CS411 Short Notes Mid term

Lec 1 to18

BY vuonlinehelp.blogspot.com

YouTube link:

https://www.youtube.com/@vuonlinehelp

Website Link:

https://vuonlinehelp.blogspot.com/

Important Note

You should never copy paste the same answer which we are providing. Please do make some changes in the solution file. If you appreciate our work, please share our website vuonlinehelp.blogspot.com with your friends.

آپ کو کبھی بھی وہی جواب کاپی پیسٹ نہیں کرنا چاہیے جو ہم فراہم کر رہے ہیں۔ براہ کرم حل فائل میں کچھ تبدیلیاں کریں۔ اگر آپ ہمارے کام کی تعریف کرتے ہیں تو براہ کرم ہماری ویب سائٹ vuonlinehelp.blogspot.com کو اپنے دوستوں کے ساتھ شیئر کریں

Visual Programming - CS411

Chapter 1

Lecture 1

Introduction

The first one is authored by Opher Etzion and Peter Niblett and is one of the very few books on event processing. Event processing is often taught as a side-concept, but we will focus on it in this course. The other book is by Adam Nathan. It is about Windows presentation foundation or WPF in short. We use it as our gui example with C#. We will introduce C# concepts and WPF concepts. They are not a pre-requisite for this course.

We will touch event-driven programming in the browser, in particular made possible by techniques called AJAX (Asynchronous JavaScript and XML.). Again not a pre-requisite.

It's primarily event driven concepts and application of these concepts on GUI programs on desktop, web, and mobile.

Here is our first example:

```
Using namespace std;
#include<iostream>
Int main()
{
Char a;
Do {
A = cin.get();
Cout << a;
} while (a!='x');
Return 0;
}
```

So what are the pros and cons of the re-factored approach?

We do one thing at a time (more focused approach). Blocking is a huge issue. Nothing is parallel. The key idea is that once you understand the approach, it scales really well to a large number of things you want to respond to.

Chapter 2

Lecture 2

What is an event?

It's an occurrence within a particular system or domain. There are two meanings:

- Something that happened
- The corresponding detection in computer world.

An event captures “some” things from the actual occurrence and multiple events may capture “one” occurrence.

Probabilistic events may or may not relate to an actual occurrence e.g. a fraud detection event on a banking transaction. Every event is represented by an event-object. There are various types of events and information in the event describes details of the particular event type. E.g. Key press, file event etc.

Interrupts and exceptions on a computer e.g. divide by zero. Patient monitored by sensors. Car sensors alerting to oil or pressure situations. Banking alerts. Road tolling. Etc. Event processing is computing that performs operations on events. Common event processing operations include reading, creating, transforming, and deleting events. The design, coding and operation of applications that use events, either directly or indirectly is called event-based programming or applications based on event-driven architecture. So what if we don’t use event-driven programming. We will poll for events. Even if you make your application non-event-driven, you still wait for events. Wait for a single event is blocking operation.

Synchronous operations are completed before the next operation can be started. Asynchronous operations can be started and we can do something else before they are completed.

Why do we want our applications to be event based?

They are easier to scale. Well suited to Visual programming where multiple GUI elements and many sources of events exist. It has a direct mapping to real world. Its deterministic-there is an exact mapping between a situation in the real world and its representation in the event processing system.

At the same time it’s approximate-the event processing system provides an approximation to real world events. So what kind of events are there in non-real time applications. Mouse and keyboard, secondary events of GUI elements, file events, message based parallel and local communication, network events etc. Here is flower delivery example from the book.

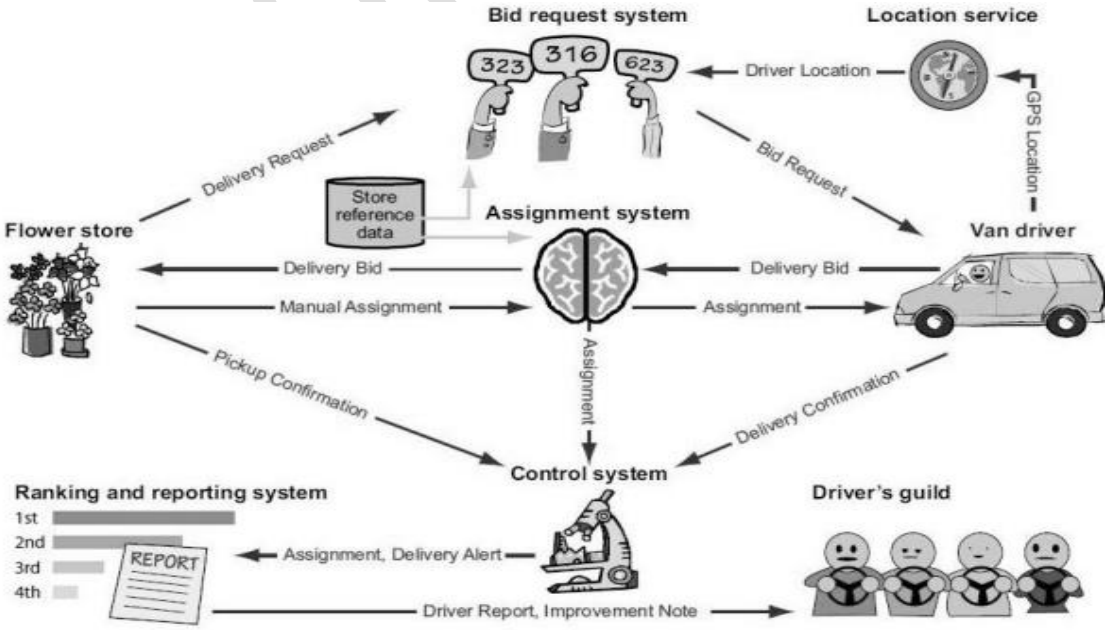


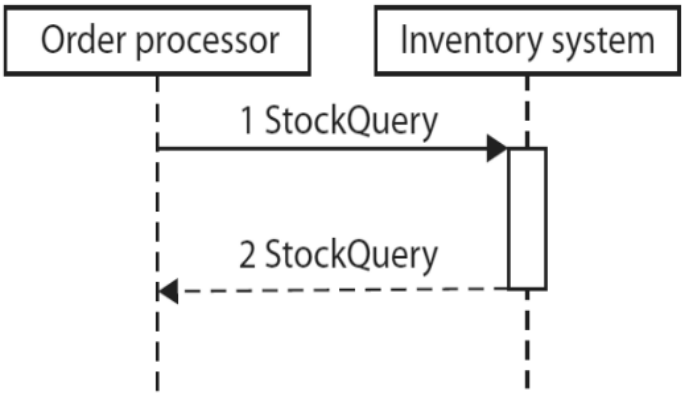
Figure 1.6 Parts of the Fast Flower Delivery application. The arrows represent the flow of events, and the pictures represent the various entities involved in the network.

Chapter 3

Lecture 3

Request and response based applications.

