SOFTWARE ENGINEERING

5TH LECTURE

Today

- IO operations
- Serialization
- Exceptions
- Delegates
- AOP
- Lambda and anonymous methods

Data input and output (1)

- .NET has a list of classes that allows us to do the following with the files:
  - copy, move, delete, check if exists, etc.
  - They are typically located in System.IO namespace.
- File – a collection of bytes saved in storage and having a name.
- When working with files, *streams* are typically used.
  - Stream allows to process files in chunks.

Data input and output (2)

Main classes in **System.IO**:

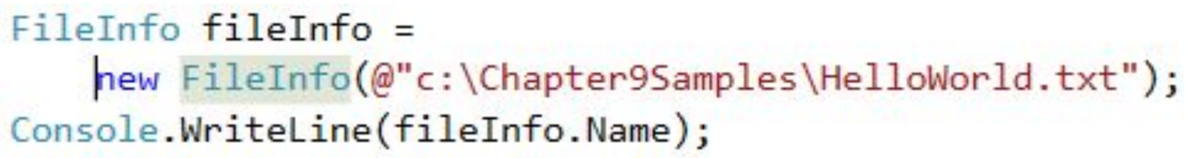
Class	Description
File	A **static** class that provides methods for creating, copying, deleting, moving, and opening files.
FileInfo	Provides for creating an **instance** of a class that provides methods for creating, copying, deleting, moving, and opening file.
Directory	A static class that provides methods for creating, moving, deleting, and enumerating through the files in a directory.
DirectoryInfo	Provides for creating an instance of a class that provides methods for creating, moving, deleting, and enumerating through the files in a directory
Path	A static class that provides methods for obtaining information or manipulating a file or directory name using a string variable.

Data input and output (3)

- FileInfo properties:

Property	Description
Directory	Gets an instance of a DirectoryInfo object for the parent directory
DirectoryName	Gets a string for the directory's full path
Exists	Returns a boolean indicating if the file exists
IsReadOnly	Returns a boolean indicating if the file is read-only
Length	Returns the size in bytes of the file
Name	Gets the name of the file

- Constructor:



```
FileInfo fileInfo =
    new FileInfo(@"c:\Chapter9Samples\HelloWorld.txt");
Console.WriteLine(fileInfo.Name);
```

  – Constructor does not open a file.
  – Method Open() is used for that

File and FileInfo methods

Method	Description
AppendAllText	Creates a StreamWriter that can be used to append text to the file
CopyTo (FileInfo) **Copy (File)**	Copies the current file to given path. Two params: copies file from one path to the other.
Create	Creates the file
Delete	Deletes the file
MoveTo (FileInfo) **Move (File)**	Same as Copy, only moves Same as CopyTo, only moves
Open	Returns a FileStream object for read, write, or read\write access
Replace	Replaces the content of a file with the contents from another file

DirectoryInfo properties

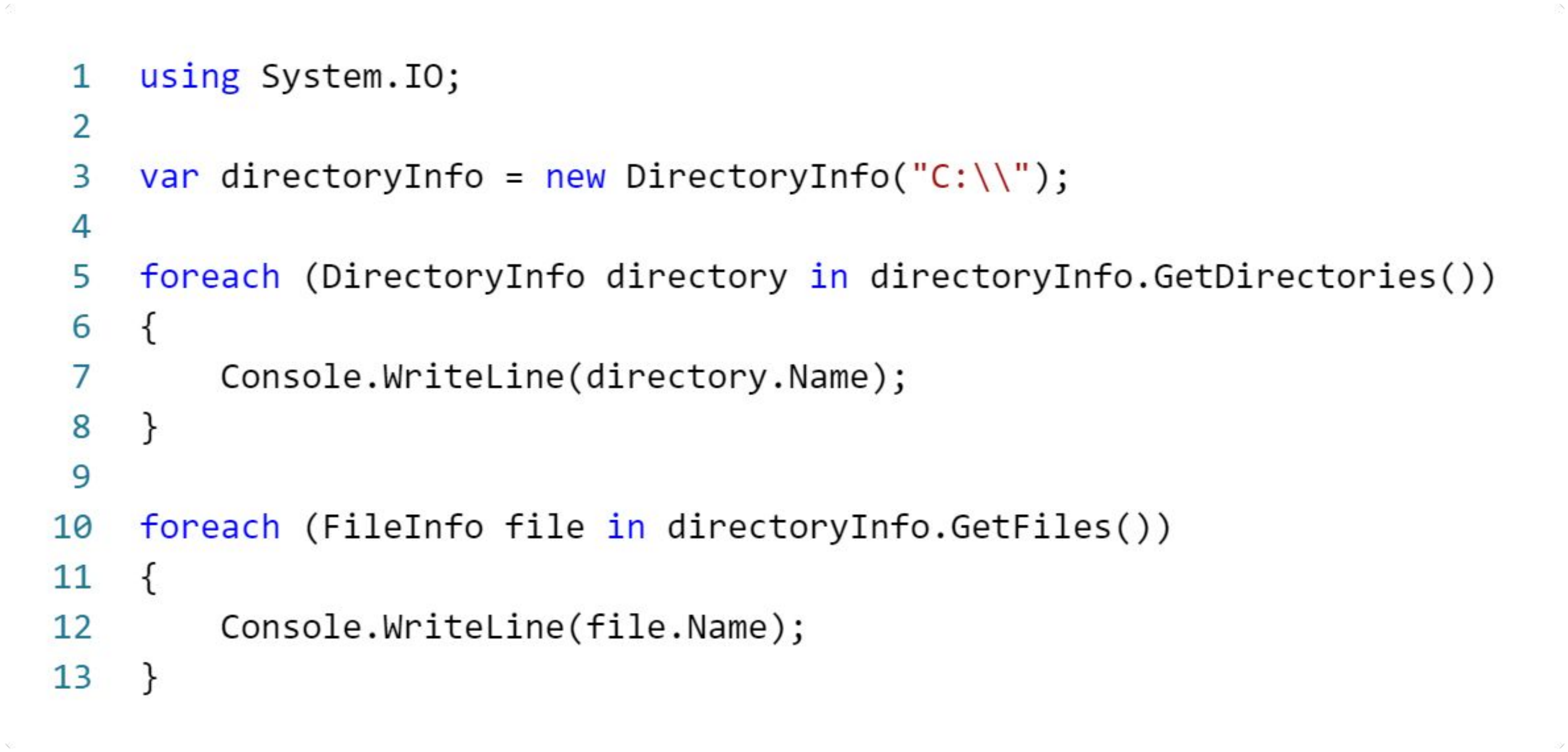
Property	**Description**
Exists	Returns a boolean indicating if the directory exists
Name	Gets the name of the DirectoryInfo instance
Parent	Returns a DirectoryInfo object of the parent directory
Root	Returns a DirectoryInfo object of the root directory

Directory and DirectoryInfo methods

Method	Description
Create (DirectoryInfo) **CreateDirectory (Directory)**	Creates the directory
Delete	Deletes the directory
GetDirectories	Returns a DirectoryInfo array of the subdirectories in the current directory
GetFiles	Returns a FileInfo array of the files in the current directory
GetFileSystemInfos	Returns a FileSystemInfo array of the files and directories in the current directory
MoveTo (DirectoryInfo) **Move (Directory)**	Moves a directory

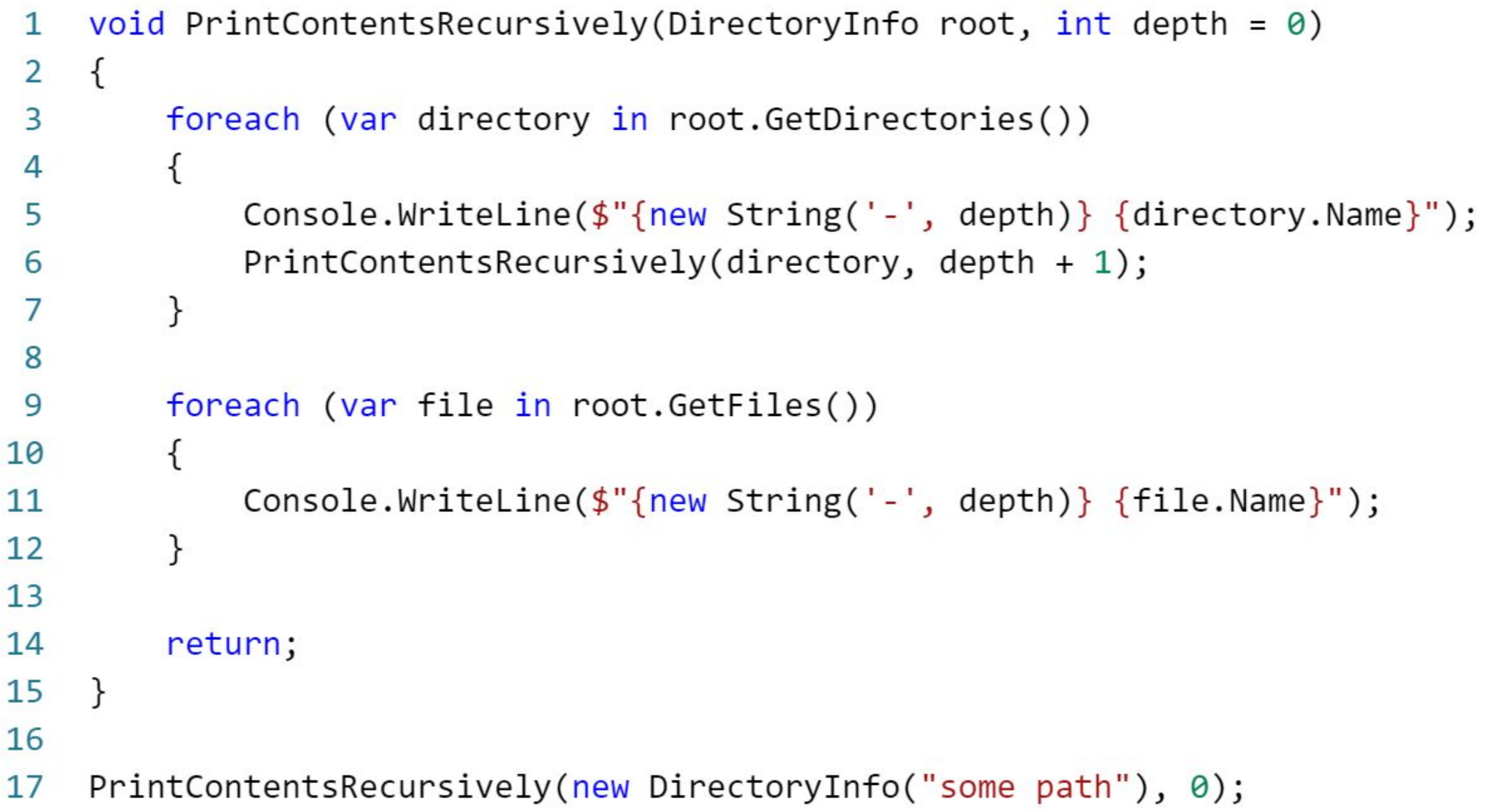
I/O: example

- Print all directories in C:\ and then all the files in C:\.



```csharp
using System.IO;

var directoryInfo = new DirectoryInfo("C:\\");

foreach (DirectoryInfo directory in directoryInfo.GetDirectories())
{
    Console.WriteLine(directory.Name);
}

foreach (FileInfo file in directoryInfo.GetFiles())
{
    Console.WriteLine(file.Name);
}
```

I/O: example (2)



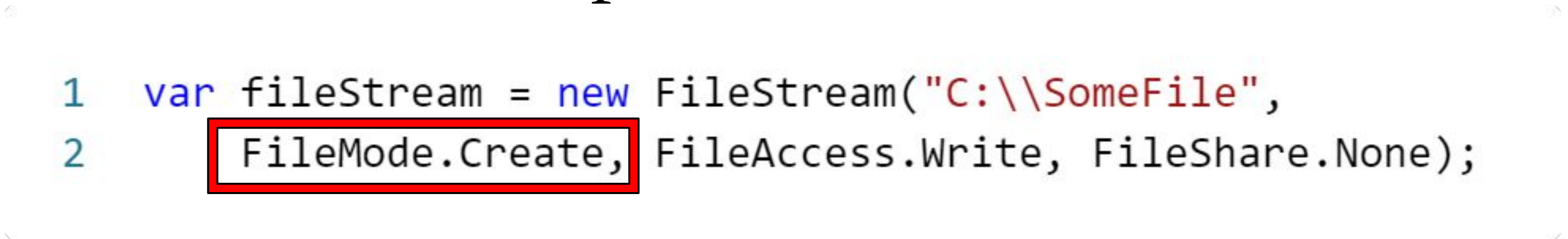
```csharp
void PrintContentsRecursively(DirectoryInfo root, int depth = 0)
{
    foreach (var directory in root.GetDirectories())
    {
        Console.WriteLine($"{new String('-', depth)} {directory.Name}");
        PrintContentsRecursively(directory, depth + 1);
    }

    foreach (var file in root.GetFiles())
    {
        Console.WriteLine($"{new String('-', depth)} {file.Name}");
    }

    return;
}

PrintContentsRecursively(new DirectoryInfo("some path"), 0);
```

I/O: example (3)

```
var directory = new DirectoryInfo("K://");

try
{
    directory.GetDirectories();
}
catch (DirectoryNotFoundException)
{
    Console.WriteLine("Directory does not exist :/");
}
catch (UnauthorizedAccessException)
{
    Console.WriteLine("Cannot access");
}

var file = new FileInfo("fileeee");

try
{
    file.OpenRead();
}
catch (FileNotFoundException)
{
    Console.WriteLine("File does not exist :/");
}
```

I/O: FileStream: FileMode

- Defines how to open the file:

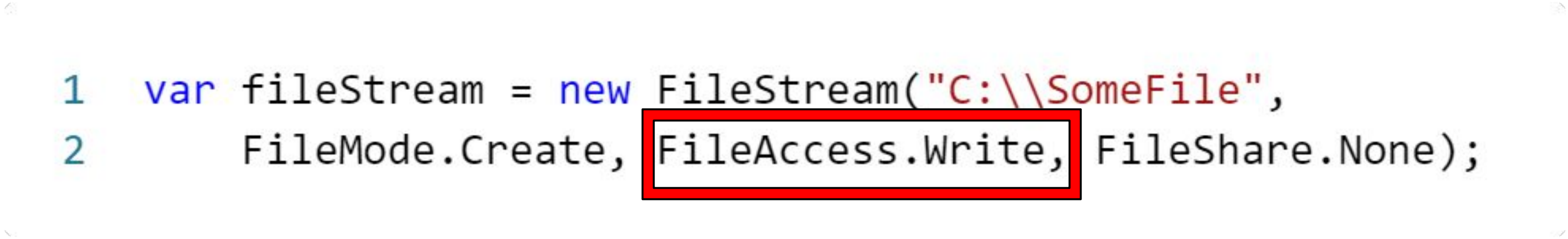


```
var fileStream = new FileStream("C:\\SomeFile",
    FileMode.Create, FileAccess.Write, FileShare.None);
```

Method	Description
Append	Opens a file if it exists and seeks to the end of the file, or creates a new file if it doesn't exist. This can be used only with FileAccess.Write.
CreateNew	Creates a new file. If the file already exists, an exception is thrown.
Create	Creates a new file. If the file already exists it will be overwritten. If the file exists and is hidden, an exception is thrown.
Open	Opens a file. If the file does not exist, an exception is thrown.
OpenOrCreate	Opens a file if it exists or creates a new file if it does not exist.
Truncate	Opens an existing file and truncates the data in the file. If the file does not exist, an exception is thrown.

