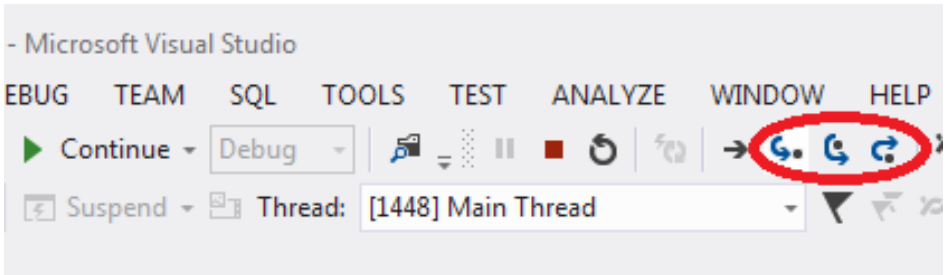
Press “Start” (as you normally do to run the program).

Breakpoints

The execution of the program will be paused when the breakpoint is reached.



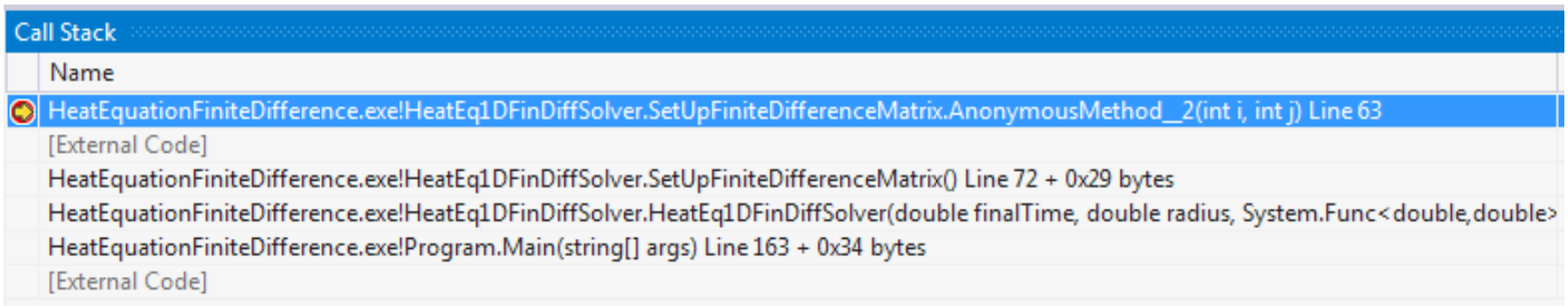
Stepping



- Step over: move to the next "statement" i.e. next ;
- Step into: move into the method being called on that line (if applicable).
- Step out: move out of the current method and into the code that called this method.

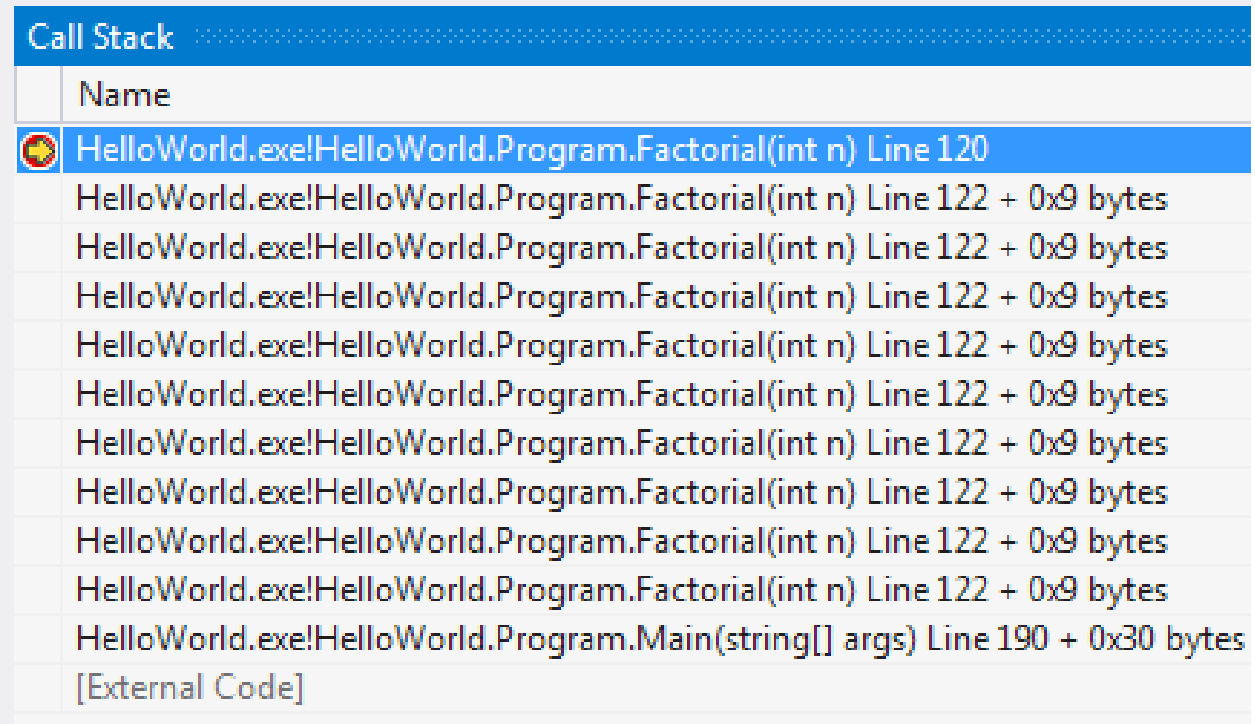
Call stack

Call stack stores information about the active methods of a computer program.



