Csharp 6-
Classes and Objects 2

ZhongkaiKevin.Li@GeorgianCollege.ca

- Partial class
- Abstract class vs Interface
- Handling exceptions
- Events and Delegates
- A simple game

1.Partial Class

- Partial types allow to split the definition of a class or a structure between multiple files
- It allows keeping definition concise and more pointed
- Partial classes are heavily used in web and windows development in order to conceal portions of the file, which are not essential for understanding of functionality

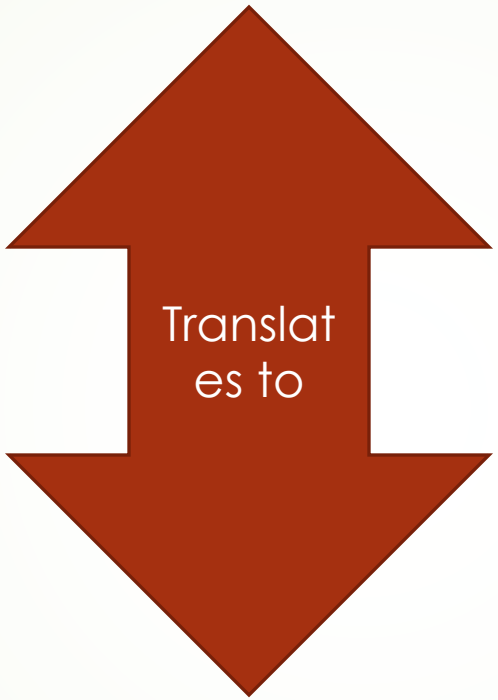
(show example in VS)

```
public partial class Employee
{
    public void DoWork()
    {
    }
}
```


```
public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

```
public class Employee
{
    public void DoWork()
    {
    }
    public void GoToLunch()
    {
    }
}
```

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```




```
partial class Earth : Planet, IRotate: IRevolve { }
```

2. Abstract class vs Interface

- Descriptions
- Samples

Abstract class

- https://www.youtube.com/watch?v=9-Lnu-OOjqs
- (Check the first half. Show codes in VS)

- Questions:
  - Would you like a gift of Animal?
  - What are the difference between using Interfaces and Abstract classes?

2.1 Answers

- **interfaces** can have no state or implementation
- a class that implements an interface must provide an implementation of all the methods of that interface

- **abstract** classes may contain state (data members) and/or implementation (methods)
- abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)

- interfaces may be multiple-inherited, abstract classes may not

2.2 Samples

2.3 Conclusions

- **Abstract** classes are used for **Modelling** a class hierarchy of similar looking classes (For example **Animal can be abstract class and Human , Lion, Tiger can be concrete derived classes.** Can you say I am going to give you an animal? Would you like to accept without knowing what exactly it is?)

- **Interface** is used for **Communication** between 2 similar / non similar classes which does not care about type of the class implementing Interface(e.g. **Height** can be interface property and it can be implemented by Human , ~~**Building , Tree**~~. It does not matter if you can eat , you can swim you can die or anything.. it matters only a thing that you need to have Height (implementation in you class)).

- Personal understanding:

- Abstract classes are good for small teams, for example for a project manager to provide some incomplete models.
- Interfaces are good for larger teams, for example, the interfaces provided by Microsoft.

3. Handling exceptions

- Exception handling help us deal with unexpected situations that occur when a program is running.
- C# uses the try, catch, and finally keywords to try actions that may not succeed, to handle failures and to clean up resources afterward.
- Exceptions can be generated by CLR, by the .NET Framework or any third-party libraries, or by application code.
- Exceptions are created by using the throw keyword.

- An exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack.
- The CLR will unwind the stack, looking for a method with a catch block for the specific exception type, and it will execute the first such catch block that if finds.
- If it finds no appropriate catch block anywhere in the call stack, it will terminate the process and display a message to the user. (Bad!)

- System.Exception is a base for all .NET exceptions (Exceptions derive directly or not directly from **System.Exception**)

- It has Message property, which gives user a message of what went wrong

- It has StackTrace property which contains a stack trace of all methods called prior to the exception being thrown.

- Stack trace is developers' bread crumbs, which show how the program was brought to the exception point.