I/O: FileStream: FileAccess

- Defines what we can do with a stream

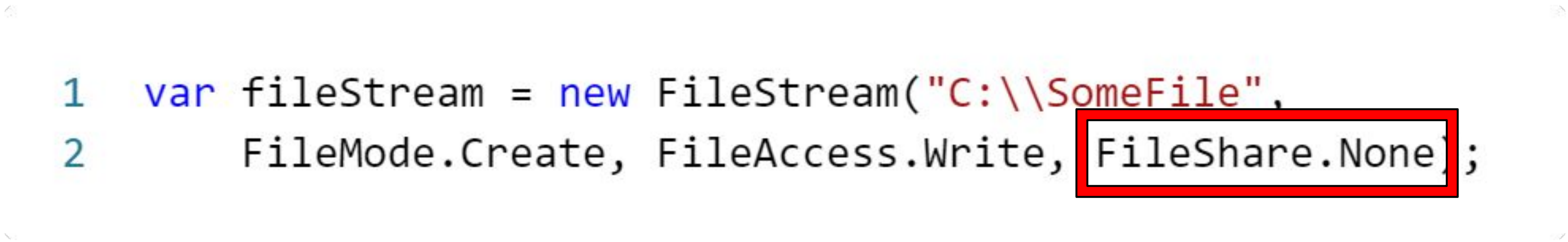


```
var fileStream = new FileStream("C:\\SomeFile",
      FileMode.Create, FileAccess.Write, FileShare.None);
```

Method	**Description**
Read	Allows to read
Write	Allows to write
ReadWrite	Allows to read and write

I/O: FileStream: FileShare

- Defines, what access other FileStream objects will have while current one is using the file:



```
var fileStream = new FileStream("C:\\SomeFile",
        FileMode.Create, FileAccess.Write, FileShare.None);
```

Method	Description
None	Does not enable another stream to open the file
Read	Enables subsequent opening of the file for reading only
Write	Enables subsequent opening of the file for writing
ReadWrite	Enables subsequent opening of the file for reading or writing
Delete	Enables for subsequent deletion of the file

I/O: StreamReader

- Used for text files
- Encoding is given, standard – UTF
- Main methods:

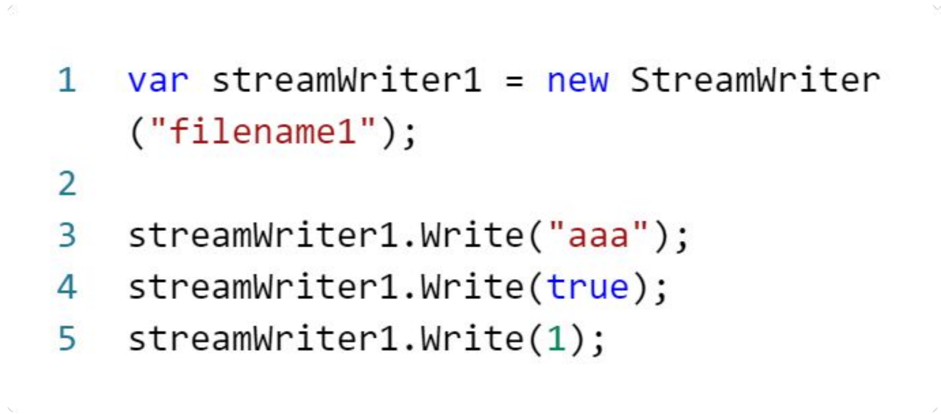
Method	**Description**
Close	Closes the stream reader and underlying stream
Peek	Returns the next character in the stream but does not move the character position
Read()	Returns the next character in the stream and moves the character position by one

Read(Char[], Int32, Int32)
ReadBlock(Char[], Int32, Int32)
	Reads the specified number of characters into the byte array

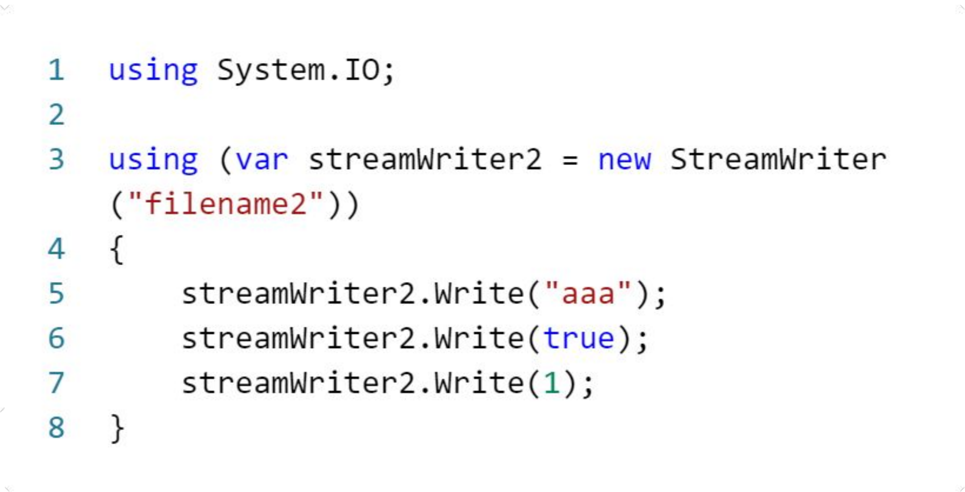
ReadLine	Reads a line of characters and returns a string
ReadToEnd	Reads all characters from the current position to the end of the file and returns a string

I/O: StreamWriter

- Main methods: **Write** / **WriteLine** / **Flush**
- Main prop: AutoFlush (default==false)
  - False – one must use **Flush** and **Close** methods.
  - True – every single **Write** actually writes to file



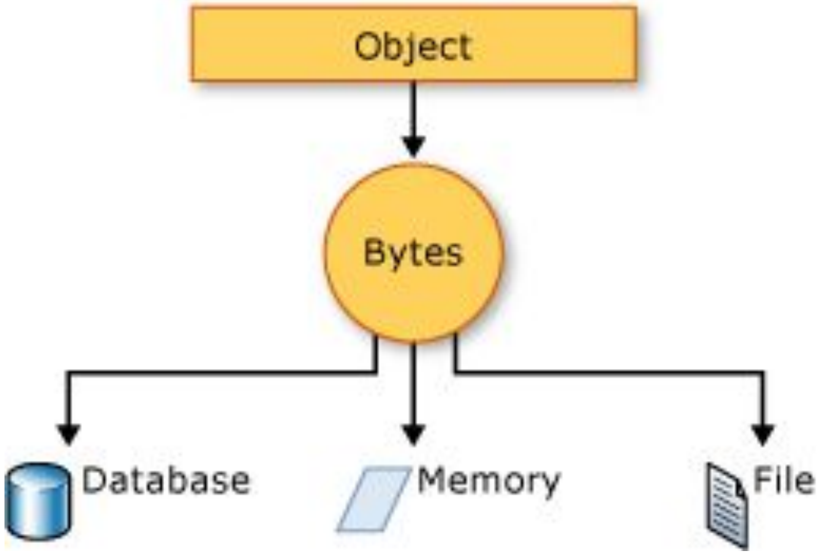
```
var streamWriter1 = new StreamWriter
("filename1");

streamWriter1.Write("aaa");
streamWriter1.Write(true);
streamWriter1.Write(1);
```



```
using System.IO;

using (var streamWriter2 = new StreamWriter
("filename2"))
{
    streamWriter2.Write("aaa");
    streamWriter2.Write(true);
    streamWriter2.Write(1);
}
```

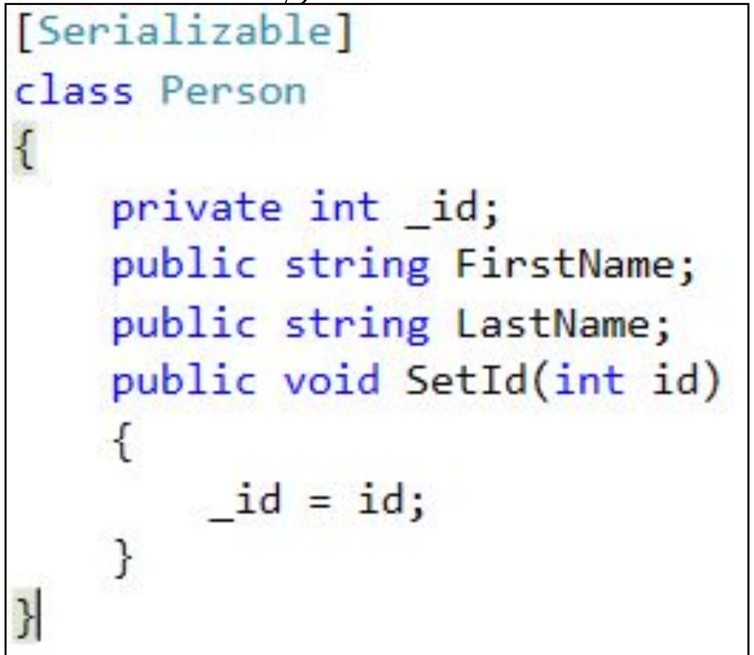
Serialization

- Serialization is a process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file.
- **Serialization** – converting object to specific format
- **Deserialization** – converting specific format back to object,
- Types:
  - Binary
  - XML
  - JSON
  - Custom



Binary serialization

- **<u>Considered to be obsolete since .NET 5.</u>**
- **System.Runtime.Serialization.Formatters.Binary**
- Two main methods: **Serialize** ir **Deserialize**.
- Used together with **FileStream** class objects.
  - In essence meaning that it's working with arrays of bytes.
- To be able to serialize in binary, attribute is a must:
  - [**Serializable**]

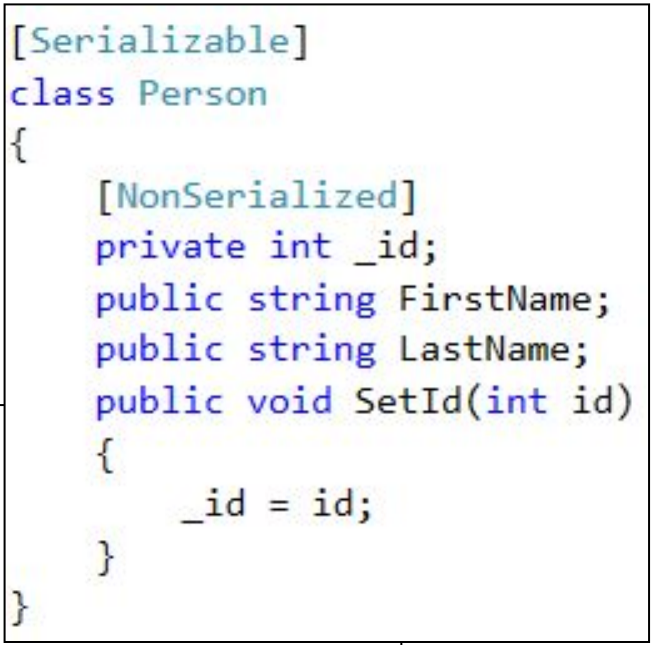


```
[Serializable]
class Person
{
    private int _id;
    public string FirstName;
    public string LastName;
    public void SetId(int id)
    {
        _id = id;
    }
}
```

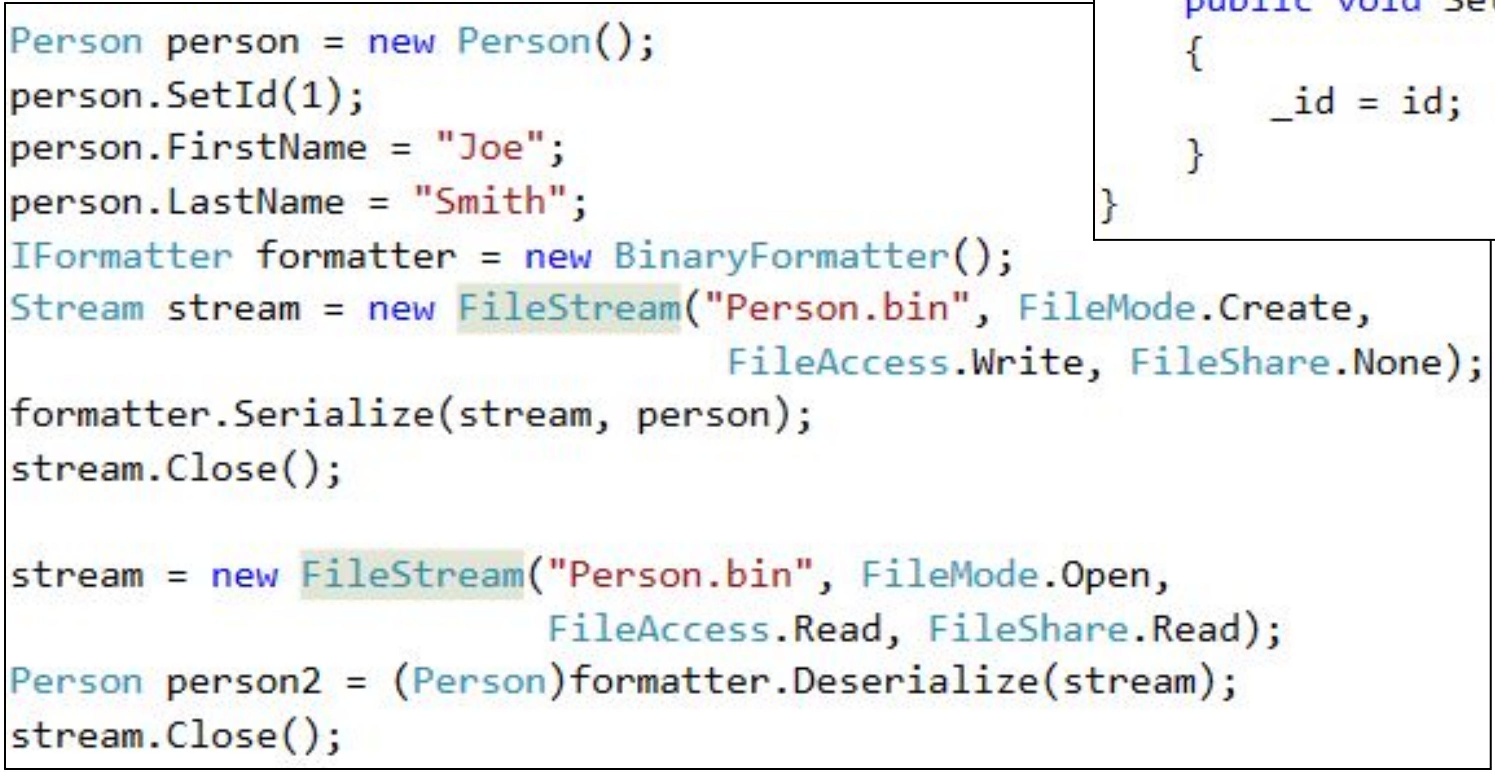
Binary serialization

- Saves private fields as well.
- To avoid saving privates:

  [**NonSerialized**]



```
[Serializable]
class Person
{
    [NonSerialized]
    private int _id;
    public string FirstName;
    public string LastName;
    public void SetId(int id)
    {
        _id = id;
    }
}
```



```
Person person = new Person();
person.SetId(1);
person.FirstName = "Joe";
person.LastName = "Smith";
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("Person.bin", FileMode.Create,
                                FileAccess.Write, FileShare.None);
formatter.Serialize(stream, person);
stream.Close();

stream = new FileStream("Person.bin", FileMode.Open,
                        FileAccess.Read, FileShare.Read);
Person person2 = (Person)formatter.Deserialize(stream);
stream.Close();
```

XML serialization

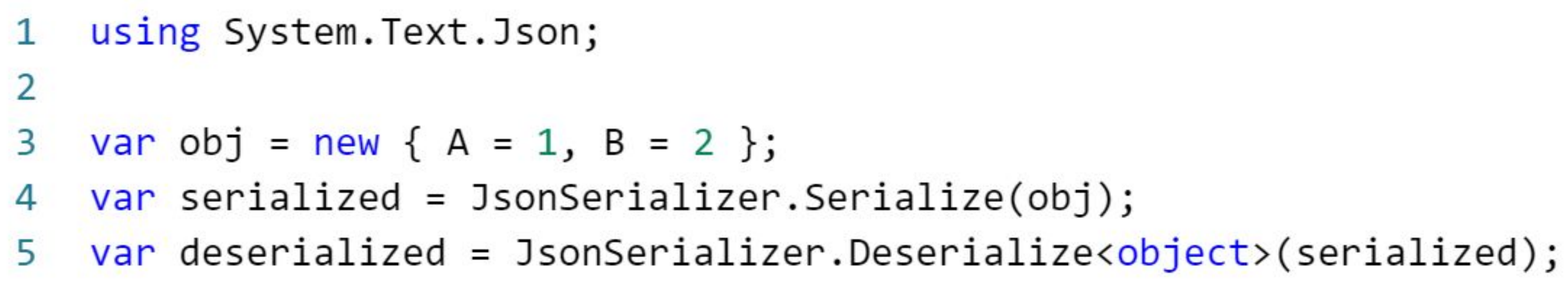
- Same as binary, only:
  - System.Xml.Serialization
  - Saves XML file
  - Saves only **public** fields and **public** classes
  - No need to add attribute

    [**Serializable**]
  - To avoid serialization an attribute is used:

    [**XmlIgnore**]

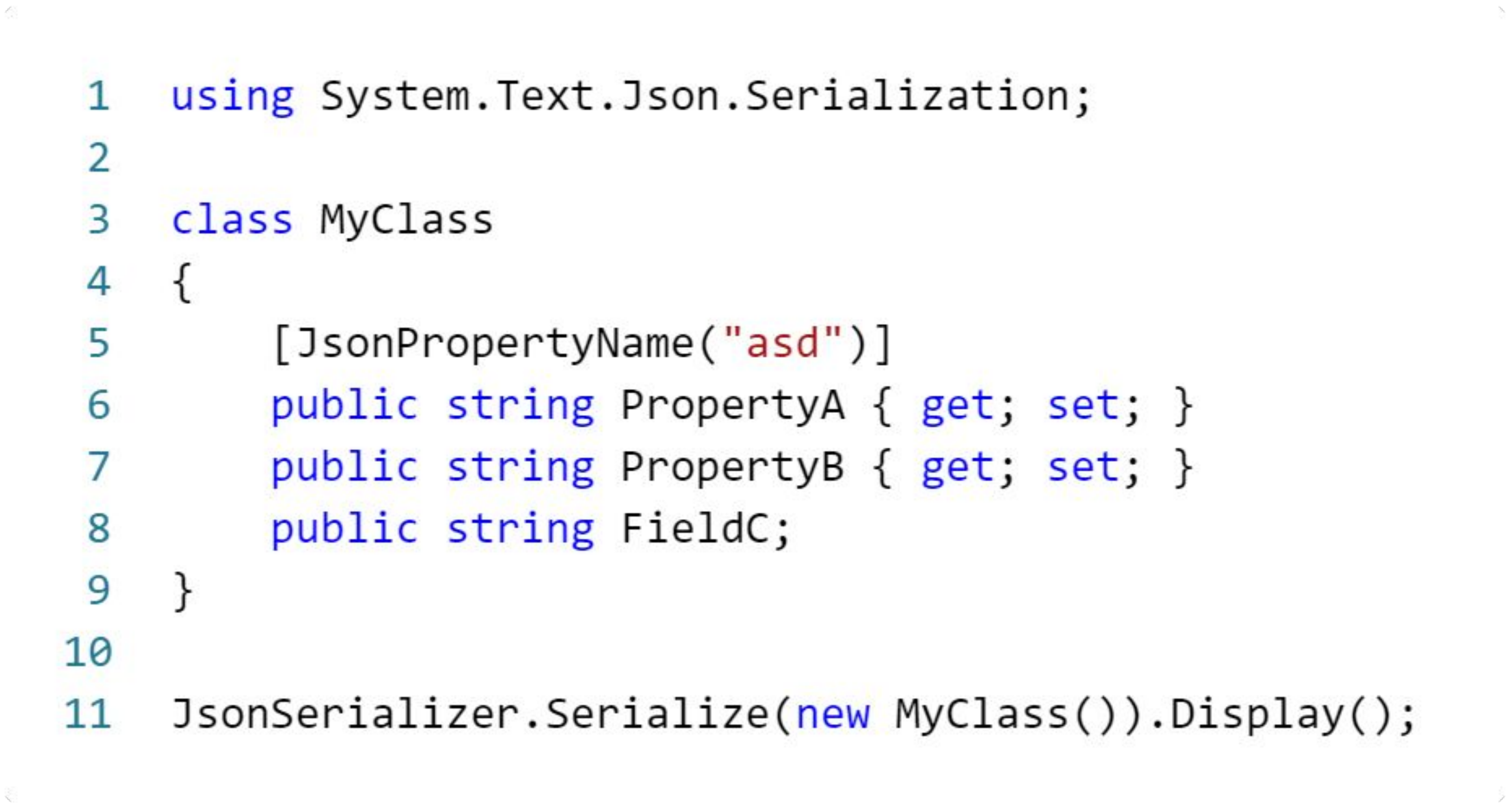
JSON serialization (1)

- Probably most popular serialization right now.
- 2 options:
  - using System.Text.Json; (personal recommendation)
  - using Newtonsoft.Json; (install from NuGet);
- When working with System.Text.Json:



```
using System.Text.Json;

var obj = new { A = 1, B = 2 };
var serialized = JsonSerializer.Serialize(obj);
var deserialized = JsonSerializer.Deserialize<object>(serialized);
```

JSON serialization (2)



```
using System.Text.Json.Serialization;

class MyClass
{
    [JsonPropertyName("asd")]
    public string PropertyA { get; set; }
    public string PropertyB { get; set; }
    public string FieldC;
}

JsonSerializer.Serialize(new MyClass()).Display();
```

JSON serialization (3)

```csharp
var serialized = JsonSerializer.Serialize(
    new MyClass {
    PropertyA = "A",
    PropertyB = "B",
    FieldC = "C",
});
serialized.Display();

var deserialized = JsonSerializer.Deserialize<MyClass>(serialized).Display();
```

```
{"asd":"A","PropertyB":"B"}
```

```
▼ Submission#8+MyClass
PropertyA   A
PropertyB   B
FieldC      <null>
```

Mistakes, errors: how to handle?

- **Normal** desire – block mistakes ASAP.
- Checking for mistakes explicitly:
  - Mistake in method gets detected
  - Error code returned via out parameter
- Advantage of such a code:
  - Easily readable error flow
  - 100% control of what's happening

Typical read from file (C lang)

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;        }
                } else { errorCode = -2;  }
        } else { errorCode = -3; }
```

```
close the file;
        if (theFileDidntClose
           && errorCode == 0) {
            errorCode = -4;}
        else {
            errorCode =
                  errorCode & -4;   }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Errors

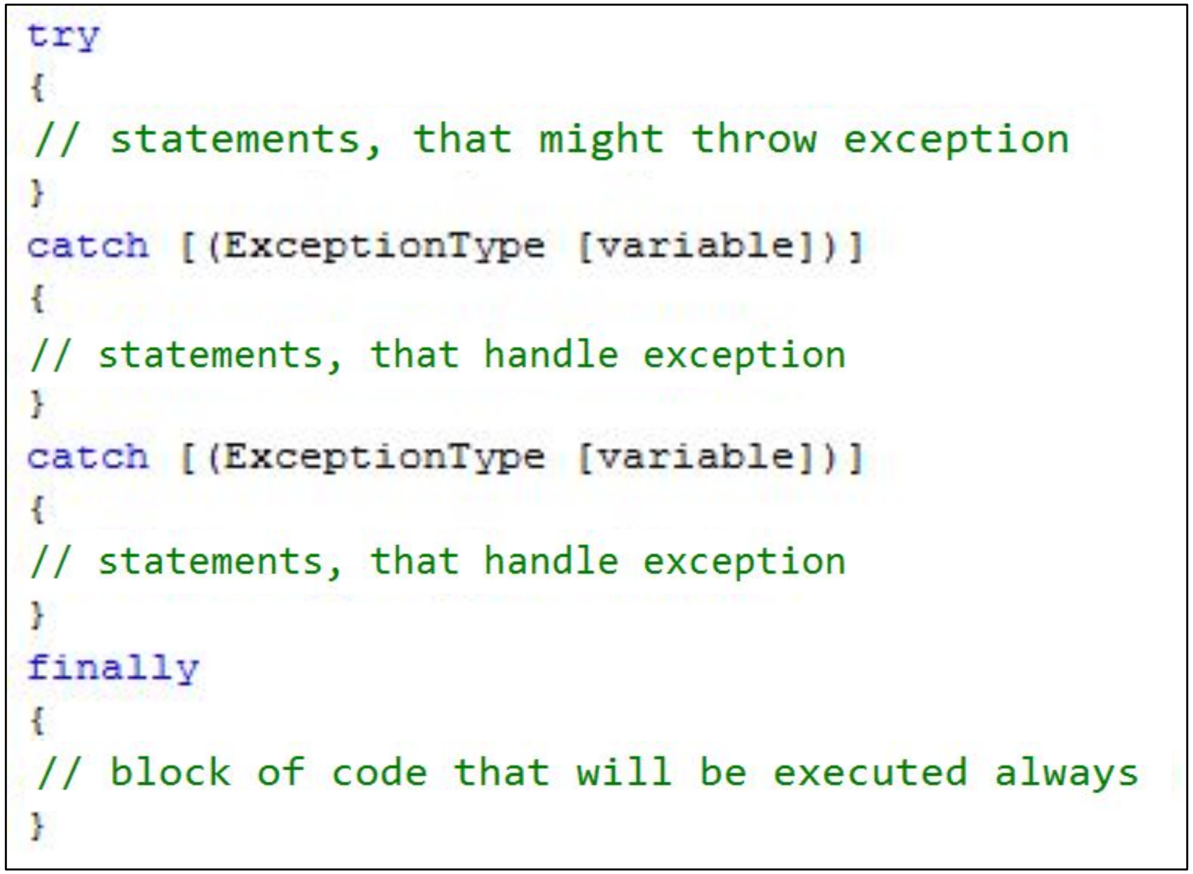
- Disadvantages:
  - Error checking is mixed with business knowledge
  - still not clear, if all errors are handled
  - code is harder:
    - readable
    - modifiable
- Each case is different, though:
  - Sometimes explicit handling is better
  - Sometimes it's better for error to happen and handle it later

Error vs exception

- There are cases, when it is impossible to pre-determine the error. For example:
  - File being sent over network,
  - File damaged while reading it,
  - Zero memory left,
  - No hard disc space
- Solution: **try-catch-finally** block

Exceptions

- **Try** block – mandatory,
- **Catch/Finally** – mandatory only one of them. No code inside is mandatory.
- Syntax:



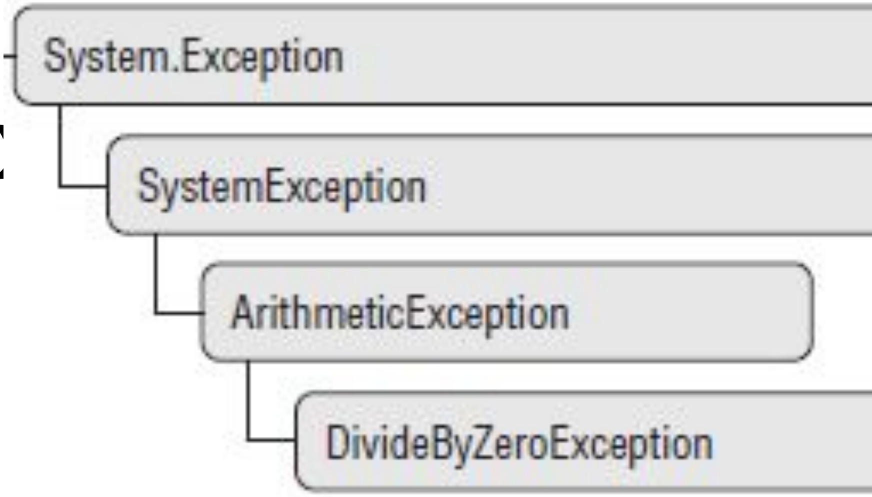
```
try
{
// statements, that might throw exception
}
catch [(ExceptionType [variable])]
{
// statements, that handle exception
}
catch [(ExceptionType [variable])]
{
// statements, that handle exception
}
finally
{
// block of code that will be executed always
}
```

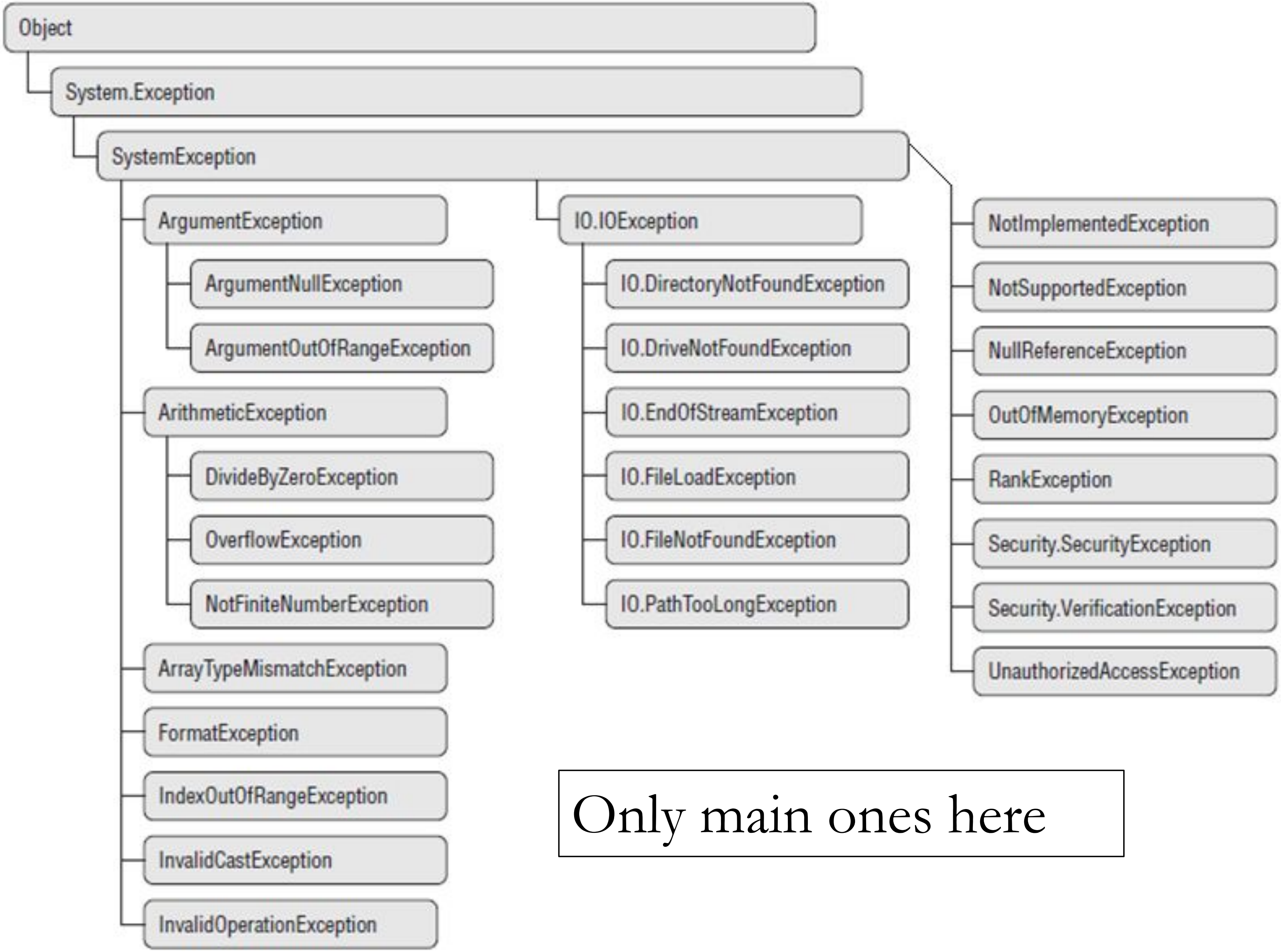
Exception handling

- Program checks all **catch** blocks one by one.
- Error that happened, is compared with catch block
- When match is found:
  1. code is executed in **catch** block
  2. other **catch** blocks are not processed further.
  3. If there is – **final** block is executed.
- Golden rules:
  - Deal only with exceptions you can do anything meaningful
  - In most cases nothing meaningful can be done