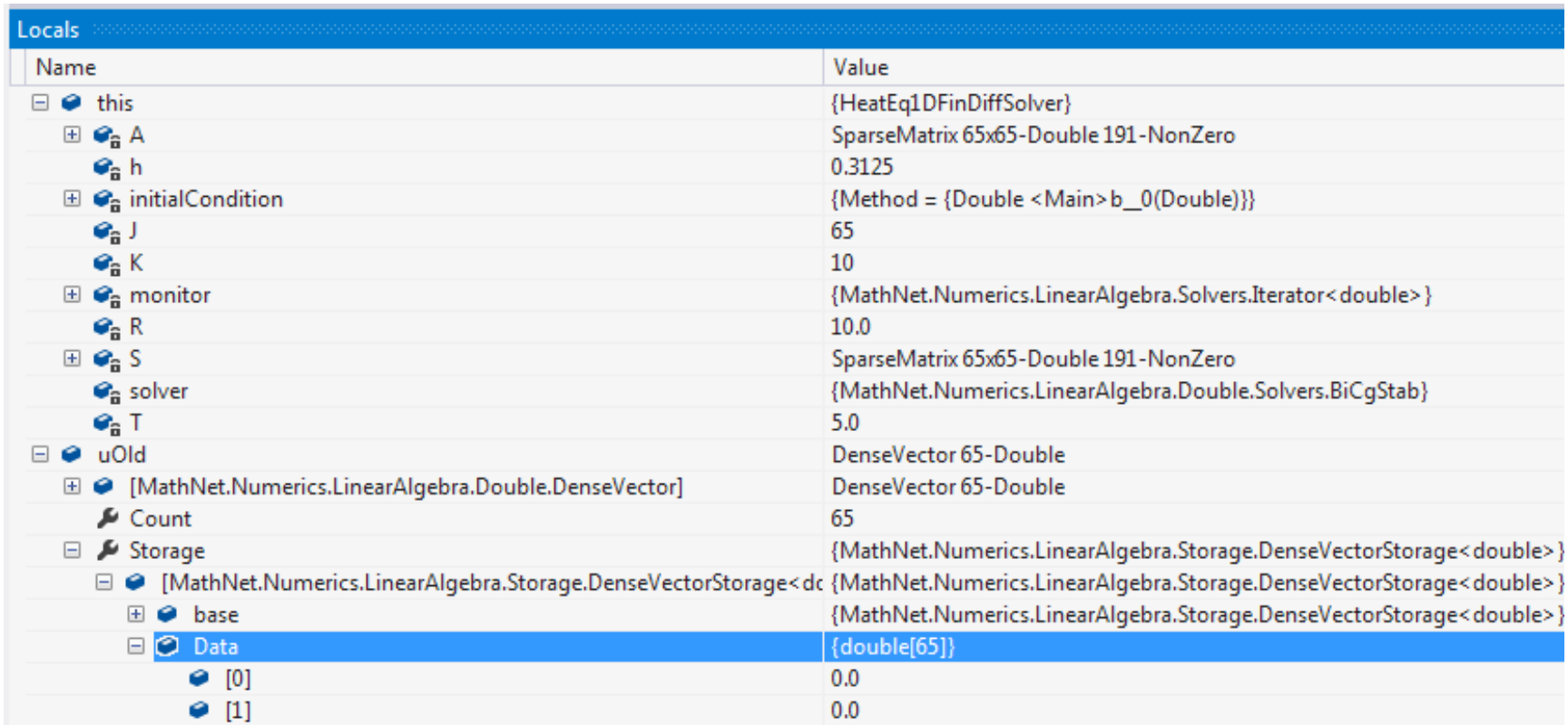
Call stack - example

Called `Factorial(10)` with breakpoint on `return n`. The call stack:



Viewing variable values

If you stop at a breakpoint or are stepping through code then you can see the values your variables take in the "Autos" and "Locals" windows:



Exceptions and debugging

If your code causes an exception (e.g. you try to access an array element that does not exist) your code will pause.

You can view the values of variables, call stack etc. to see why you got to this points.

Next week...Lecture 5

Using libraries

- Data types in `System.Collections.Generic`
- Two examples:
  - `LinkedList`
  - `KeyValuePair`
  - `Hashtable`
- Complex numbers in `System.Numerics.Complex`.
- Linear algebra in `MathNet.Numerics.LinearAlgebra`.
  - `MathNet.Numerics.LinearAlgebra.Vector`
  - `MathNet.Numerics.LinearAlgebra.Matrix`

Related reading: Wright, P. - Beginning Visual C# 2005 Express Edition. Chapter 13.
Duffy, D. J. and Germani, A. - C# for Financial Markets, Chapter 5.

Next week...Lecture 6

- Input / output with Excel:
  - Library vs. Executable
  - ExcelDNA
  - Basic input & output
- What makes good code?
- Overview of assignment 0
- Submission and grading mechanism