Example is a web browser which sends queries and updates. It performs synchronous interactions. It looks like this:



Events are based on the principle of decoupling

Let's compare events and request-response based architecture. Events have already happened whereas requests are asking something to happen. Service requester or client contacts the service provider or the server. In event-driven architecture event producer sends event to event consumer.

We send a request to query the state or use "change events". Event has meaning independent of its producers and consumers whereas the same cannot be said about requests. Events are often one way (push events). Events decouple producers and consumers. Events can be processed asynchronously. Latency is reduced in comparison to the "pull" style.

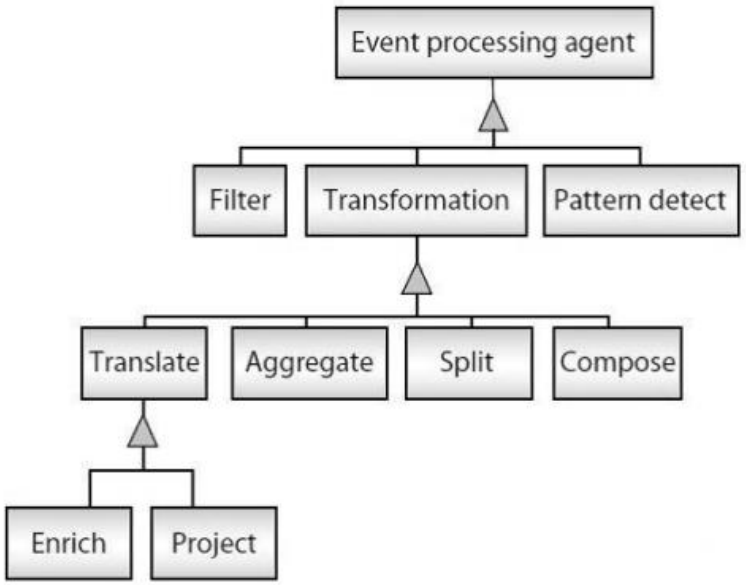
Events can also be compared to messages. Messages may contain events. Events may live outside events like in event logs. An event channel is a subscription mechanism for events. It further decouples consumers and producers. Can even be an intermediate XML file to store events.

Event-driven architecture and service-driven architecture can also be compared. Event-based programming, also called event-driven architecture (EDA) is an architectural style in which one or more components in a software system execute in response to receiving one or more event notifications. Service-oriented architecture (SOA) is built from request-response. It moves away from monolithic applications. There are many similarities and often a component can provide both modes of contacting it.

- EVENT PRODUCER: An event producer is an entity at the edge of an event process- ing system that introduces events into the system.
- EVENT CONSUMER: An event consumer is an entity at the edge of an event pro- cessing system that receives events from the system.
- RAW EVENT: A raw event is an event that is introduced into an event processing system by an event producer.
- DERIVED EVENT: A derived event is an event that is generated as a result of event processing that takes place inside an event processing system.

- STATELESS EVENT PROCESSING: An event processing agent is said to be stateless if the way it processes one event does not influence the way it processes any subsequent events.
- EVENT STREAM: An event stream (or stream) is a set of associated events. It is often a temporally totally ordered set (that is to say, there is a well-defined timestamp-based order to the events in the stream).

A stream in which all the events must be of the same type is called a homogeneous event stream; a stream in which the events may be of different types is referred to as a heterogeneous event stream. Here are types of event agents.



What aspects of real occurrence do event objects capture?

An event type is a specification for a set of event objects that have the same semantic intent and same structure; every event object is considered to be an instance of an event type. An event attribute is a component of the structure of an event. Each attribute has a name and a data type. It can also have occurrence/detection time, certainty, source, location. Events can be composed, generalized, and specialized. Let's experiment with flower delivery example.

Can you find events and their attributes?

Event producers can produce events that are hardware generated, software generated, or from human interaction. Event consumers have similar types.

Examples are locking or unlocking a door, raising or lowering a barrier, applying the brakes on a vehicle, opening or closing a valve, controlling a railroad switch, and turning a piece of equipment on or off. Try to find producers and consumers in flower delivery application.

Chapter 4

Lecture 4

History of Java and J++

Microsoft wanted to extend Java to communicate it with COM. Sun did not want that as it would make Java platform dependent. Microsoft wanted a clean-room implementation of Java.

What is a clean-room implementation?

Clean room design (also known as the Chinese wall technique) is the method of copying a design by reverse engineering and then recreating it without infringing any of the copyrights and trade secrets associated with the original design. Clean room design is useful as a defense against copyright and trade secret infringement because it relies on independent invention. However, because independent invention is not a defense against patents, clean room designs typically cannot be used to circumvent patent restrictions.

Where the name for C #does comes from:

The initial name was "Cool", which stood for "C-like Object Oriented Language". What's the difference between .NET and C#.

- C# design most directly reflects
- .NET (CLR) design but C# is a language and .NET is the platform.

Install setup

As a first step, you should get Visual Studio from http://www.microsoft.com/visualstudio. Click Downloads and choose "Visual Studio Express 2012 for Windows Desktop". Choose "Download now" or "Install now". You can save iso file to disk (right click to mount in win8) otherwise there are free utilities to mount it or burn it on a cd.

Install and register online for continued use. To create a new project chooses File, New Project, Visual C#, Console Application. Then click the Start key.

Here are some notable features of C#.

- No global variables or functions.
- Locals cannot shadow global variables.
- There is a strict boolean type.
- Memory address pointers can only be used in specifically marked "unsafe" blocks and require permissions
- No instruction to "free" memory. Only garbage collection.
- Try-finally block.
- No "multiple inheritance" but interfaces supported.
- Operator overloading allowed.
- More type-safe (only integer widening allowed).
- Enumeration members are scoped.
- Property syntax for getters and setters.
- No checked exceptions.
- Some functional programming features like function objects and lambda expressions.

The common type system of C# has value types and reference types. Instances of value types do not have referential identity nor referential comparison semantics. Value types are derived from System. Value type and can always be created, copied and have a default value (int, float, char, System. date time, enum, struct).

In contrast, reference types have the notion of referential identity. Default equality and inequality comparisons test for referential rather than structural equality unless overloaded e.g System. String. Not “always” possible to create an instance of a reference type, copy an existing instance, perform a value comparison on two instances, though specific types “can” provide such services by exposing a public constructor or implementing a corresponding interface e.g. Object, System. String, System. Array.

Comments are written

Pre-processor directives like #if, #else, #endif, #region, #endregion are supported. Comments are written using // and /* */.

Here is how to box and unbox variables in C#. Boxing stores a value type in a reference type.

```
Int foo = 42;// Value type.

Object bar = foo;// foo is boxed to bar.

Int foo2 = (int)bar;// Unboxed back to value type.
```

The C# specification details a minimum set of types and class libraries that the compiler expects to have available. In practice, C# is most often used with some implementation of the Common Language Infrastructure (CLI), which is standardized as ECMA-335 Common Language Infrastructure (CLI).