Exception handling order

- Error is matched with exception, when it can be compared to exception or it's parents.
- Example: **DivideByZeroException** is caught by any catch block, described above in the hierarchy

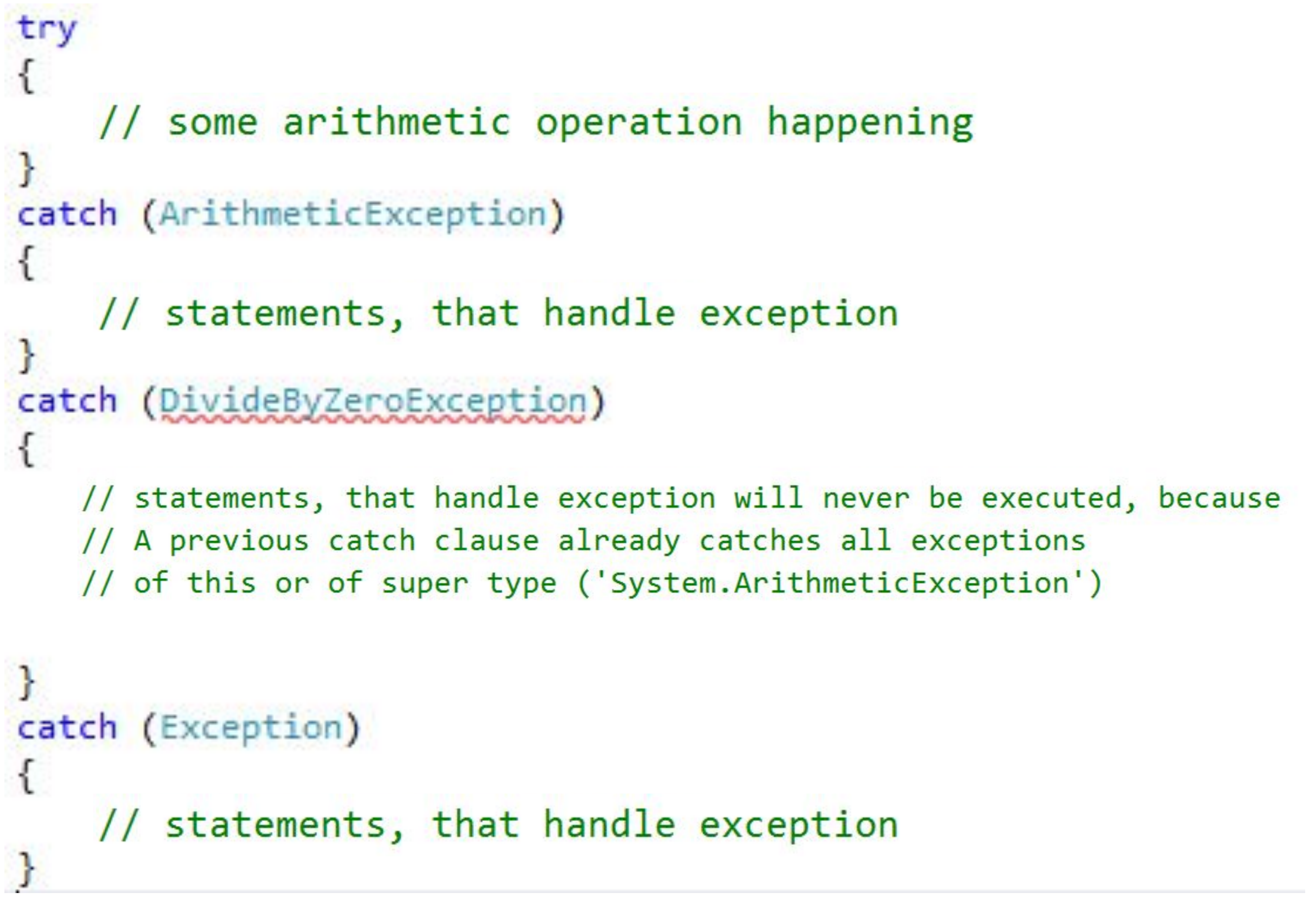




Only main ones here

Bad example

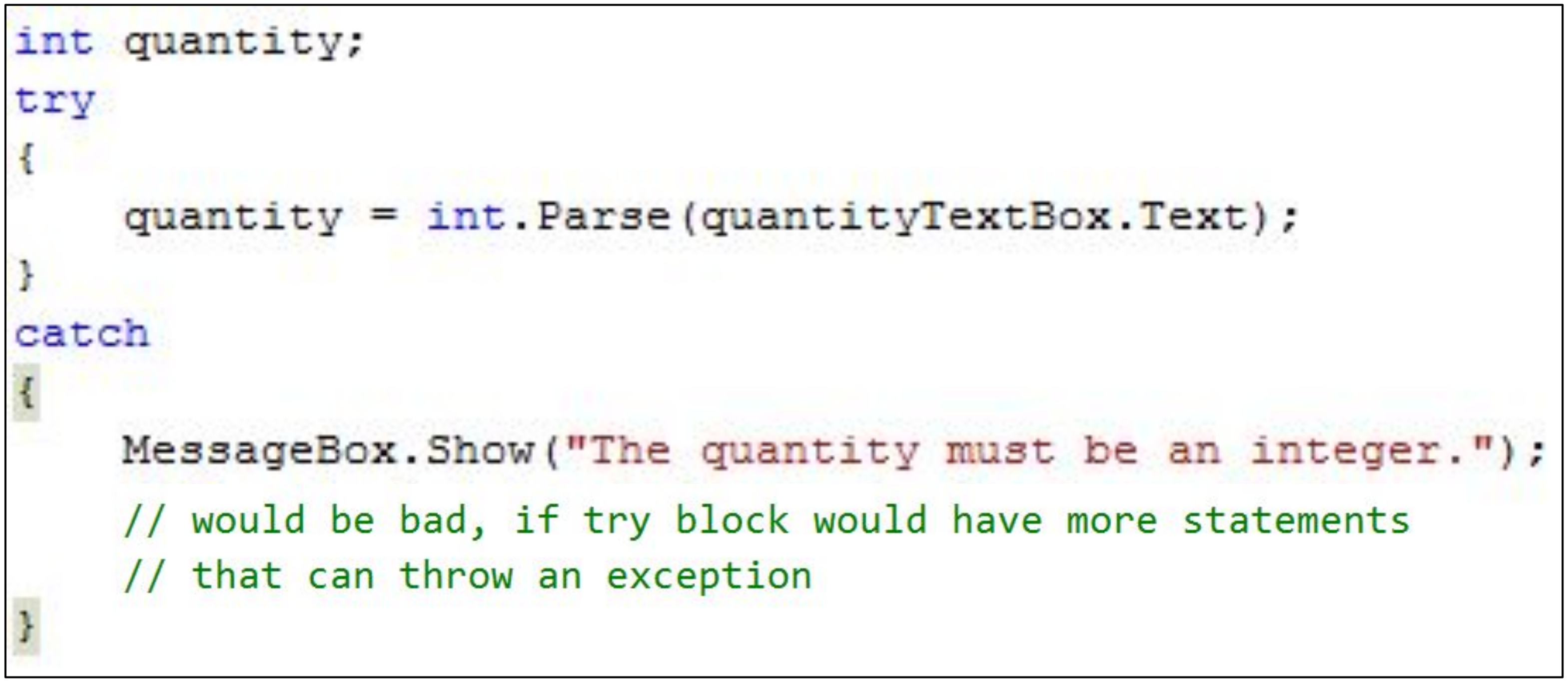
- Compiler helps, though:



```
try
{
    // some arithmetic operation happening
}
catch (ArithmeticException)
{
    // statements, that handle exception
}
catch (DivideByZeroException)
{
   // statements, that handle exception will never be executed, because
   // A previous catch clause already catches all exceptions
   // of this or of super type ('System.ArithmeticException')

}
catch (Exception)
{
    // statements, that handle exception
}
```

Passing of information

- **catch** block is allowed not to catch any exception, but then it works as System.Exception, only without extra inf.



```
int quantity;
try
{
    quantity = int.Parse(quantityTextBox.Text);
}
catch
{
    MessageBox.Show("The quantity must be an integer.");
    // would be bad, if try block would have more statements
    // that can throw an exception
}
```

Passing of information

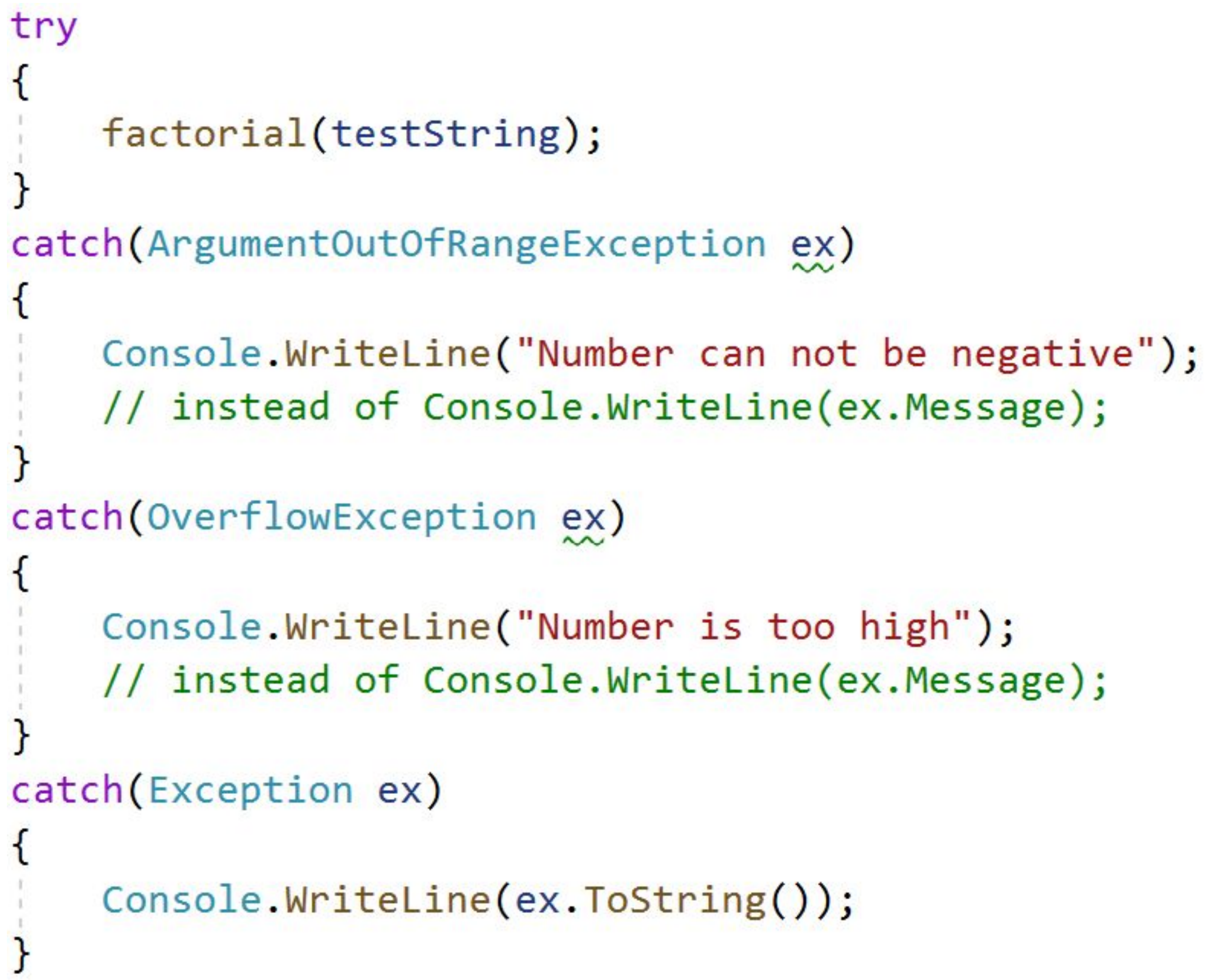
- Information passed through **Exception** (or inheriting) class variable is given in its properties:

Property	**Description**
Data	A collection of key/value pairs that give extra information about the exception.
InnerException	An Exception object that gives more information about the exception. Some libraries catch exceptions and wrap them in new exception objects to provide information that is more relevant to the library. In that case, they may include a reference to the original exception in the InnerException property.
Message	A message describing the exception in general terms. Not user-friendly all the time.
Source	The name of the application or object that caused the exception.
StackTrace	A string representation of the program's stack trace when the exception occurred.
TargetSite	Information about the method that threw the exception.