```
class ExceptionTest
    {
        static double SafeDivision(double x, double y)
        {
            if (y == 0)
                throw new System.DivideByZeroException();
            return x / y;
        }
        static void Main()
        {
            // Input for test purposes. Change the values to see
            // exception handling behavior.
            double a = 98, b = 0;
            double result = 0;

            try
            {
                result = SafeDivision(a, b);
                Console.WriteLine("{0} divided by {1} = {2}", a, b,
                        result);
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("Attempted divide by zero.");
            }
        }
    }
```

- Use a **try** block around the statements that might throw exceptions.
- Once an exception occurs in the try block, the flow of control jumps to **the first associated exception handler** that is present anywhere in the call stack.
- In C#, the **catch** keyword is used to define an **exception handler.**
- **Do not catch an exception unless you can handle it and leave the application in a known state.** If you catch System.Exception, **rethrow** it using the **throw** keyword at the end of the catch block.

- **If a catch block defines an exception variable,** you can use it to obtain more information about the type of exception that occurred.

- Exceptions can be explicitly generated by a program by using the throw keyword.

- **Exception objects contain detailed information** about the error, such as the state of the call stack and a text description of the error.

- Exception handling supports a finally block.
- The finally block runs after the try block and any catch blocks have finished executing, whether or not an exception was thrown.
- The finally block cannot access variables that are declared within the try block

- Code in a finally block is executed even if an exception is thrown.
- Use a finally block to release resources, for example to close any streams or files that were opened in the try block.

```
throw new System.DivideByZeroException();
```

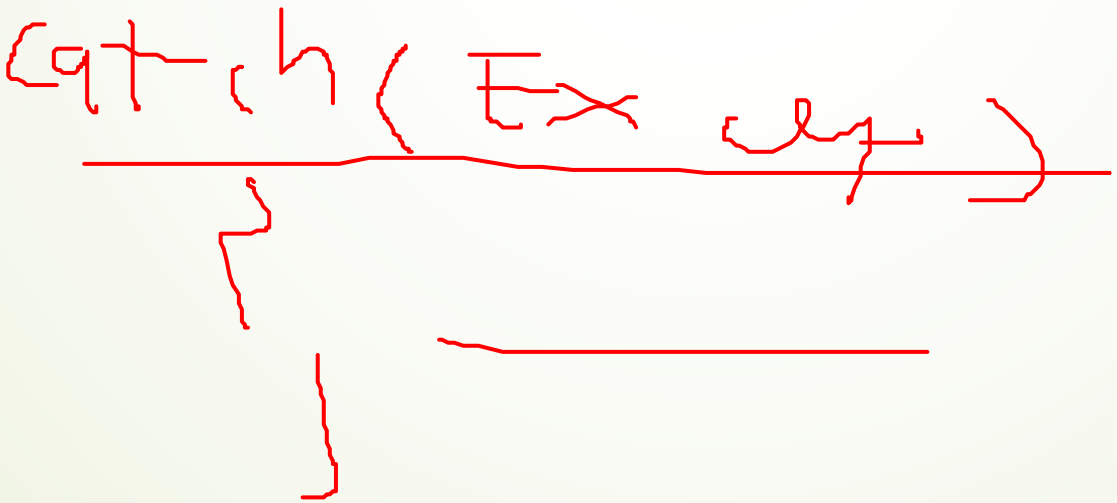
Create an instance of an exception-derived class, optionally configuring properties

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

    }
}
private static void TestThrow()
{
    CustomException ex =
        new CustomException("Custom exception in TestThrow()");

    throw ex;
}
//////////////////////////////////////////
static void TestCatch()
{
    try
    {
        TestThrow();
    }
    catch (CustomException ex)
    {
        System.Console.WriteLine(ex.ToString());
    }
}
```

- You can catch more specific exception types by adding extra catch blocks.

- The catch blocks should be specified as most-specific to least-specific because this is the order in which the runtime will examine them.

```
try
{
    int i = int.Parse(s);
}
catch (ArgumentNullException)
{
     Console.WriteLine("You need to enter a value");
}
catch (FormatException)
{
      Console.WriteLine("{0} is not a valid number. Please try again", s);
}
```



```
public static class Program
{
  public static void Main()
  {
    string s = Console.ReadLine();
    try
    {
      int i = int.Parse(s);
    }
    catch (ArgumentNullException)
    {
      Console.WriteLine("You need to enter a value");
    }
    catch (FormatException)
    {
      Console.WriteLine("{0} is not a valid number. try again", s);
    }
    finally
    {
      Console.WriteLine("Program complete.");
    }
  }
}
```

- Finally block can be prevented from running using *Environment.FailFast*.
- It has two overloads,
  - one that only takes a *string*
  - one that also takes an *exception*.
- When this method is called, the message (and optionally the exception) are written to the Windows application event log, and the application is terminated.

```
public static class Program
    {
        public static void Main()
        {

            string s = Console.ReadLine();
            try
            {
                int i = int.Parse(s);
                if (i == 0) Environment.FailFast
                        ("Special number entered");
            }
            finally
            {
                Console.WriteLine("Program complete.");
            }
            Console.ReadLine();
        }
    }
```

‘throw e;’ vs. ‘throw;’

- https://stackoverflow.com/questions/730250/is-there-a-difference-between-throw-and-throw-ex
- Test and Check it in Visual Studio

4. Delegates and Events

- A `delegate` is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure.