Here is a hello world program in C#.

```
Using System;

 Class Program {

Static void Main() {

 Console.writeline("Hello world!");

 }

}
```

Chapter 5

Lecture 5

Boolean types

Here is an example with Boolean types.

```
Using System;
Class Booleans
{
Public static void Main()
{
Bool content = true;
Bool nocontent = false;
Console.writeline("It is {0} that C\# Station provides C\# programming language content.", content);
Console.writeline("The statement above is not {0}.", nocontent);
}
}
```

Here are the integer types in C#.

Type	Size (in bits)	Range
sbyte	8	-128 to 127
byte	8	0 to 255
short	16	-32768 to 32767
ushort	16	0 to 65535
int	32	-2147483648 to 2147483647
uint	32	0 to 4294967295
long	64	-9223372036854775808 to 9223372036854775807
ulong	64	0 to 18446744073709551615
char	16	0 to 65535

And here are the floating point and decimal types in C#.

Type	Size (in bits)	precision	Range
float	32	7 digits	1.5×10^{-45} to 3.4×10^{38}
double	64	15-16 digits	5.0×10^{-324} to 1.7×10^{308}
decimal	128	28-29 decimal places	1.0×10^{-28} to 7.9×10^{28}

The System.String type supports the following sequences.

Char	Meaning	Value
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The escape character is while @ is the verbatim character. The following operators are supported by C#.

Category (by precedence)	Operator(s)	Associativity
Primary	x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate	left
Unary	+ - ! ~ ++x --x (T)x	right
Multiplicative	* / %	left
Additive	+ -	left
Shift	<< >>	left
Relational	< > <= >= is as	left
Equality	== !=	right
Logical AND	&	left
Logical XOR	^	left
Logical OR	|	left
Conditional AND	&&	left
Conditional OR	||	left
Null Coalescing	??	left
Ternary	?:	right
Assignment	= *= /= %= += -= <<= >>= &= ^= |= =>	right

Array type

Here is an example of using the Array type in C#.

```
Using System;

Class Array

{

Public static void Main(){

Int[] myints = { 5, 10, 15 };

Bool[][] mybools = new bool[2][];

Mybools[0] = new bool[2];

Mybools[1] = new bool[1];
```

```
Double[,] mydoubles = new double[2, 2];
String[] mystrings = new string[3];
Console.writeline("myints[0]: {0}, myints[1]: {1}, myints[2]: {2}", myints[0], myints[1], myints[2]);
Mybools[0][0] = true;
Mybools[0][1] = false;
Mybools[1][0] = true;
Console.writeline("mybools[0][0]: {0}, mybools[1][0]: {1}", mybools[0][0], mybools[1][0]);
Mydoubles[0, 0] = 3.147;
Mydoubles[0, 1] = 7.157;
Mydoubles[1, 1] = 2.117;
Mydoubles[1, 0] = 56.00138917;
Console.writeline("mydoubles[0, 0]: {0}, mydoubles[1, 0]: {1}", mydoubles[0, 0], mydoubles[1, 0]);
Mystrings[0] = "Joe";
Mystrings[1] = "Matt";
Mystrings[2] = "Robert";
Console.writeline("mystrings[0]: {0}, mystrings[1]: {1}, mystrings[2]: {2}", mystrings[0], mystrings[1],
mystrings[2]);
}
}
```

You can make jagged arrays or multi-dimensional arrays. Jagged is basically array of arrays. Array size is any integer type value. It uses a zero-based index.

Control statements like if and else work much like C++.

```
If (myint< 0 || myint == 0) {
Console.writeline("Your number {0} is less than or equal to zero.", myint);
} else if (myint> 0 && myint <= 10){
Console.writeline("Your number {0} is in the range from 1 to 10.", myint);
} else if (myint> 10 && myint <= 20){
Console.writeline("Your number {0} is in the range from 11 to 20.", myint);
} else if(myint> 20 && myint <= 30){
Console.writeline("Your number {0} is in the range from 21 to 30.", myint);
} else{ Console.writeline("Your number {0} is greater than30.", myint); }
```

Switch statement can work for booleans, enums, integral types, and strings. You break the switch statement

Branching statement	Description
break	Leaves the switch block
continue	Leaves the switch block, skips remaining logic in enclosing loop, and goes back to loop condition to determine if loop should be executed again from the beginning. Works only if switch statement is in a loop as described in Lesson 04: Control Statements - Loops.
goto	Leaves the switch block and jumps directly to a label of the form "<labelname>:"
return	Leaves the current method. Methods are described in more detail in Lesson 05: Methods.
throw	Throws an exception, as discussed in Lesson 15: Introduction to Exception Handling.

Branching statements

With one of the following branching statements:

The following four kinds of loops are available.

1. While (<boolean expression >) {< statements > }
2. Do {< statements > } while (<boolean expression >);
3. For (<initializer list >;<boolean expression > ;< iterator list >) { < statements >}
4. foreach (<type><> in<list>) {< statements > }

Methods are of the form.

Attributes modifiers return-type method-name(parameters) { statements }
Elements are accessed using the dot operator. Object variables are references to the original object not the object themselves. Here is an example of a method.

```
Using System;

Class onemethod

{

Public static void Main()

{

String mychoice;

Onemethod om = new onemethod();

Mychoice = om.getchoice();

}

String getchoice()

{
Return “example”;

}
}
```

Chapter 6

Lecture 6

Namespaces allow name reuse

For example, a Console class can reside in multiple libraries. Here is an example of namespace declaration:

```
// Namespace Declaration
Using System;
// The C\# Station Namespace
Namespace csharp_station
{
// Program start class
Class namespacecss {
// Main begins program execution.
Public static void Main()
{
// Write to console
Console.writeline("This is the new C\# Station Namespace.");
}
}
}
```

Multiple classes

Multiple classes are conventionally stored in multiple files. To create a new class in IDE, use the new class wizard. Multiple .cs files can be compiled as csc a.cs b.cs c.cs. Default ctors are written with no arguments when no ctor is written. An initializer list can be used to use an alternate constructor.

```
Public outputclass() : this("Default Constructor String") { }
```

```
Multiple ctors can be written. Types of class members in C# are instance and static. For every new occurrence there are new instance vars.
 Outputclass oc1 = new outputclass("outputclass1");
Outputclass oc2 = new outputclass("outputclass2");
```

Now oc1.printstring or oc2.printstring will access different instance variables. On the other hand static members have only one copy.

```
Public static void staticprinter(){
Console.writeline("There is only one of me.");
}
```

Static ctor exists to initialize class static members. It's called only once. It has no parameters. Destructors (dtors) are called by garbage collector. "Public" methods accessed outside class. "Public" classes accessed outside assembly. A class can contain Constructors, Destructors, Fields, Methods, Properties, Indexers, Delegates, Events, and Nested Classes.

Inheritance introduces the concept of base classes and derived classes. Declaration of inheritance is done as Derived:

- Base
- Only single inheritance is supported.