- **ToString()** returns Message + StackTrace

Passing of information

- Typically you don’t want to pass **.Message** to user:

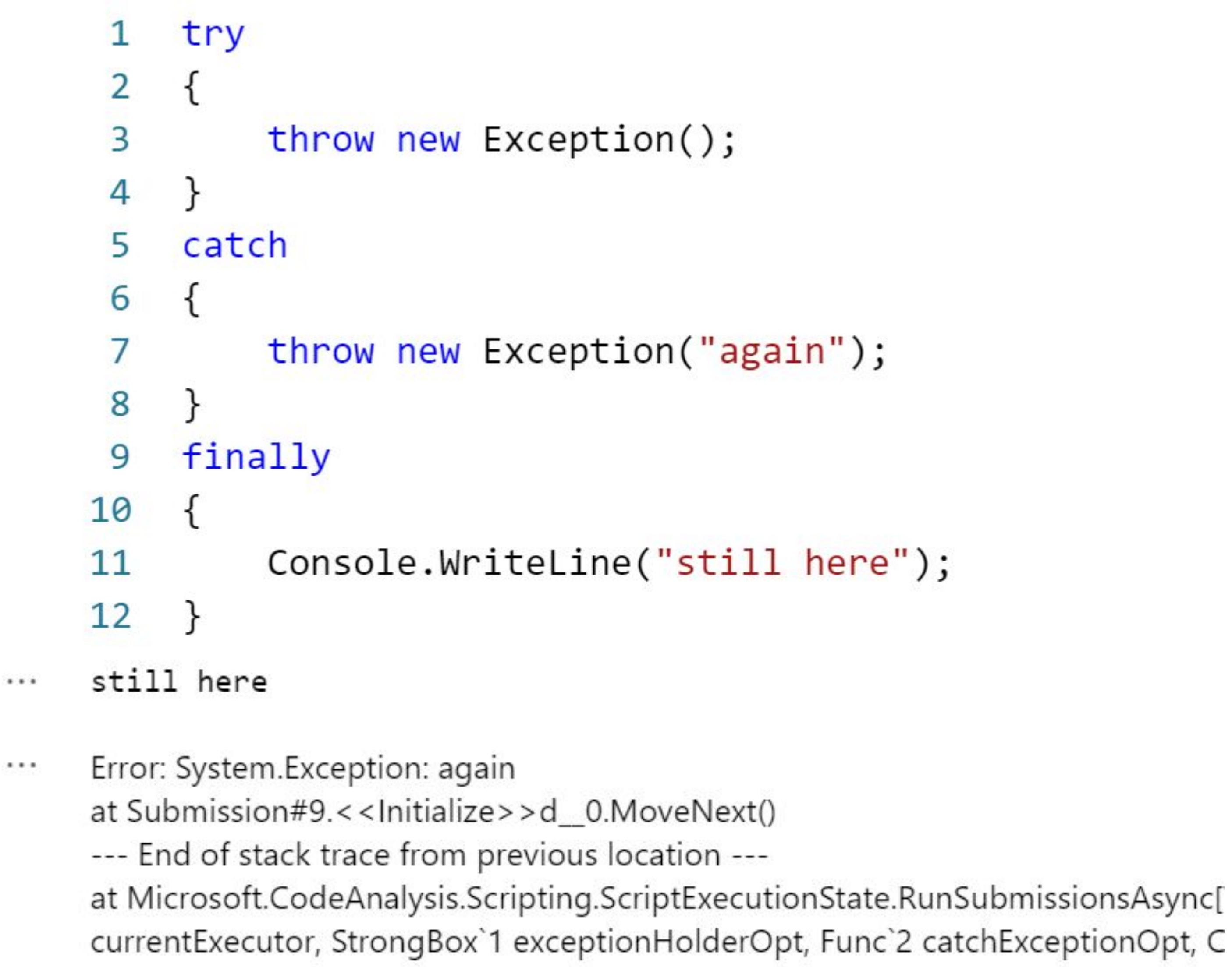


```
try
{
    factorial(testString);
}
catch(ArgumentOutOfRangeException ex)
{
    Console.WriteLine("Number can not be negative");
    // instead of Console.WriteLine(ex.Message);
}
catch(OverflowException ex)
{
    Console.WriteLine("Number is too high");
    // instead of Console.WriteLine(ex.Message);
}
catch(Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

Exception handling: finally

- **Finally** block works all the time:
  - No error happened and no catch fired
  - Error happened, and **one** of catch'es fired
  - Error happened, and **none** of catch'es fired
  - Try had return inside of it
  - Catch had return inside of it
  - Catch block fired new exception

Exceptions: finally



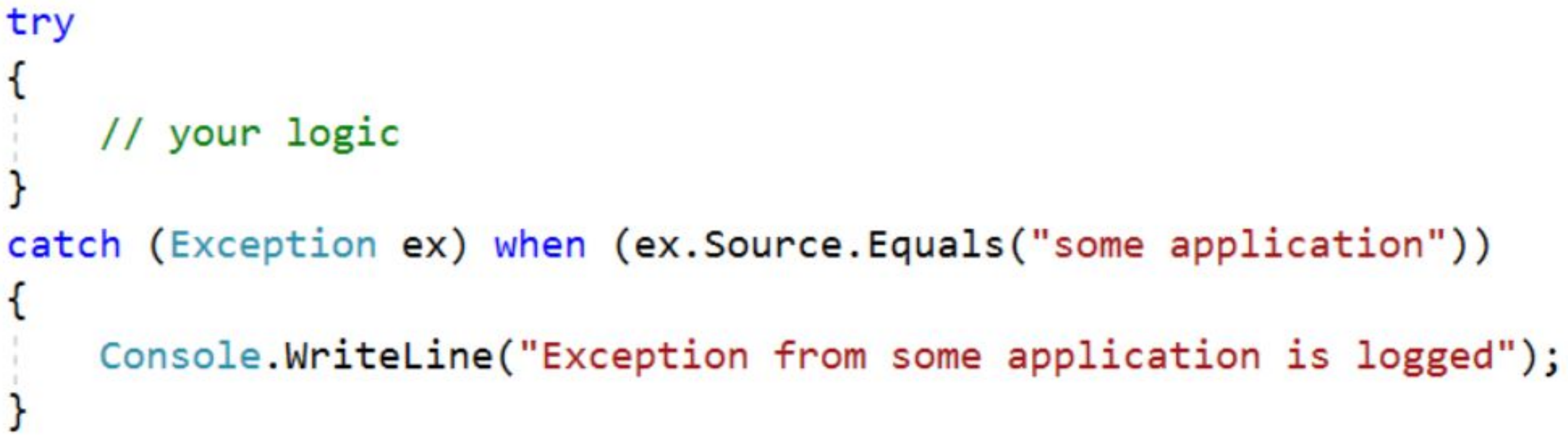
```csharp
try
{
    throw new Exception();
}
catch
{
    throw new Exception("again");
}
finally
{
    Console.WriteLine("still here");
}
```

```
still here
```

```
Error: System.Exception: again
at Submission#9.<<Initialize>>d__0.MoveNext()
--- End of stack trace from previous location ---
at Microsoft.CodeAnalysis.Scripting.ScriptExecutionState.RunSubmissionsAsync[
currentExecutor, StrongBox`1 exceptionHolderOpt, Func`2 catchExceptionOpt, C
```

Exceptions filter (from c# 6.0)

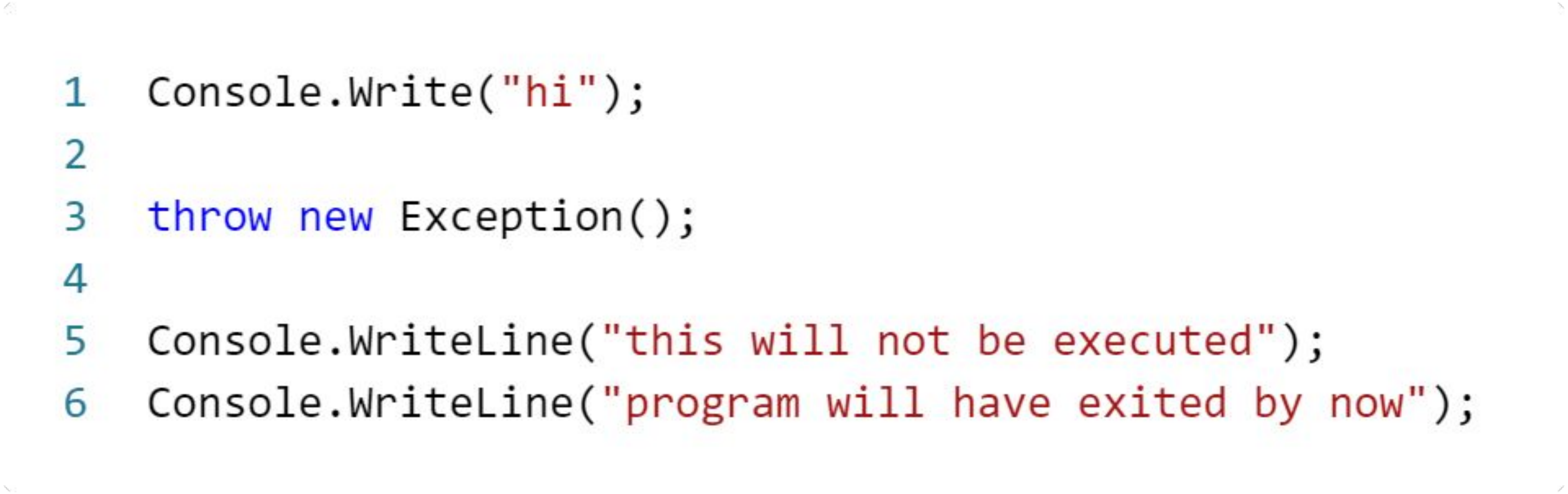
- Exception will enter catch block **only if** condition specified in **when** returns **true.**
- If exception does not enter the catch block, stack trace is not being unwound – which means that your program will preserve information about the initial method that threw the exception.



```
try
{
    // your logic
}
catch (Exception ex) when (ex.Source.Equals("some application"))
{
    Console.WriteLine("Exception from some application is logged");
}
```

Unhandled exceptions

- Messages are propagated up the call stack.
- If the exception reaches entry point of main thread and it is still not caught - the program will halt executing and exit with error code.



```
Console.Write("hi");

throw new Exception();

Console.WriteLine("this will not be executed");
Console.WriteLine("program will have exited by now");
```

Passing exception info

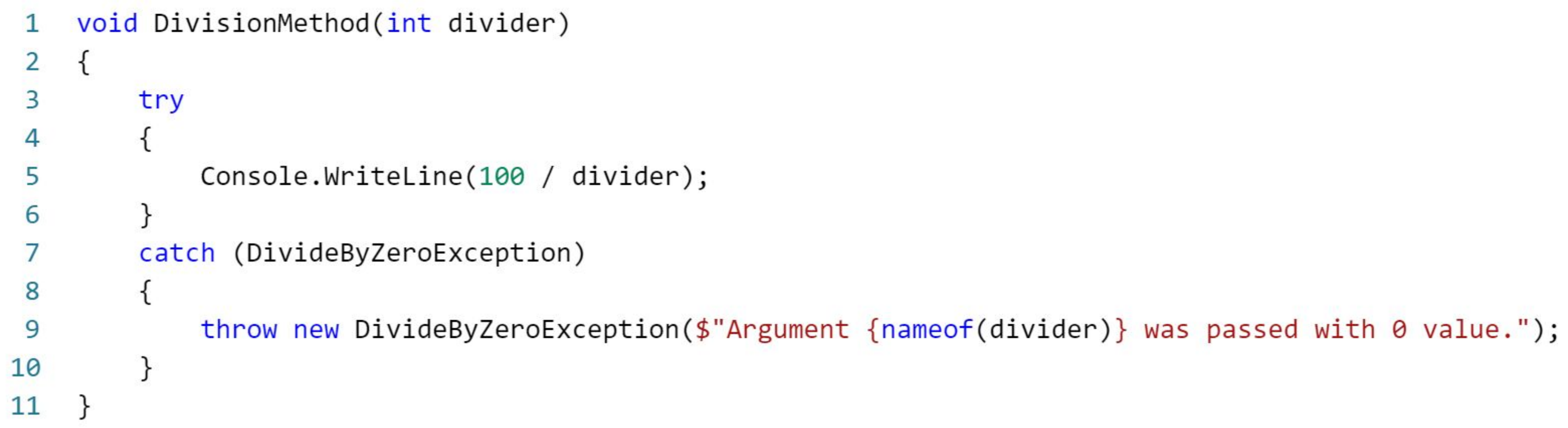
- It is NOT good to do outputting to user directly when the exception is caught.
- Usually, a special class/library/package/the framework itself is used for that, and that lib does not have anything to do with the UI
- Keeping it SOLID :)
- AOP could be used (Aspect Oriented Programming).

Passing exception info

- If having such a library, the code that uses it, decides which exceptions:
  - To catch
  - To ignore
  - To rethrow
- Catching and not doing anything about it... is worst!
- Different from Java – no forced exception checking

Passing exception info

- If possible to determine, why error happened, while handling the exception extra info should be added to rethrown exception

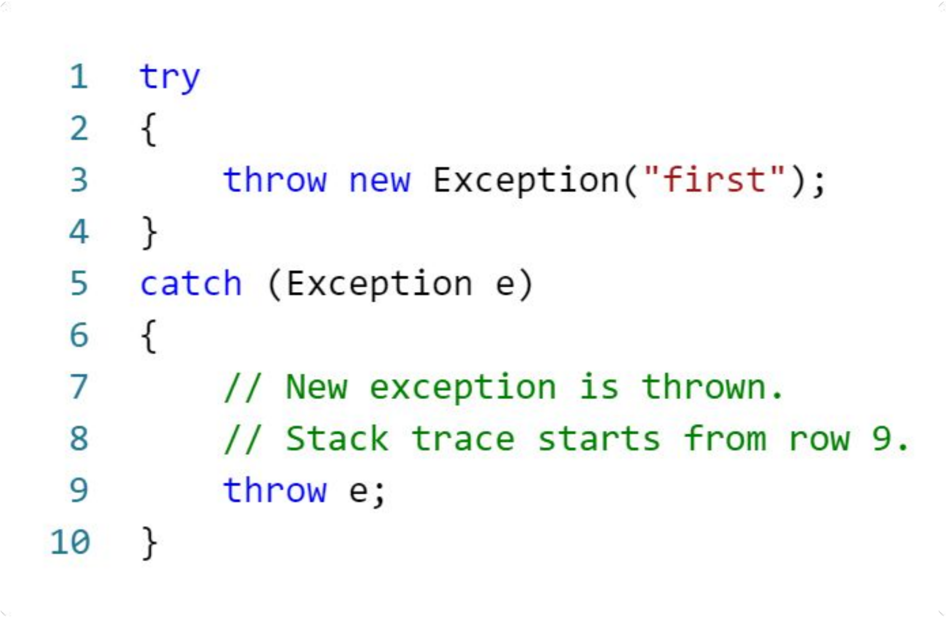


```csharp
void DivisionMethod(int divider)
{
    try
    {
        Console.WriteLine(100 / divider);
    }
    catch (DivideByZeroException)
    {
        throw new DivideByZeroException($"Argument {nameof(divider)} was passed with 0 value.");
    }
}
```

- If method cannot add extra info (or do anything useful), highly possible, than this method should not catch that exception (just to bubble it up).

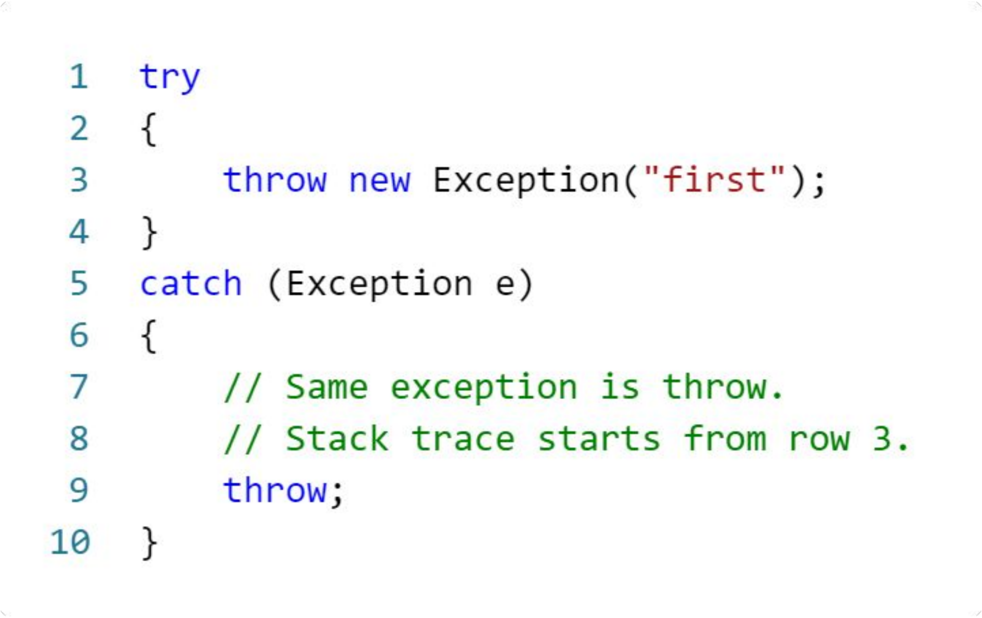
re-throwing

Not very good