- A delegate allow us to specify what the function we'll be calling *looks like* without having to specify *which* function to call.

- Delegate declaration looks just like the declaration for a function, except that, **we're declaring the signature of functions that this delegate can reference.**

(Try to remember exactly even if you do not understand now.)

```
using System;

namespace Akadia.BasicDelegate
{
    // Declaration
    public delegate void SimpleDelegate();

    class TestDelegate
    {
        public static void MyFunc()
        {
            Console.WriteLine("I was called by delegate ...");
        }

        public static void Main()
        {
            // Instantiation
            SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);

            // Invocation
            simpleDelegate();
        }
    }
}
```

```
class Program
    {
        public class MyClass
        {
            // Declare no return type.
            public delegate void LogHandler(string message);
            // Check if it is pointing to a function .
            public void Process(LogHandler logHandler)
            {
                if (logHandler != null)     logHandler("Process() begin");
                if (logHandler != null)     logHandler("Process() end");
            }
        }

        // Test Application to use the defined Delegate
        public class TestApplication
        {
            static void Logger(string s)
            {
                Console.WriteLine(s);
            }

            static void Main(string[] args)
            {
                MyClass myClass = new MyClass();

                // Crate an instance of the delegate.
                MyClass.LogHandler myLogger = new MyClass.LogHandler(Logger);
                myClass.Process(myLogger);
            }
        }
    }
```

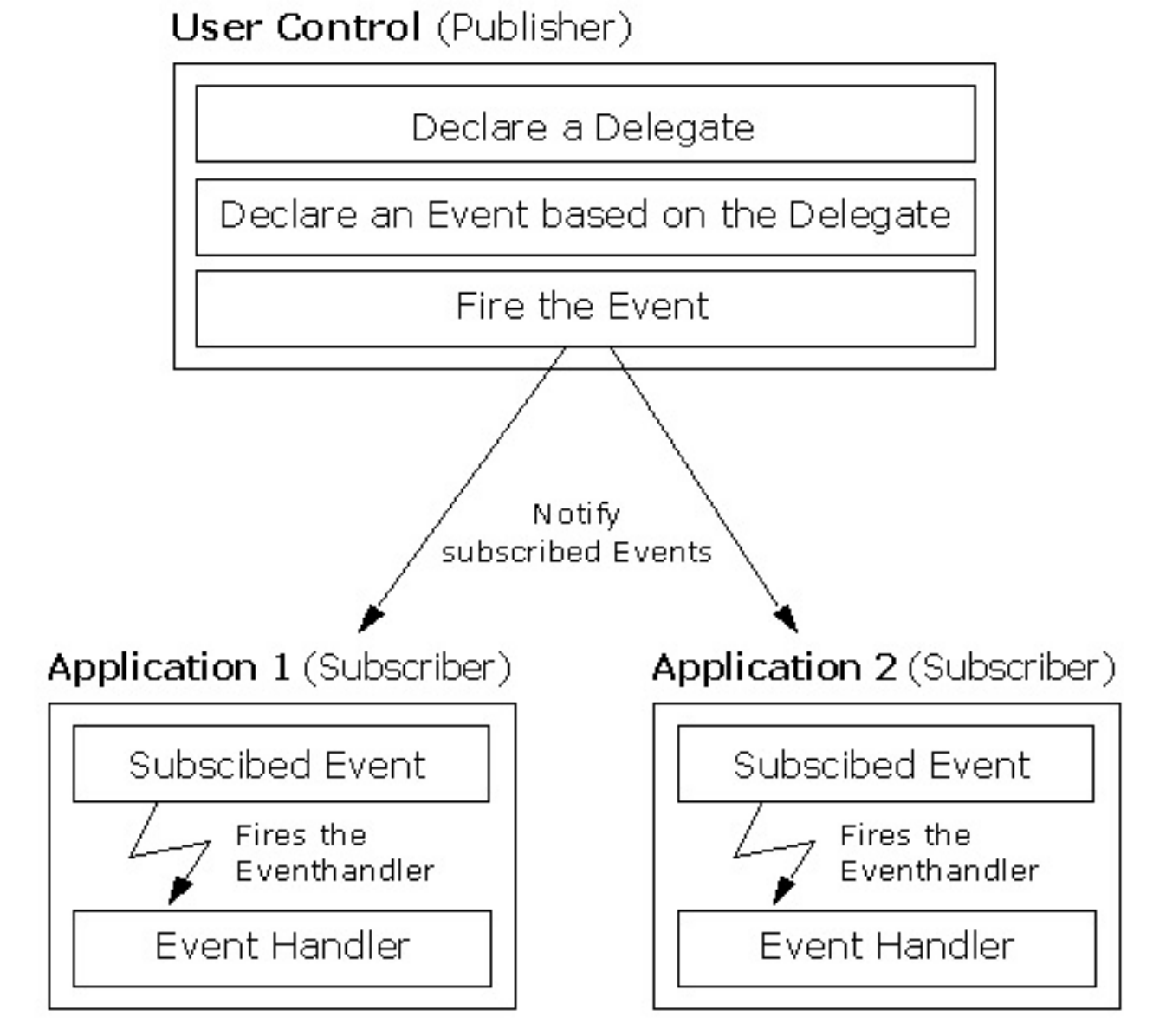
```
public class FileLogger
    {
        FileStream fileStream;
        StreamWriter streamWriter;
        // Constructor
        public FileLogger(string filename)
        {
            fileStream = new FileStream(filename, FileMode.Create);
            streamWriter = new StreamWriter(fileStream);
        }
        // Member Function which is used in the Delegate
        public void Logger(string s)
        {
            streamWriter.WriteLine(s);
        }
        public void Close()
        {
            streamWriter.Close();
            fileStream.Close();
        }
    }
    public class TestApplication
    {
        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");
            MyClass myClass = new MyClass();
            MyClass.LogHandler myLogger = new MyClass.LogHandler(fl.Logger);

            myClass.Process(myLogger);

fl.Close();
```