Derived class

Derived class is exactly the same as base. Child class "is a" parent class except that derived is more special. Bases are automatically initialized before derived. How can derived communicate with the base i.e. the specialized part communicate with the generic part. By using ": base ()" at ctor time or "base.x()" later. "New" keyword used to override methods. Cast back to base type to call an overridden method of base.

Polymorphism

Polymorphism means to invoke derived class methods through base class reference during run-time. Normally (as we saw) base class casting means base class method is called. It's handy when a group of objects is stored in array/list and we invoke some method on all of them. They don't have to be the same type but if related by inheritance, they can be cast.

Polymorphic

Polymorphic methods are declared with "virtual" keyword and the "override" keyword is used to provide an implementation. Polymorphism needs the signatures to be the same. Because of inheritance, the three derived classes can be treated as the base class. Because of polymorphism, the methods from drived can still be called.

Chapter 7

Lecture 7

Auto-implemented properties

Auto-implemented properties improve the common-case we saw. Here is the same example using auto implemented properties. Properties have the same idea as getters and setters. They just allow simplified syntax. Same idea of backing store and a simplified syntax for the general case is enabled by auto implemented properties.

```
Using System;
Public class Customer
{
 Public int ID { get; set; }
 Public string Name { get; set; }
}
Public class autoimplementedcustomermanager
{
 Static void Main()
 {
```

```
Customer cust = new Customer();
Cust.ID = 1;
Cust.Name = "Amelio Rosales";
Console.writeline(
"ID: {0}, Name: {1}",
Cust.ID,
Cust.Name);
Console.readkey();
}
}
```

Indexer

An indexer enables your class to be treated like an array. However you have your internal data representation not an actual array it's implemented using "this" keyword and square brackets syntax. It looks like a property implementation.

Indexers can take any number of parameters. The parameters can be integers, strings, or enums. Additionally they can be overloaded with different type and number of parameters.

Struct

It's a value-type, whereas class is a reference-type. Value types hold their value in memory where they are declared, but reference types hold a reference to an object in memory. If you copy a class, C# creates a new copy of the reference to the object and assigns the copy of the reference to the separate class instance. Structs can't have destructors, can't have implementation inheritance (still can have interface inheritance). Many built-in types are structs like System.Int32 is a C# int. C# built-in types are aliases for .NET Framework types. Syntax of struct and class are very similar.

Interfaces

They are declared like a class but have no implementation. There are only declarations of events, indexers, methods and/or properties. They are inherited by classes which provide the real implementation.

What are interfaces good for if they don't implement functionality?

They are great for putting together plug-n-play like architectures where components can be interchanged at will. The interface forces each component to expose specific public members that will be used in a certain way.

Interfaces define a contract. For instance, if class foo implements the idisposable interface, it is making a statement that it guarantees it has the Dispose () method, which is the only member of the idisposable interface. Any code that wishes to use class foo may check to see if class foo implements idisposable. When the answer is true, then the code knows that it can call foo.Dispose()

To define an interface, we use the following syntax.

```
Interface imyinterface

{

Void methodtoimplement();
```

```
}
The "I" prefix is a convention. There is no implementation, only semi-colon. There are only method signatures.
Let's now see how its used.
Class interfaceimplementer : imyinterface
{
Static void Main()
{
Interfaceimplementer iimp = new interfaceimplementer();
Iimp.methodtoimplement();
}
Public void methodtoimplement()
{
Console.writeline("methodtoimplement() called.");
}
}
```

Chapter 8

Lecture 8

Delegate

A delegate is a reference to a method. It's like function pointers in some other languages. Methods are algorithms that operate on data. Sometimes the data needs a special operation e.g. A comparator in a sorting routine. We can use a bad solution which would an if then else for all types we want our sorting routine to work for. There are two good solutions. One is to implement a comparator interface in all types we want and then pass an instance of a type that provides that interface. The other solution is using delegates i.e. References to functions.

A C# event is a class member that is activated whenever the event it was designed for occurs (fires).

Anyone interested in the event can register and be notified as soon as the event fires. At the time an event fires, methods registered with the event will be invoked.

Events and delegates work hand in hand

Any class, including the same class that the event is declared in, may register one of its methods with the event. This occurs through a delegate, which specifies the signature of the method that is registered for the event. The delegate may be one of the pre-defined .NET delegates or one that you

declare yourself. Then, you assign the delegate to the event, which effectively registers the method that will be called when the event fires.

This class inherits from Form, which essentially makes it a Windows Form. This automatically gives you all the functionality of a Windows Form, including Title. Bar, Minimize/Maximize/Close buttons, System Menu, and Borders. It is started by calling the Run() method of the static Application object with a reference to the form object as its parameter. The += syntax registers a delegate with the event. Its fired like a method call. The Click event is pre-defined, event handler delegate is also already in .NET. All you do is define your callback method (delegate handler method) that is invoked when someone presses the clickme button. The onclickmeclicked() method, conforms to the signature of the eventhandler delegate.

Exception handling

There can be unforeseen errors in the program. There are some normal errors and some exceptional errors e.g. File i/o error, system out of memory, null pointer exception. Exceptions are "thrown". They are derived from System.Exception class. They have a message property, contain a stacktrace, a tostring method. Identifying the exceptions you'll need to handle depends on the routine you're writing e.g. System.IO.File.openread() could throw securityexception, argumentexception, argumentnullexception, pathtoolongexception, directorynotfoundexception, unauthorizedaccessexception, filenotfoundexception, or notsupportedexception.

Here is an example of exception handling:

```
Using System;
Using System.IO;
Class trycatchdemo
{
 Static void Main(string[] args)
 {
 Try
 {
 File.openread("nonexistentfile");
 }
 Catch(Exception ex)
 {
 Console.writeline(ex.tostring());
 }
 }
}
```

A single exception can be handled differently as well:

```
catch (FileNotFoundException fnfEx)
{
   // Handle the case when the file is not found
   Console.WriteLine("File not found: " + fnfEx.ToString());
}
catch (Exception ex)
{
```

```
    // Handle any other types of exceptions
    Console.WriteLine("An error occurred: " + ex.ToString());
}
```

First handler that handles an exception catches it. Otherwise the exception bubbles up the caller stack until someone handles it.

Chapter 9

Lecture 9

Attributes add declarative information

They are used for various purposes during runtime. They can be used at design time by application development tools.

For example, dllimpor-tattribute allows a program to communicate with Win32 libraries. Obsolete-attribute causes a compile-time warning to appear. Such things would be difficult to accomplish with normal code. Attributes add metadata to your programs.

When your C# program is compiled, it creates a file called an assembly, which is normally an executable or DLL library. Assemblies are self-describing because they have metadata. Via reflection, a program's attributes can be retrieved from its assembly metadata. Attributes are classes that can be written in C# and are used to decorate your code with declarative information. It's a powerful concept. It lets you extend your language by creating customized declarative syntax with attributes.