```
 1  try
 2  {
 3      throw new Exception("first");
 4  }
 5  catch (Exception e)
 6  {
 7      // New exception is thrown.
 8      // Stack trace starts from row 9.
 9      throw e;
10  }
```

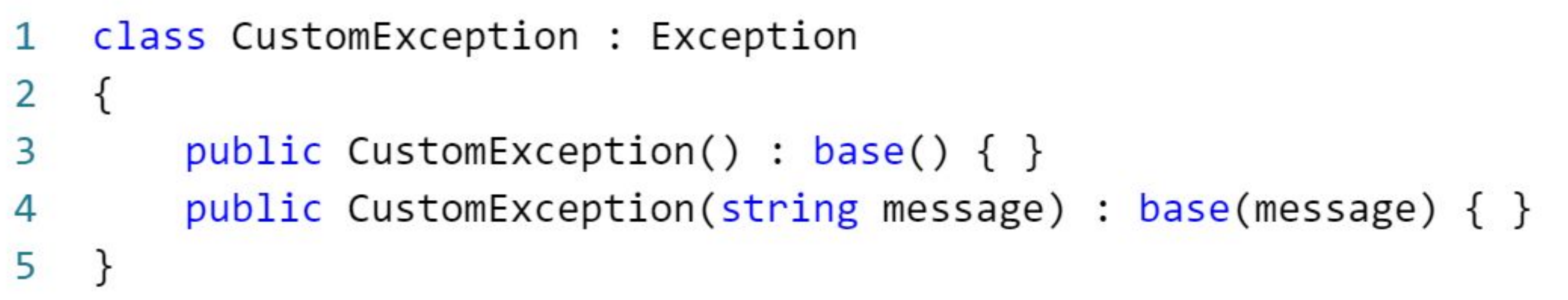
Good



```
 1  try
 2  {
 3      throw new Exception("first");
 4  }
 5  catch (Exception e)
 6  {
 7      // Same exception is throw.
 8      // Stack trace starts from row 3.
 9      throw;
10  }
```

Creating custom exception class

- It is recommended to use existing ones.
- If really needed, inheritance happens from Exception.
  - In the past it **was** suggested to inherit ApplicationException when creating custom exceptions, however this is **no longer the case**.
- Name has **Exception** postfix



```
class CustomException : Exception
{
    public CustomException() : base() { }
    public CustomException(string message) : base(message) { }
}
```

Exceptions: practical advices

- If **catch** (or **finally)** has some risky code inside, inner **try-catch-finally** can be used, but separate method is advised.

- How to catch all exceptions:
  - Code segment is created, catching all exceptions.
  - Last catch is generic: System.Exception
  - Program is tested with bad data
  - When entering **catch** (Exception ex) a specific exception is caught and then new code catching it is added.

AOP – Aspect Oriented Programming

- "In computing, aspect-oriented programming (AOP) is a programming paradigm that aims to **increase modularity** by allowing the separation of **cross-cutting concerns**. It does so by adding additional behavior to existing code **without modifying** the code itself …" – Wikipedia.
- "Aspect - The combination of the pointcut and the advice is termed an aspect. In the example above, we add a logging aspect to our application by defining a pointcut and giving the correct advice." – Wikipedia.
  - Can be roughly interpreted as "what code and where to insert".
- SOLID - strongly related to single responsibility principle.

Interceptor

- Works as a proxy – intercepts a call to a method before it happens and after it returns.
- Allows to do cross-cutting functionality in one place.
- In C# can be used as an attribute or injected using dependency injection container.
- Have to use Castle.Core (most popular) or other nuget that provides interceptor functionality.

How aspect would look like

```csharp
public class LogAttributeInterceptor : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        // Check if the method is marked with the LogAttribute
        if (invocation.Method.GetCustomAttributes(typeof(LogAttribute), true).Length > 0)
        {
            try
            {
                invocation.Proceed();
            }
            catch
            {
                Console.WriteLine("Exception occured");
            }
        }
    }
}

```

Caveats of these interceptors

- Interceptor is aware of attributes and check for them:

```csharp
if (invocation.Method.GetCustomAttributes(typeof(LogAttribute), true).Length > 0)
```

- Intercepted methods must be *virtual*:

```csharp
public virtual void MethodToIntercept()
```

- Classes have to be instantiated via proxy:

```csharp
var generator = new ProxyGenerator();
var interceptor = new LogAttributeInterceptor();

var service = generator.CreateClassProxy<MyService>(interceptor);
```

Benefits of using interceptors for handling exceptions

- Exception handling logic is written in one place, so if you need to change your logging provider – you will be able to do it in one place.
- Provides a unified log – there is one logging style.
- Separates logging concern from the class.

Delegates (1)

- A delegate is a data type much as a class or structure is.
- Class and struct define the properties, methods, and events provided by a class or structure.
- In contrast, a delegate is a type that defines the parameters and return value of a method.
- Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object.
- You can think of delegate as type declaration, but for methods only.

Delegates (2)

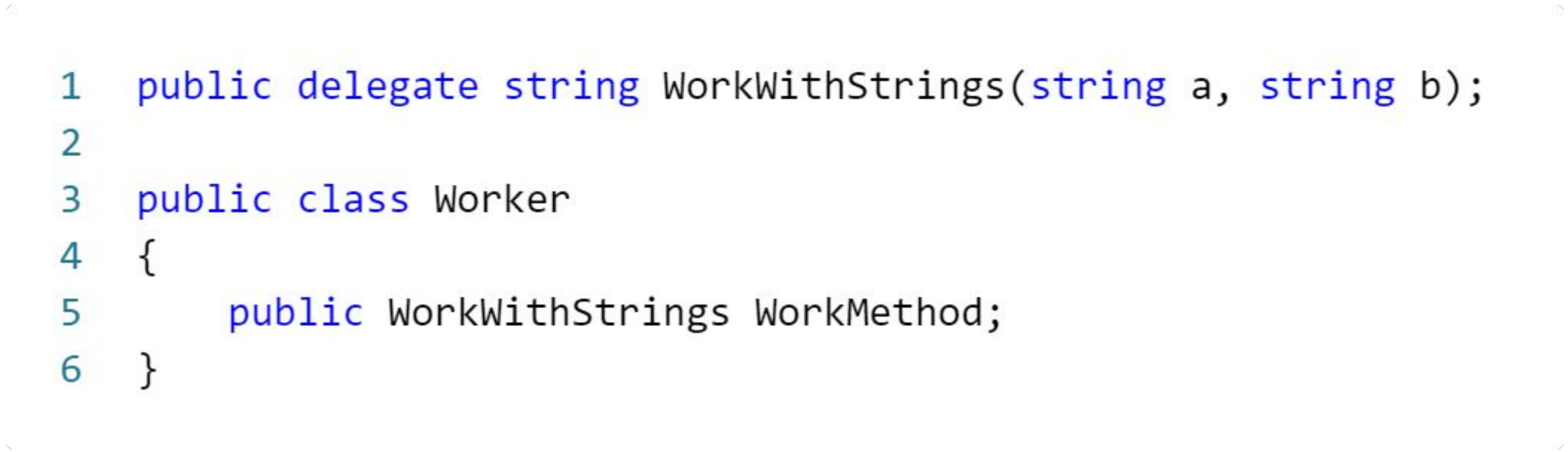
- Description:

[accessibility] delegate returnType DelegateName([parameters]);

  - **accessibility**: for example **public** or **private**
  - **delegate**: mandatory keyword.
  - **returnType**: what method is returning.
  - **delegateName**: type name
  - **parameters**: The parameter list that a method of this delegate type should take.

Delegates

- Can be declared in method, in class and outside class:

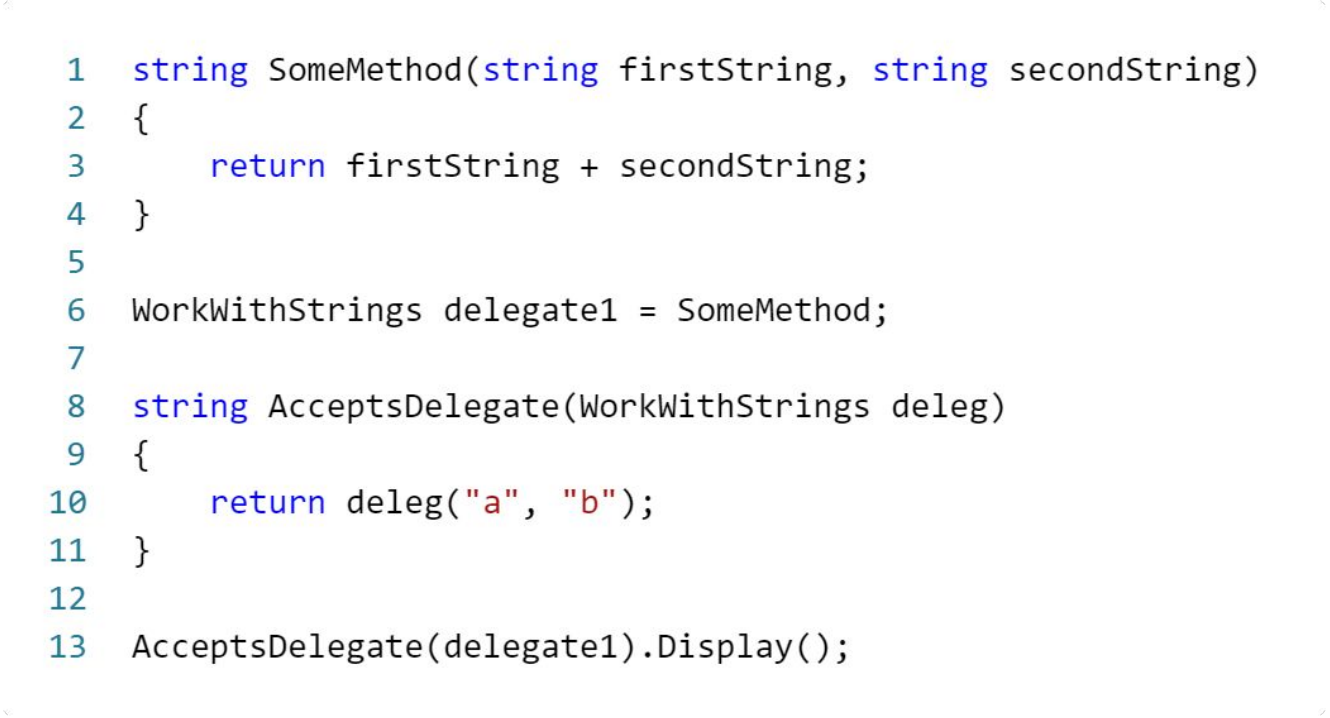


```
public delegate string WorkWithStrings(string a, string b);

public class Worker
{
    public WorkWithStrings WorkMethod;
}
```

- **WorkWithStrings** delegate **matches** all methods that take **2 strings as arguments** and **returns a string** as a return type.

Creating variables

- Delegates can be used as variable types.
- Methods/actions/functions matching delegate pattern can be assigned to these variables.



```
string SomeMethod(string firstString, string secondString)
{
    return firstString + secondString;
}

WorkWithStrings delegate1 = SomeMethod;

string AcceptsDelegate(WorkWithStrings deleg)
{
    return deleg("a", "b");
}

AcceptsDelegate(delegate1).Display();
```

Working with variables

```
WorkWithStrings addString = (string a, string b) => a + b;
WorkWithStrings returnFirst = (string a, string b) => a;

var worker1 = new Worker
{
    WorkMethod = addString,
};

var worker2 = new Worker
{
    WorkMethod = returnFirst,
};

worker1.DoWork("a", "b").Display();
worker2.DoWork("a", "b").Display();
```

Displays:

```
ab
a
```

Multicast delegates

- Multiple delegates can be added with + operator. This is called ***multicasting***.
- They work on FIFO order on variables.
- Right-most delegate value is returned.

```
WorkWithStrings multicast1 = (a, b) => { Console.WriteLine("1"); return "1"; };
WorkWithStrings multicast2 = (a, b) => { Console.WriteLine("2"); return "2"; };

var multicastWorker = new Worker
{
    WorkMethod = multicast1 + multicast2,
};

Console.WriteLine(multicastWorker.DoWork("a", "b"));
```

Multicast delegate – return value

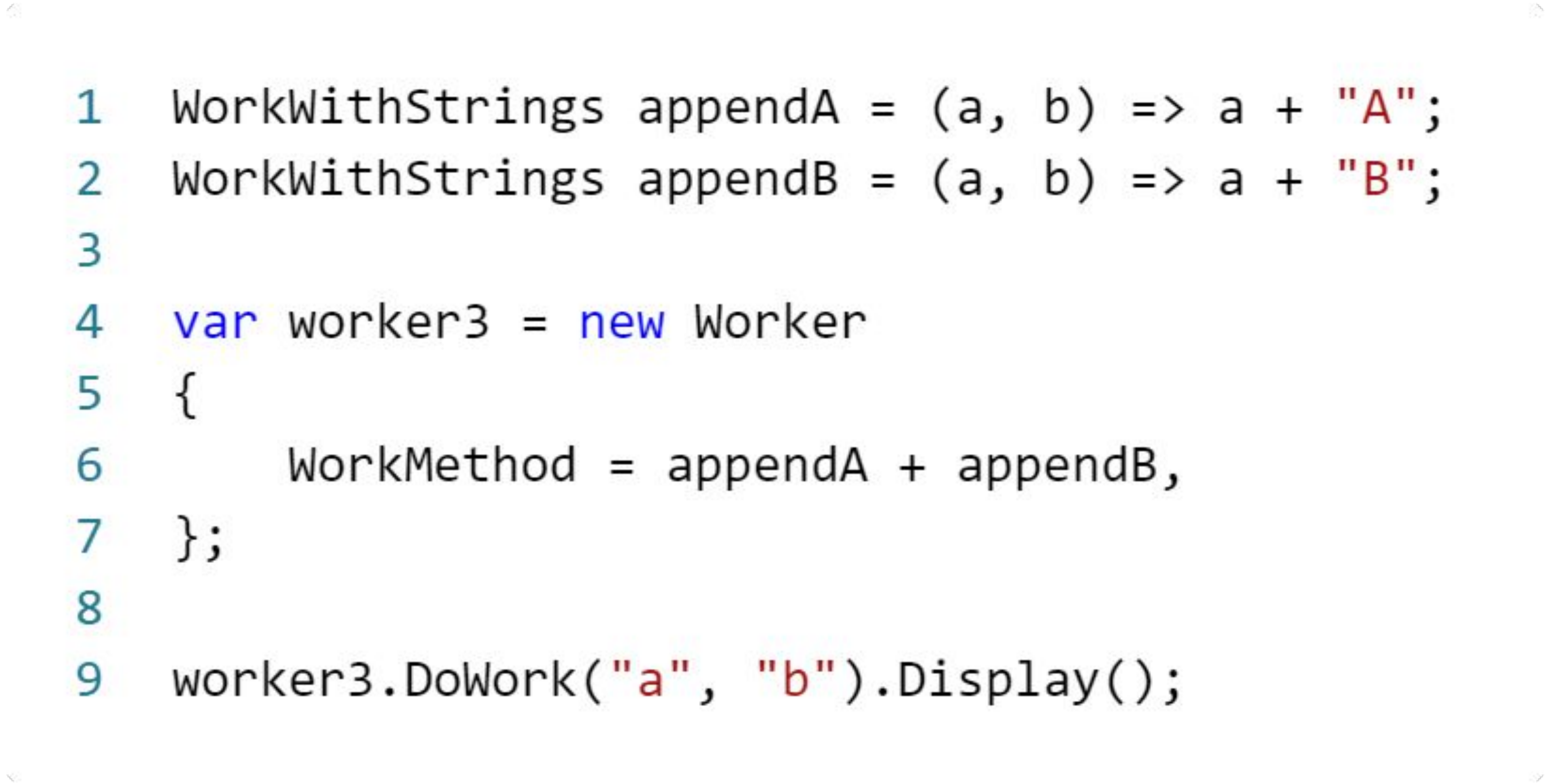
- If you invoke multicast delegate and assign it to variable, only the last delegate value is stored in variable;
- If you want to use all delegates return value from multicast delegate – GetInvocationList() method has to be used in a foreach cycle;
- Most of the times multicast delegate will be used where return value is not needed, but actions in particular order needs to be executed;

Reasonable example