No change to Process() function; the code to all the delegate is the same regardless of whether it refers to a **static** or **member** function.

Events

- The Event model in C# has its roots in the event programming model that is popular in asynchronous programming.
- The foundation behind it is the idea of "publisher and subscribers."
- ***publishers*** will do some logic and publish an "event." They will then send out their event only to ***subscribers*** who have subscribed to receive the specific event.

- Any object can *publish* a set of events to which other applications can *subscribe*. When the publishing class raises an event, all the subscribed applications are notified.



Event Handlers

- Event Handlers in the .NET Framework return void and take two parameters.
  - **The first parameter is the source of the event**; that is the publishing object.
  - The second parameter is an object derived from EventArgs.
- **Events are properties** of the class publishing the event.
- Events can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the event.

```
public class Clock
    {
        private int _hour;        private int _minute;        private int _second;
        public delegate void SecondChangeHandler(object clock, TimeInfoEventArgs timeInformation);
        // The event we publish
        public event SecondChangeHandler SecondChange;
        // The method which fires the Event
        protected void OnSecondChange(object clock, TimeInfoEventArgs timeInformation)
        {
            // Check if there are any Subscribers
            if (SecondChange != null)
            {
                // Call the Event
                SecondChange(clock, timeInformation);
            }
        }
        // Set the clock running, it will raise an event for each new second
        public void Run()
        {
            for (; ; )
            {   Thread.Sleep(1000);
                // Get the current time
                System.DateTime dt = System.DateTime.Now;
                // If the second has changed notify the subscribers
                if (dt.Second != _second)
                {
                    // Create the TimeInfoEventArgs object to pass to the subscribers
                TimeInfoEventArgs timeInformation = new TimeInfoEventArgs(dt.Hour, dt.Minute, dt.Second);

                    // If anyone has subscribed, notify them
                    OnSecondChange(this, timeInformation);
                }
                // update the state
                _second = dt.Second;     _minute = dt.Minute;  _hour = dt.Hour;

            } } }
```

- The Clock class could simply print the time rather than raising an event.
- The advantage of the publish / subscribe is that any number of classes can be notified when an event is raised.
- The subscribing classes do not need to know how the Clock works,
- Clock does not need to know what they are going to do in response to the event.
- Similarly a button can publish an Onclick event, and any number of unrelated objects can subscribe to that event, receiving notification when the button is clicked.

- **The publisher and the subscribers are decoupled by the delegate.**
- This is highly desirable as it makes for more **flexible** and **robust** code.
- The clock can change how it detects time without breaking any of the subscribing classes.
- The subscribing classes can change how they respond to time changes without breaking the Clock.