Attributes are generally applied physically in front of type and type member declarations. They are declared with square brackets, "[" and "]" surrounding the attribute such as " [obsoleteattribute]". The "Attribute" part of the attribute name is optional i.e. "[Obsolete]" is correct as well. Parameter lists are also possible.

```
Using System;

Class basicattributedemo

{

 [Obsolete]

 Public void myfirstdeprecatedmethod()

 {

 Console.writeline("Called myfirstdeprecatedmethod().");

 }

 [obsoleteattribute]

 Public void myseconddeprecatedmethod()

 {

 Console.writeline("Called myseconddeprecatedmethod().");

 }

 [Obsolete("You shouldn't use this method anymore.")]

 Public void mythirddeprecatedmethod()

 {
```

```
 Console.writeline("Called mythirddeprecatedmethod().");
 }
 // make the program thread safe for COM
 [stathread]
 Static void Main(string[] args)
 {
 Basicattributedemo attrdemo = new basicattributedemo();
 Attrdemo.myfirstdeprecatedmethod();
 Attrdemo.myseconddeprecatedmethod();
 Attrdemo.mythirddeprecatedmethod();
 }
}
```

Stathread is a common attribute you will see later. It stands for Single Threaded Apartment model which is used for communicating with unmanaged COM.

Attribute parameters can be either positional parameters or named parameters. Usually named parameters with optional stuff, but positional can be optional as well e.g. "[Obsolete("You shouldn't use this method anymore.", true)] public void mythirddeprecatedmethod()" will give an error instead of warning.

The attribute dllimportattribute has both positional and named parameters "[dllimport("User32.dll", entrypoint="messagebox")]". Positional parameters come before named parameters. There is no order requirement on named parameters.

```
Using System;
[assembly: clscompliant(true)]
Public class attributetargetdemo
{
 Public void nonclscompliantmethod(uint nclsparam)
 {
 Console.writeline("Called nonclscompliantmethod().");
 }
 [stathread]
 Static void Main(string[] args)
 {
 Uint myuint = 0;
 Attributetargetdemo tgtdemo = new attributetargetdemo();
 Tgtdemo.nonclscompliantmethod(myuint);
 }
}
```

We rarely write new attributes but we use them extensively.

Enums (or enumerations) are strongly typed constants. They are unique types that allow you to assign symbolic names to integral values. Enum of one type may not be implicitly assigned to an enum of another type (even though the underlying value of their members is the same). All assignments between different enum types and integral types require an explicit cast. It allows you

to work with integral values, but using a meaningful name. North, South, East, and West or the zet of integers 0, 1, 2, and 3. C# type, enum, inherits the Base Class Library (BCL) type, Enum.

```
Using System;
// declares the enum
Public enum Volume
{
 Low,_
 Medium,
 High
}
// demonstrates how to use the enum
Class enumswitch
{
 Static void Main()
 {
 // create and initialize
 // instance of enum type
 Volume myvolume = Volume.Medium;
 // make decision based
 // on enum value
 Switch (myvolume)
 {
 Case Volume.Low:
 Console.writeline("The volume has been turned Down."); Break;
Case Volume.Medium:
 Console.writeline("The volume is in the middle.");
 Break;
 Case Volume.High:
 Console.writeline("The volume has been turned up."); Break;
 }
 Console.readline();
 }
}
```

Default underlying type of an enum is "int". It can be changed by specifying a "base". Valid base types include byte, sbyte, short, ushort, int, uint, long, and ulong. Default value of first member is 0. You can assign any member any value. If it is unassigned it gets +1 the value of its predecessor.

Overloaded operators

Overloaded operators must be static. They must be declared in the class for which the operator is defined. It is required that matching operators are both defined, e.g. == and!= so that the behaviour is consistent. Access modifiers on types and assemblies provide encapsulation. They are: private, protected, internal, protected internal, and public.

Generic collections are used:

There is a very useful List collection. It is better than arraylist of objects in earlier .NET versions. Another useful collection is Dictionary¡tkey,tvalue vs. A non-generic hashtable of objects. We write "using System.Collections.Generic" to include these collections. List<int> myints = new List<int>();

```
Myints.Add(1);
Myints.Add(2);
Myints.Add(3);
For(int i= 0; i < myints.Count; i++)
{
Console.writeline("myints:{0}", myints[i]);
}
Public class Customer
{
Public Customer(int id, string name)
{
ID = id;
Name = name;
}
Public int ID
{
Get;
Set;
}
Public string Name
{
Get;
Set;
}
}
```

Chapter 10

Lecture 10

Anonymous methods

It is used with delegates and handlers and events. Anonymous methods result in much less code. Anonymous method is a method without a name. You don't declare anonymous methods, Instead

they get hooked up directly to events. With delegates

1) you declare the delegate,
2) write a method with a signature defined by the delegate interface,
3) declare the event based on that delegate, and
4) Write code to hook the handler method up to the delegate.

To delcare an anonymous method, you just use keyword "delegate" followed by method body.

Discuss debugging in Visual Studio

By printing output frequently we can debug. We can use Console.Write to print. Breakpoints allow stopping the program during execution. Press F5 and execution will stop at breakpoint. You can hover over variabeles for their current values. Locals, watch, call stack, immediate window give you different information about the program. Your changes will have effect on the current execution. Conditional breakpoints have a hit count and when hit condition.

XML

XML or extensible Mark-up Language is widely used for exchanging data. Its readable for both humans and machines. It's a stricter version of HTML. It's made of tags, attributes, and values.

There are two methods to read XML document Using:

- xmldocument
- xmlreader.

Xmldocuments reads entire document in memory, Lets you go forward, backward, even apply xpath searches on it.

Xmlreader is fast, uses less memory and provides one element at a time.

```
Using System;

Using System.Text;

Using System.Xml;

Namespace parsingxml

{

 Class Program

 {

 Static void Main(string[] args)

 {

 Xmlreader xmlreader =

xmlreader.Create("http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml");  While
(xmlreader.Read())

 {

 If ((xmlreader.nodetype == xmlnodetype.Element) && (xmlreader.Name ==  "Cube"))
```

```
{
If (xmlreader.hasattributes)
Console.writeline(xmlreader.getattribute("currency") + ": " +  xmlreader.getattribute("rate"));
}
}
Console.readkey();
}
}
}
```

Using xmldocument is easier. No order is required. Documentelement is the root element and childnodes is the set of children of any node.

```
Namespace parsingxml
{
Class Program
{
Static void Main(string[] args)
{
Xmldocument xmldoc = new xmldocument();
Xmldoc.Load("http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml"); Foreach (xmlnode xmlnode in xmldoc.documentelement.childnodes[2].childnodes[0].Child
Console.writeline(xmlnode.Attributes["currency"].Value + ": "+ xmlnode.Attribut Console.readkey();
}
}
}
```

An xmlnode is derived from xmlelement and contains Name, innertext, innerxml, outerxml, and Attributes. Xpath is a cross-platform Xml Query language. We will look at basic examples only. We will see xmldocument methods that take xpath queries and return xmlnode(s) in particular selectsinglenode and selectnodes.

```
Using System;
Using System.Text;
Using System.Xml;
Namespace parsingxml
{
```

```
Class Program
{
Static void Main(string[] args)
{
Xmldocument xmldoc = new xmldocument();
Xmldoc.Load("http://rss.cnn.com/rss/edition_world.rss");  Xmlnode titlenode =
xmldoc.selectsinglenode("//rss/channel/title");  If (titlenode != null)
Console.writeline(titlenode.innertext);
Console.readkey();
}
}
}
```

Chapter 11

Lecture 11

Example

Chapter 12

Lecture 12

WPF (Windows Presentation Foundations)

It was publicly announced in 2003 (codenamed Avalon). WPF 4 was released in April 2010. It has a steep learning curve. Code has to be written in many places. There are multiple ways to do a particular task.