```
delegate Person UpdatePerson(Person person);

class Person
{
    public string Name { get; set; }
}

class Updater
{
    public UpdatePerson Updates;

    public void DoUpdates(Person person)
    {
        Updates(person);
    }
}

UpdatePerson addTitle = (p) => { p.Name = "Sir " + p.Name; return p; };
UpdatePerson addLastName = (p) => { p.Name = p.Name + " Namington"; return p; };

var updater = new Updater { Updates = addTitle + addLastName, };
var person = new Person { Name = "Guy", };

updater.DoUpdates(person);
person.Name.Display();
```

Exam like question

⬇️ what does it display?



```
WorkWithStrings appendA = (a, b) => a + "A";
WorkWithStrings appendB = (a, b) => a + "B";

var worker3 = new Worker
{
    WorkMethod = appendA + appendB,
};

worker3.DoWork("a", "b").Display();
```

Working with multicasts

- Addition and subtraction works based on list addition and subtraction:
  - When adding – in the end.
  - When subtracting – last instance found is subtracted.
  - When no instance is found, same value is kept.

Expression	Result
null + d1	d1
d1 + null	d1
d1 + d2	[d1, d2]
d1 + [d2, d3]	[d1, d2, d3]
[d1, d2] + [d2, d3]	[d1, d2, d2, d3]
[d1, d2] - d1	d2
[d1, d2] - d2	d1
[d1, d2, d1] - d1	[d1, d2]
[d1, d2, d3] - [d1, d2]	d3
[d1, d2, d3] - [d2, d1]	[d1, d2, d3]
[d1, d2, d3, d1, d2] - [d1, d2]	[d1, d2, d3]
[d1, d2] - [d1, d2]	null

Generic delegates

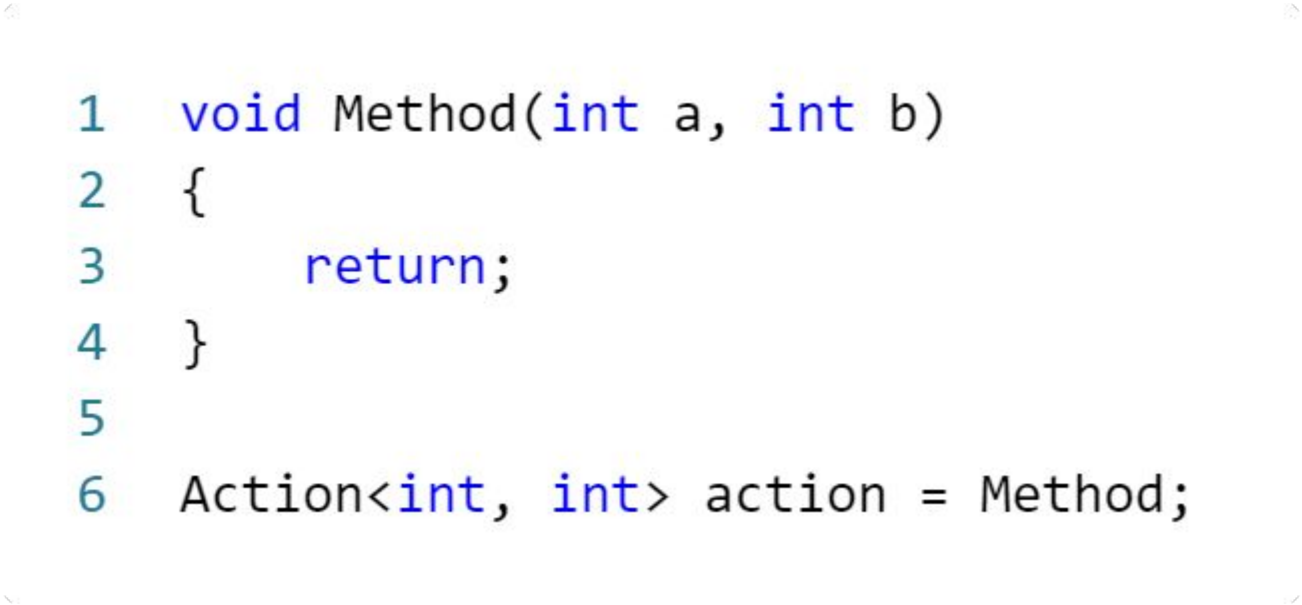
- Example:

```
// Not a very meaningful example
public delegate T2 GenericDelegate<T1, T2>(T1 t1);

int Method(string input)
{
    return 0;
}

GenericDelegate<string, int> del = Method;
```

- In general it is advised to use those only in generic classes

Action

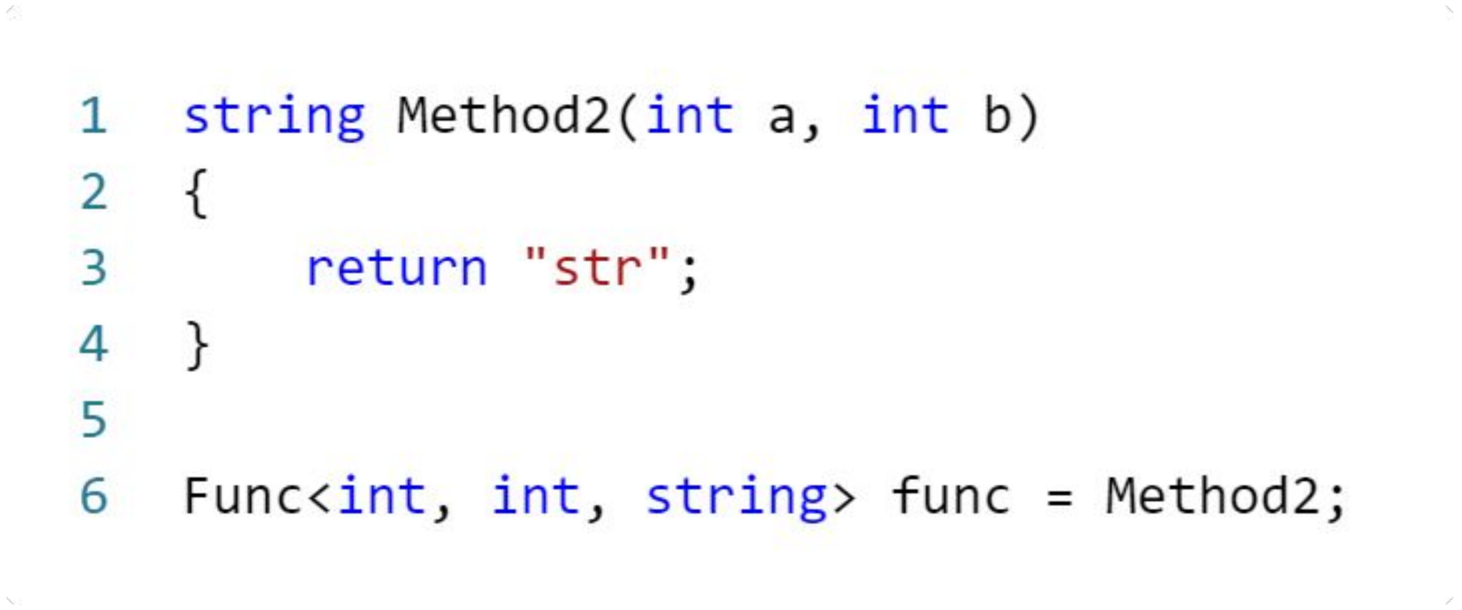
- **Action** is parameterized delegate and matches methods that return void.
- Generic arguments match method argument types.



```
void Method(int a, int b)
{
    return;
}

Action<int, int> action = Method;
```

Standard delegates: **Func**

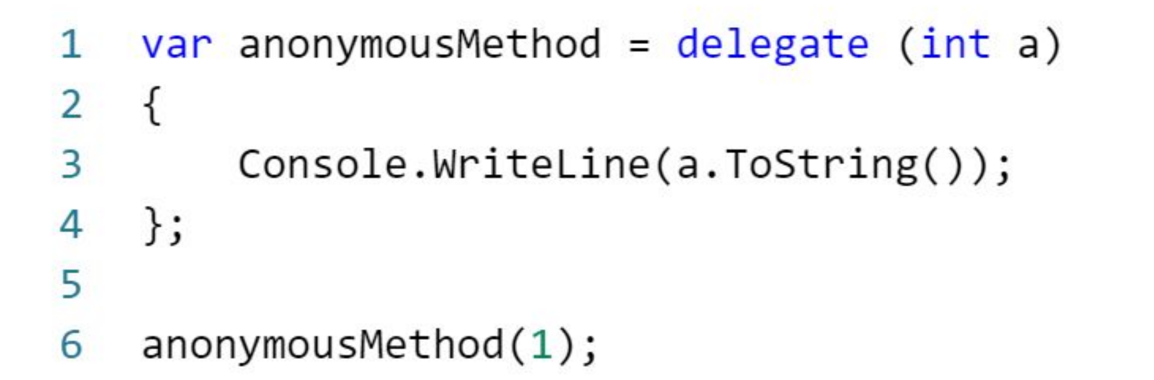
- **Func** is parameterized delegate, which matches methods that return some value.
- Last generic argument is the return type. Initial generic arguments (until the last one) are the function input arguments.



```
string Method2(int a, int b)
{
    return "str";
}

Func<int, int, string> func = Method2;
```

Anonymous methods

- Method, which does not have a name
- Instead of creating method, a delegate is created, which points to peace of the code.
- Syntax, which allows to assign to delegate variable:
- **delegate([params]) {code... }**
  - **delegate**: mandatory
  - **params**: params to pass
  - **code**: any code. Including with "return" inside of it"



```
var anonymousMethod = delegate (int a)
{
    Console.WriteLine(a.ToString());
};