WPF enables polished user interfaces which are getting a lot of attention. It enables rapid iterations and major interface changes throughout the development process. It allows keeping user interface description and implementation separate. Developers can create an "ugly" application which designers can re-theme. Win32 style of programming makes such re-theming difficult. The code to re-paint the user interface is mixed up with program logic. GDI was an earlier user interface library introduced in windows 1.0 in 1985. Opengl was a leap ahead introduced in the 90's with directx coming in 95 and directx 2 in 96. GDI+ is a newer user interface library based on directx. It is also used behind Xbox graphics. Next was Windows Forms which is the primary way of user interface design in C#. XNA comes with managed library for directx and is great for game development (.net/com interoperability not required). A simple example is drawing bitmaps on buttons which can be efficiently done using GDI.

The highlights of WPF are

1) Broad integration (2D, 3D, video, speech libraries etc.)
2) Resolution Independence with WPF giving emphasis on vector graphics
3) Hardware accelration as it is based on Direct3D but it can work using software pipeline also if Direct3D hardware is not available.
4) Declarative programming using extensible Application Markup Language (XAML; pronounced "Zammel"). Custom attribute and configuration files were always there but XAML is very rich.
5) Rich composition and customization e.g. you can create a combobox filled with animated Buttons or a Menu filled with live video clips! And it is quite easy to skin applications.

In short, WPF aims to combine the best attributes of systems such as directx (3D and hardware acceleration), Windows Forms (developer productivity), Adobe Flash (powerful animation support) and HTML (declarative mark-up). The first release in November 2006 was WPF 3.0 because it shipped as part of the .NET Framework 3.0. WPF 3.5 came a year later. Next version as part of .NET 3.5 SP1 came in August 2008. WPF Toolkit released in Aug 2008 was experimental. The toolkit has quick releases. Regarding tool support, WPF extensions for Visual Studio 2005 came a few months after the first WPF release and a public release of expression blend. Now, Visual Studio 2012 is a first class WPF development environment. It's mostly re-written using WPF and expression blend is 100% WPF and is great for designing and prototyping WPF apps.

New things that came in WPF 3.5/3.5SP1 include Interactive3D with 2d elements in 3d scenes, first class interoperability with directx, Better data binding using XLINQ and better validation and debugging which reduces code, Better special effects, High performance custom drawing, Text improvements, enhancements to Partial-trust apps, improved deployment, and improved performance.

Things that came with WPF 4.0 include multi-touch support - compatible with surface API v2, win7 support like jump lists, new common dialogs etc., new controls like datagrid, calendar etc, easing animation functions (bounce, elastic), enhanced styling with visual state manager, improved layout on pixel boundaries, non-blurry text but some limitations so must opt-in, deployment improvements, and performance improvements.

Silverlight in comparison is a light-weight version for web. It chose to follow WPF appraoch. First released in 2007 and in April 2010 4th version was released near WPF 4. There is often confusion when to use one or the other. Both can run in and outside the web. Silverlight is ostly a subset of WPF but there are some incompatibilities. Should I use full .net or partial .net and should I have the ability to run on other devices e.g. Macs. Ideally common codebase should work for both but today best case is using #ifdefs to handle incompatibilities.

XAML is primarily used to describe interfaces in WPF and Silverlight. It is also used to express activities and configurations in workflow foundation (WF) and windows communication foundation (WCF). It's a common language for programmers and other experts e.g. UI design experts. Field specific development tools can be made. Field experts are graphic designers. They can use a design tool such as expression blend. Other than co-ordinating with designers, XAML is good for a concise way to represent UI or hierarchies of objects, encourages separation of front-end and back-end, tool support with copy and paste, used by all WPF tools.

XAML is xml with a set of rules. It's a declrative programming language for creating and initializing objects. Itsa way to use .net apis.

Comparisons with SVG, HTML etc are misguided. It has few keywords, not much elemetns. It doesn't make sense without .net, like C# doesnt make sense without .net. Microsoft formalized XAML vocabluries, e.g. WPF XAML vocabluary.

Online specifications for XAML object specification language and WPF and Silverlight vocabluaries are available. XAML is used by other technologies as well although originally it was designed for WPF. Also using XAML in WPF projects is optional. Everything can be done in procedural code as well (althought it's rare to find it done that way).

Chapter 13

Lecture 13

XAML

XAML specification defines rules that map .NET namespaces, types, properties, and events into XML namespaces, elements, and attributes. Let's see XAML and equivalent C#.

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" Content="OK"/>
```

And the corresponding C# code is

```
System.Windows.Controls.Button b= new System.Windows.Controls.Button(); B.Content= "OK";
```

Declaring an XML element in XAML (known as an object element) is equivalent to instantiating the corresponding .NET object via a default constructor. Setting an attribute on the object element is equivalent to setting a property of the same name (called a property attribute) or hooking up an event handler of the same name (called an event attribute).

XAML can no longer run standalone in the browser because of the butto click method - event handlers are attached before properties are set.

Mapping to the namespace used above and other WPF namespaces is hard-coded in-side the WPF assemblies. It's just an arbitrary string like any namespace. The root object element in XAML must specify at least one XML namespace that is used to qualify itself and any child lements.

This is the XAML language namespace, which maps to types in the System.Windows.Markup namespace but also defines some special directives for the XAML compiler or parser. The period distinguishes property elements from object elements.

WPF provides type converters for many common data types: Brush, Color, fontweight, Point, and so on. Classes deriving from typeconverter (brushconverter, colorconverter, and so on) convert from string to the corresponding types.

You can also write your own type converters for custom data types.

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" Content="OK">
<Button.Background>
```

markup extensions

Markup extensions, like type converters, enable you to extend the expres-siveness of XAML. Both can evaluate a string attribute value at runtime (except for a few built-in mark-up extensions for performance reasons) and produce an appropriate object based on the string. As with type converters, WPF ships with several markup extensions built in.

Unlike type converters, however, markup extensions are invoked from XAML with explicit and consistent syntax. For this reason, using markup extensions is a d.

For example, if you want to set a control's background to a fancy gradient brush with a simple string value, you can write a custom markup extension that supports it even though the built-in brushconverter does not. Whenever an attribute value is enclosed in curly braces ({}), the XAML compiler/parser treats it as a markup extension value rather than a literal string (or something that needs to be type-converted).

It works because positional parameters have corresponding property value. Markup extension has the real code to be executed.

An object element can have three types of children:

- a value for a content property,
- collection items,
- Or a value that can be type-converted to the object element.

Designated property is the content property.

```
<Button xmlns=''http://schemas.microsoft.com/winfx/2006/xaml/presentation''  Content=''OK''/>
```

Could be rewritten as follows:

```
<Button xmlns=''http://schemas.microsoft.com/winfx/2006/xaml/presentation''> OK </Button>

Also

<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">  <Button.Content>

<Rectangle Height="40" Width="40" Fill="Black"/> </Button.Content>

</Button>
```

Could be rewritten as follows:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"> <Rectangle
Height="40" Width="40" Fill="Black"/>

</Button>
```

There is no requirement that the content property must actually be called Content; classes such as combobox, listbox, and tabcontrol (also in the System.Windows.Controls namespace) use their Items property as the content property. Its designated with a custom attribute.

XAML enables you to add items to the two main types of collections that support indexing: lists and dictionaries.

```
<listbox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">  <listbox.Items>

<listboxitem Content="Item 1"/>
```

```
<listboxitem Content="Item 2"/> </listbox.Items>
</listbox>
```

Chapter 14

Lecture 14

XAML works with classes

```
<solidcolorbrush>White</solidcolorbrush>
```

As discussed in the last lecture, this is equivalent to the following:

```
<solidcolorbrush Color="White"/>
```

It works because a typeconverter exists that converts string to solidcolorbrush, although no designated content property exists.

```
<Brush>White</Brush>
```

In fact, the above also works even although Brush is abstract. It works because of a type converter can convert a string to a solidcolorbrush.

XAML works with classes designated with marked attributes. Usually these classes have default ctors and useful properties. But what about other classes not designed for XAML. E.g. Hashtable.

```
System.Collections.Hashtable h
H.Add("key1", 7);
H.Add("key2", 23);
Heres how it can be represented in XAML:
<Collections: Hashtable
 Xmlns: collections = "clr-namespace: System.Collections; assembly = mscorlib"  Xmlns: sys = "clr-namespace: System; assembly = mscorlib"
 Xmlns: x = "
<Sys: Int32 x: Key = "key1"> 7 </ sys: Int32>
<Sys: Int32 x: Key = "key2"> 23 </ sys: Int32>
</ Collections: Hashtable>
```

XAML child element rules

If the type implements ilist, call ilist.Add for each child. Otherwise, if the type implements idictionary, call idictionary.Add for each child, using the x:Key attribute value for the key and the element for the value. (Although XAML2009 checks idictionary before ilist and supports other collection interfaces, as described earlier.)

If the parent supports a content property (indicated by System.Windows.Markup.content property attribute) and the type of the child is compatible with that property, treat the child as its value. Otherwise, if the child is plain text and a type converter exists to transform the child into the parent type (and no properties are set on the parent element), treat the child as the input to the type converter and use the output as the parent object instance. Otherwise, treat it as unknown content and potentially raise an error.

We can mix XAML and procedural code. Let's try this with xamlreader and xamlwriter.

```
Window window= null;
Using (filestream fs = new filestream("mywindow.xaml", filemode.Open, fileaccess.Read)) {
//Get the root element, which we know is a Window
Window= (Window)xamlreader.Load(fs);
}
```

Let's see how to grab the OK button by walking the children (with hard-coded knowledge!)

```
Stackpanel panel= (stackpanel)window.Content;
Button okbutton = (Button)panel.Children[4];
```

For easier fetching, we can name XAML elements.

```
<Button x:Name="okbutton">OK</Button>
```

Let's grab the OK button, knowing only its name

```
Button okbutton=(Button)window.findname("okbutton");
```

Compiling xaml involves 3 steps

- converting a XAML file into a special binaryformat,
- embedding the converted content asa binary resource in the assembly being built,
- And performing the plumbing that connects XAML with procedural code automatically.

C# and VB have best support in typical case; the first step is specifying a subclass for the root element in a XAML file. Use the Class keyword for that.

Chapter 15

Lecture 15

BAML

BAML is binary application markup language. It just a compressed representation of XAML. There is even a BAML reader available. Earlier there was CAML which stands for compiled application markup language but it’s not used now.

Some glue code is generated when we use x:Class. Its kind of same as loading and parsing the XAML file. We must call initializecomponent and we can refer named elements like class members. Procedural code can even be written inside the XAML file.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" x:Class="mynamespace.mywindow">
<Button Click="button_Click">OK</Button>

 <x:Code>

 <![CDATA[

 Void button_Click(object sender, routedeventargs e)

 {

 This.Close();

 }

 ]]>

 </x:Code>

</Window>
```

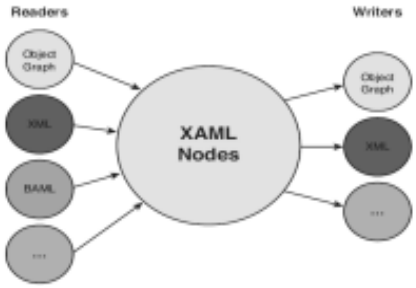
We must avoid]]. If we have to use them, we need &lt &amp etc. There is no good reason to embed code. Internally its build into .cs file by build system. BAML can be converted back to XAML.

```
System.Uri uri = new System.Uri("/wpfapplication1;component/mywindow.xaml",
System.urikind.Relative);

Window window= (Window)Application.loadcomponent(uri);

String xaml = xamlwriter.Save(window);
```

There are different loading mechanisms. We can load from resource, identified by original xaml file, or actually integrated baml loaded.

Key features of XAML 2009 include full generics support using typearguments, dictionary keys of any type, builtin system data types, creating objects with non-default ctors, getting instances via factory methods, event handlers can be markup extensions returning delegates, define additional members and properties.

System.Xaml.xamlxmlreader etc. Can be extended with readers and writers for a lot of formats. It abstracts differences in Xaml formats like accessing a content property, property element, or a property attribute.