anonymousMethod(1);
```

Lambda expressions

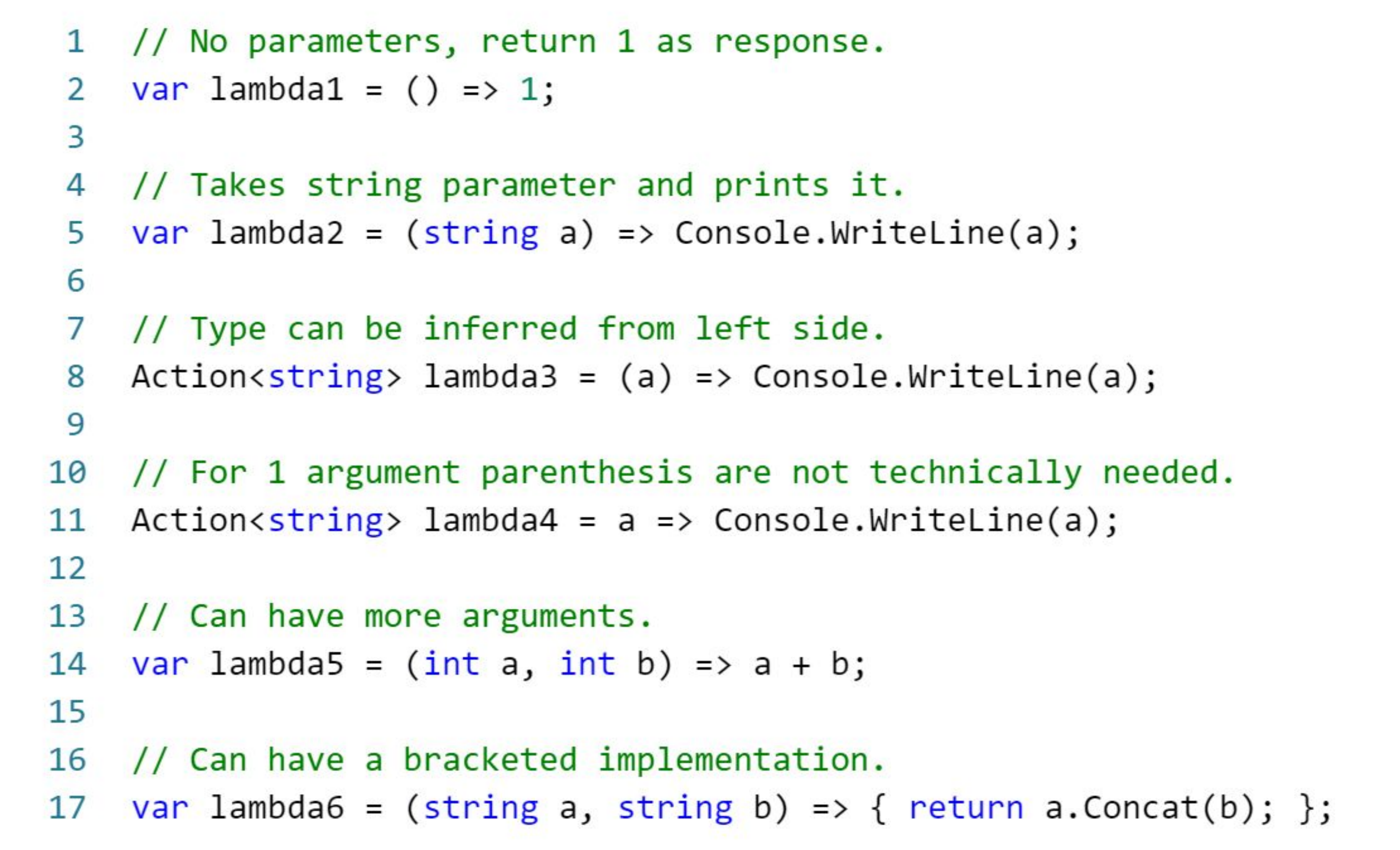
- Lambda expression – in simple and narrow-minded way – a delegate written in other way.
- Three categories: expression lambdas, statement lambdas and async lambdas.
- **lambda expression** is an anonymous function that you can use to create delegates or **expression** tree types.

Expression lambda

([variables]) => *expression*[params to be used]

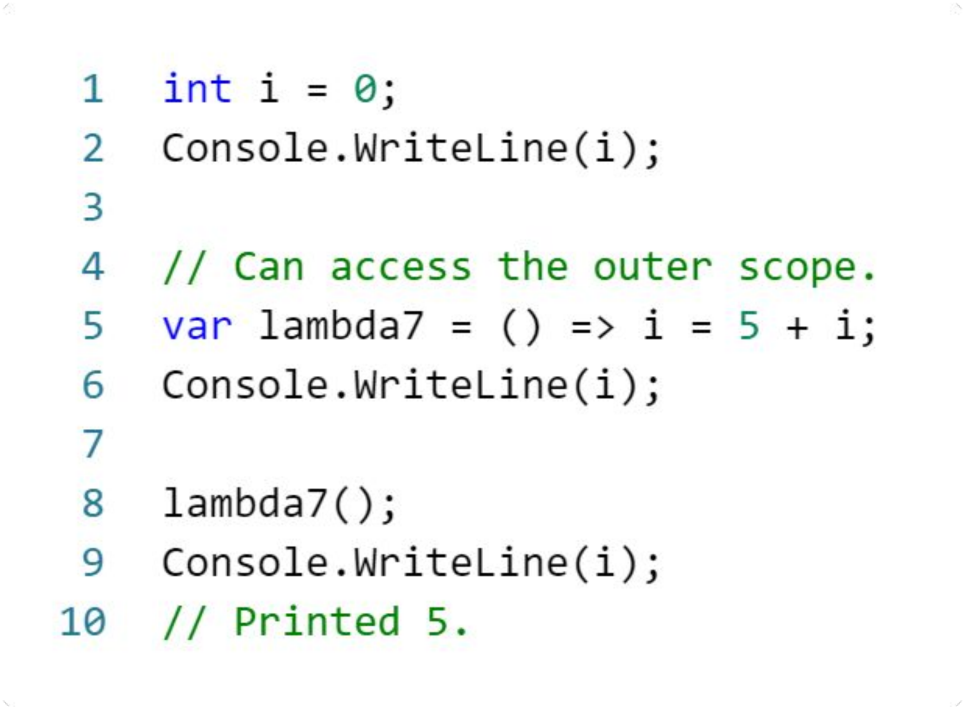
- This sentence is assigned to a delegate
- This is *expression* lambda (since only one sentence)
- **()** means empty param list
- Variable type is inferred by compiler

Expression lambda: examples

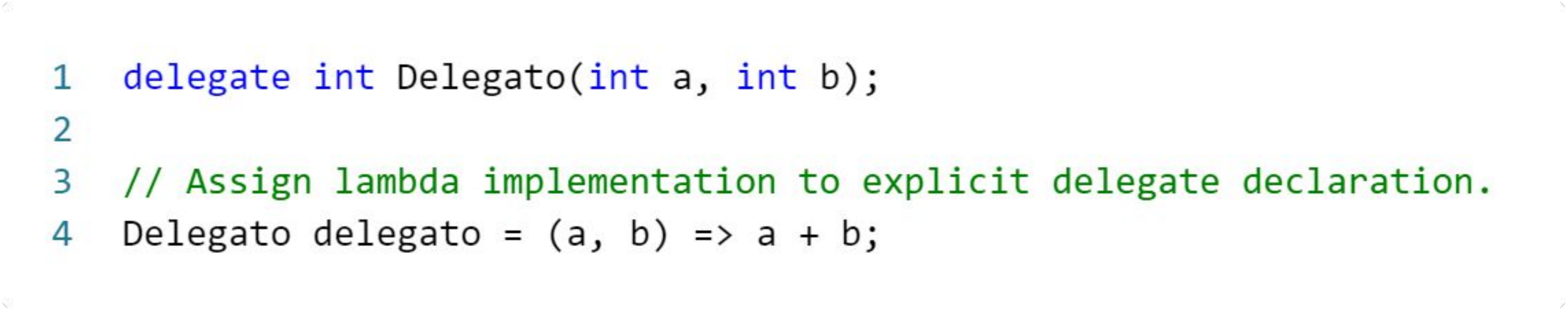


```csharp
// No parameters, return 1 as response.
var lambda1 = () => 1;

// Takes string parameter and prints it.
var lambda2 = (string a) => Console.WriteLine(a);

// Type can be inferred from left side.
Action<string> lambda3 = (a) => Console.WriteLine(a);

// For 1 argument parenthesis are not technically needed.
Action<string> lambda4 = a => Console.WriteLine(a);

// Can have more arguments.
var lambda5 = (int a, int b) => a + b;

// Can have a bracketed implementation.
var lambda6 = (string a, string b) => { return a.Concat(b); };
```

Some more lambdas



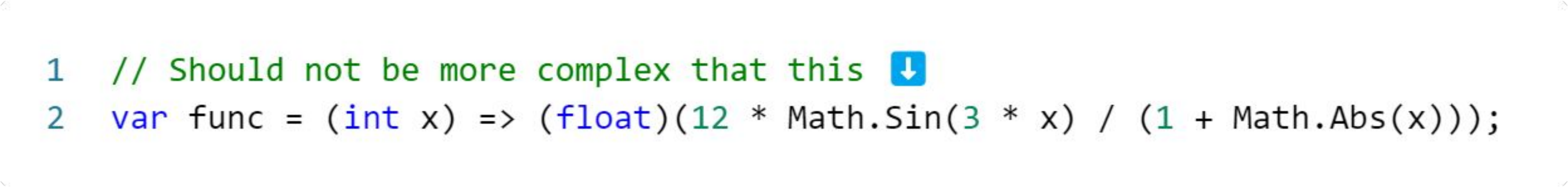
```
int i = 0;
Console.WriteLine(i);

// Can access the outer scope.
var lambda7 = () => i = 5 + i;
Console.WriteLine(i);

lambda7();
Console.WriteLine(i);
// Printed 5.
```



```
delegate int Delegato(int a, int b);

// Assign lambda implementation to explicit delegate declaration.
Delegato delegato = (a, b) => a + b;
```

Expression lambda

- Ideally lambdas should be one liners.
- Should not be more complex than:

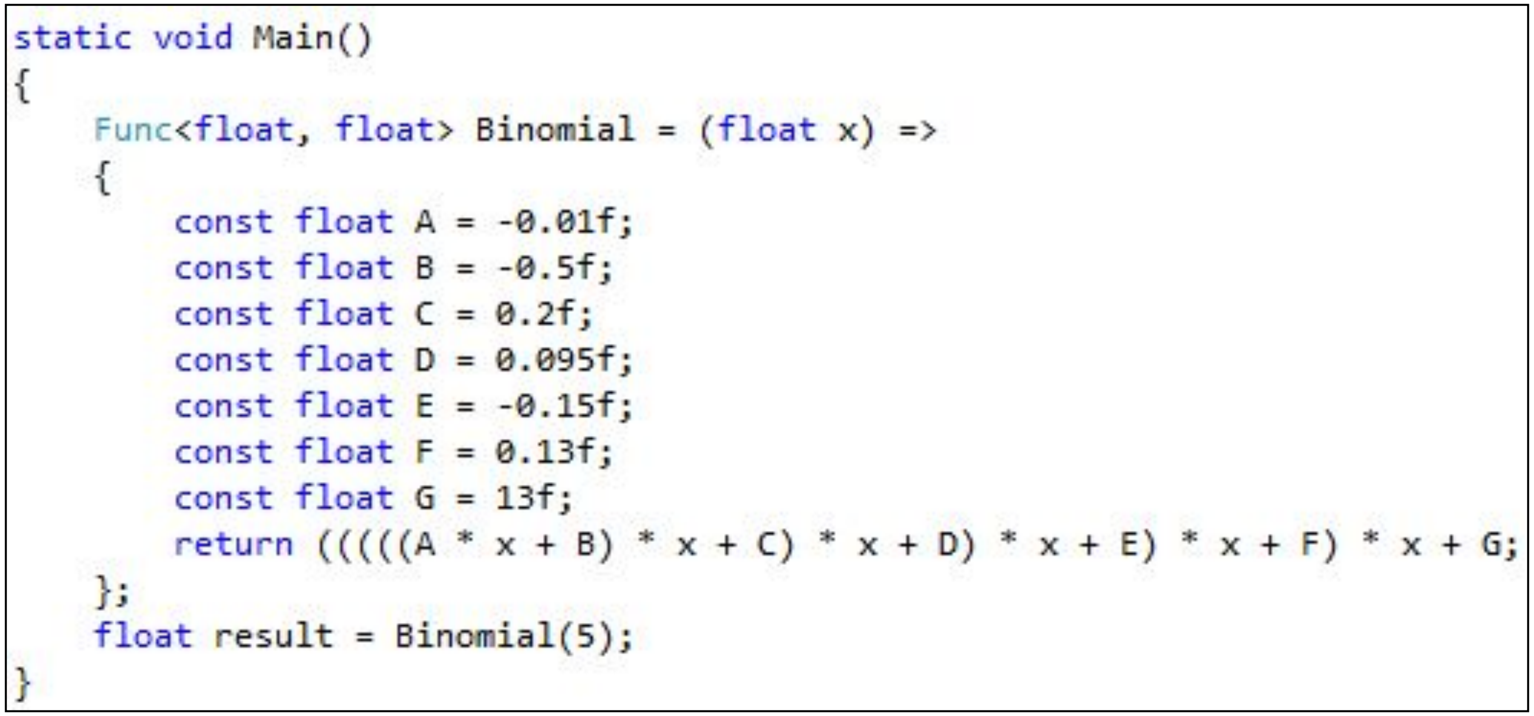


```
// Should not be more complex that this ⬇
var func = (int x) => (float)(12 * Math.Sin(3 * x) / (1 + Math.Abs(x)));
```

- Otherwise - just consider writing a regular method.

Statement lambda

- Must have {}:



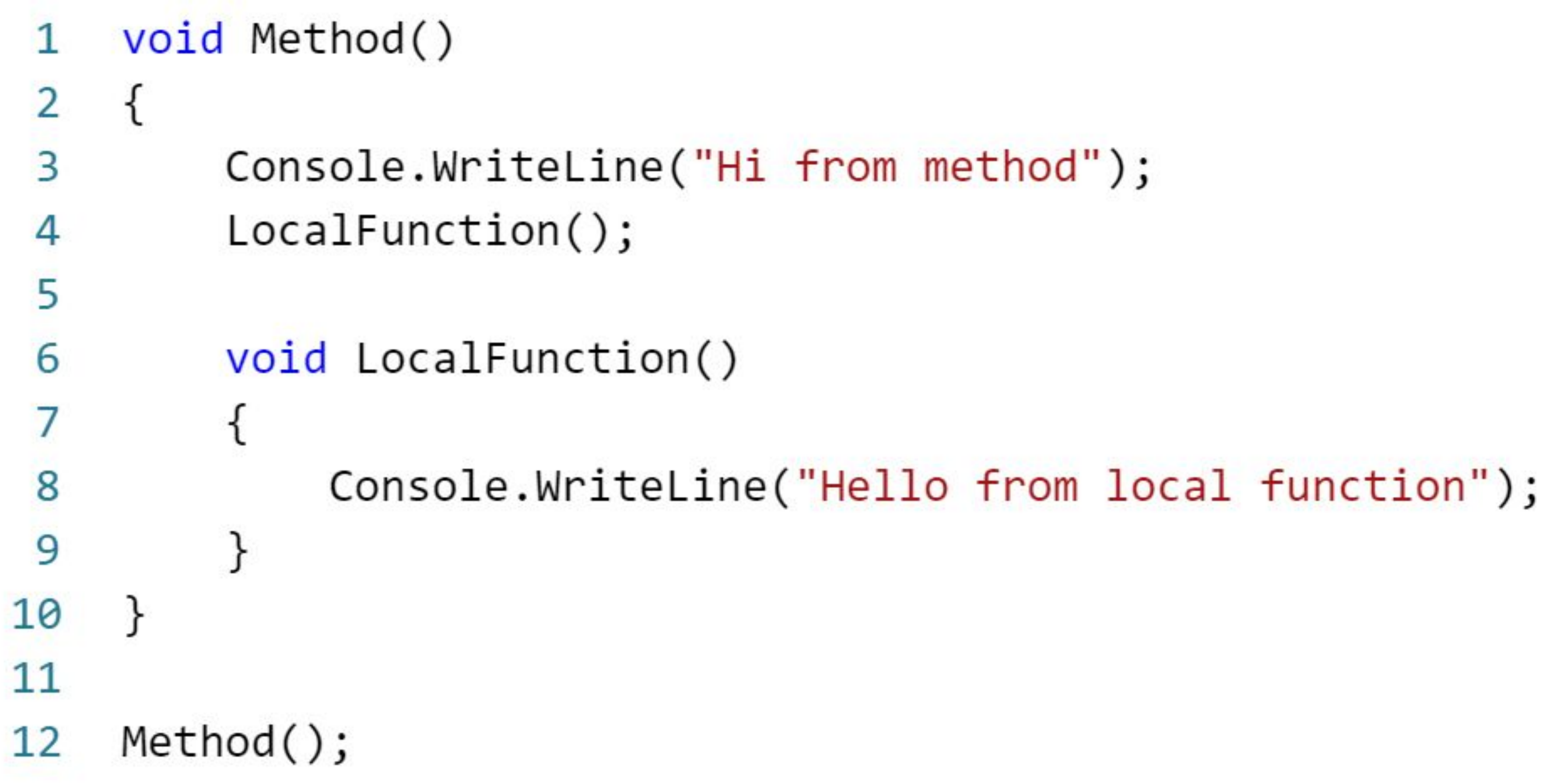
```
static void Main()
{
    Func<float, float> Binomial = (float x) =>
    {
        const float A = -0.01f;
        const float B = -0.5f;
        const float C = 0.2f;
        const float D = 0.095f;
        const float E = -0.15f;
        const float F = 0.13f;
        const float G = 13f;
        return (((((A * x + B) * x + C) * x + D) * x + E) * x + F) * x + G;
    };
    float result = Binomial(5);
}
```

- Usually consider writing method instead.

Local functions (methods)

- Looks very much like anonymous methods – but they are different
- Has better performance
- Can be generic
- Can be declared after referenced
- Recommended over lambdas if you want to implement simple logic that needs to be reused in same method, but shouldn't be used in other same class methods.

Local functions - example



```
void Method()
{
    Console.WriteLine("Hi from method");
    LocalFunction();

    void LocalFunction()
    {
        Console.WriteLine("Hello from local function");
    }
}

Method();
```

Literature

- Main:
  - MCSD toolkit:
    - Exception handling
    - Working with delegates
    - Performing I/O operations
    - Understanding serialization
  - MSDN:
  - Delegates (C# Programming Guide)
    - Covariance and Contravariance in Generics
  - How to: Combine Delegates (Multicast Delegates)(C# Programming Guide)
- Extra:
  - Codebetter.com: Foundations of Programming
    - pt 8 – Back to Basics: Exceptions
  - JsonSerialization: https://www.newtonsoft.com/json/help/html/Introduction.htm

Next lecture

- Events
- Lazy type
- Intro to LINQ

Questions

?