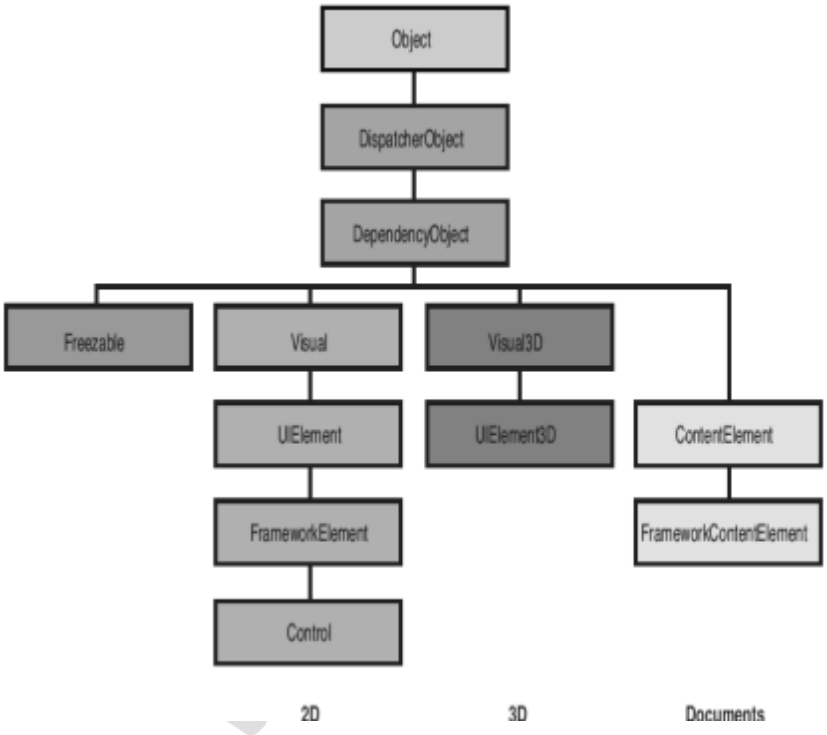


XAML 2006 keywords include x:asyncrecords - Controls the size of asynchronous XAML-loading chunks, x:Class, x:classmodifier - visibility, public by default, x:Code, x:connectionid - not for public use, x:fieldmodifier - field visibility (internal def), x:Key, x:Name, x:Shared - =false means same resource instance not shared, x:subclass - only needed when partial not supported, x:Synchronous mode - xaml loaded in async mode, x:typearguments - used only with root with x:Class in xaml2006, x:Uid - represents system.uri, and x:xdata - data opaque for xaml parser.

Markup extensions often confused as keywords are x:Array - use with x:Type to define type of array, x:Null, x:Reference - to named element, x:Static - static property field, and x:Type - just like typeof operator. In summary, we can convert xaml into C#, often apis are optimized for xaml and can look clunky in C#, wpf applications have deep hierarchy, small building blocks. Two complaints are that xml too verbose to type and slow to parse (tools and baml fix it partially).

wpf fundamentals

WPF concepts are above and beyond .net concepts. Its the cause of the steep learning curve of wpf. It has a deep inheritance hierarchy. There are a handful of fundamental classes.



Object is the base class for all .NET classes and the only class in the figure that isn't WPF specific.

Dispatcherobject is the base class meant for any object that wishes to be accessed only on the thread that created it. Most WPF classes derive from dispatcherobject and are therefore inherently thread-unsafe. The Dispatcher part of the name refers to wpfs version of a Win32-like message loop.

Dependencyobject is the base class for any object that can support dependency properties, one of the main topics in this chapter.

Freezable

Freezable is the base class for objects that can be "frozen" into a read-only state for performance reasons. Freezables, once frozen, can be safely shared among multiple threads, unlike all other dispatcherobjects. Frozen objects can never be unfrozen, but you can clone them to create unfrozen copies. Most Freezables are graphics primitives such as brushes, pens, and geometries or animation classes.

Visual is the base class for all objects that have their own 2D visual representation.

Uielement is the base class for all 2D visual objects with support for routed events, command binding, layout, and focus.

- These features are "Input Events: Keyboard, Mouse, Stylus, and Multi-Touc."
- Visual3D is the base class for all objects that have their own 3D visual representation.
- Uielement3d is the base class for all 3D visual objects with support for routed events, command binding, and focus.

Contentelement is a base class similar to uielement but for document-related pieces of content that don't have rendering behavior on their own. Instead, contentelements are hosted in a Visual-derived class to be rendered onthe screen. Each contentelement often requires multiple Visuals to render correctly (spanning lines, columns, and pages).

Frameworkelement is the base class that adds support for styles, data binding, resources, and a few common mechanisms for Windows-based controls, such as tooltips and context menus. Frameworkcontentelement is the analog to frameworkelement for content.

Control is the base class for familiar controls such as Button, listbox, and statusbar. Control adds many properties to its frameworkelement base class, such as Foreground, Background, and fontsize, as well as the ability to be completely restyled. Part III, "Controls," examines wpfs controls in depth.

Logical and visual trees

Xaml good for UI because of hierarchical nature. Logical tree exists even if there is no xaml. Properties, events, resources are tied to logical trees. Properties propagated down and events can be routed up or down the tree. It's a simplification of what's actually going on when rendered. Visual tree can be thought of as an extension of the logical tree though some things can be dropped as well. Visual tree exposes visual implementation details e.g. A listbox is a border, two scrollbars and more. Only things from visual or visual3d appear in a visual tree.

Visual tree is empty until the dialog box is rendered. Navigating either can be done in instance methods of the elements themselves e.g. Visual class has protected members visualparent, visualchildrencount, and getvisualchild. Frameworkelement and frameworkcontentelement have Parent property and logicalchildren property.

Chapter 16

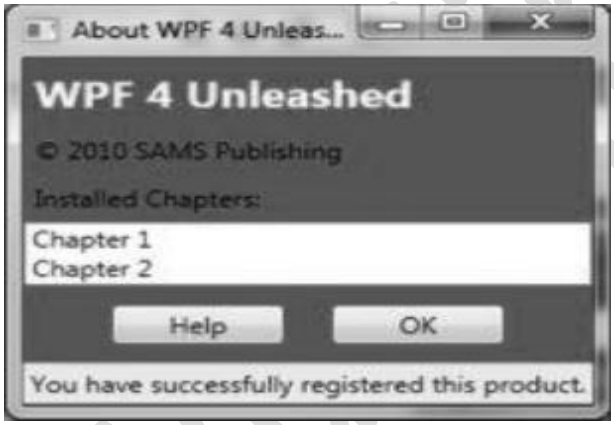
Lecture 16

Dependency properties

They complicate .net types but once you realize the problem it solves, you realize its importance. Dependency properties depend on multiple providers for determining its value at any point in time. These providers can be an animation continuously changing its values. A parent whose property propagates down. Arguably biggest feature is change notification. Motivation is to add rich functionality from declarative mark-up. Key to declarative-friendly design is heavy use of properties. Button e.g. has 111 public properties (98 inherited). Properties can be set using xaml, directly or in a design tool, without procedural code. Without the extra work in dependency properties, it would be hard. Implementation of a dependency property and then see its benefits on top of .net properties. Change notification, property value inheritance, and support for multiple providers are key features.

Understanding most nuances is important only for custom control authors. However what they are and how they work important for everyone. Can only style and animate dependency properties. After using wpf for a while, one wishes all properties were dependency properties. In practice, dependency properties are normal .net properties with extra wpf infrastructure. No .net language other than xaml has an intrinsic understanding of dependency properties.

Let's look at a standard dependency property implementation.




```
Public class Button: buttonbase

{

// The dependency property

Public static readonly dependencyproperty isdefaultproperty;

Static Button()

{

// Register the property

Button.isdefaultproperty = dependencyproperty.Register("isdefault", typeof(bool), typeof

}

// A .NET property wrapper(optional)
```

```
Public bool isdefault
{
Get
{
Return(bool)getvalue(Button.isdefaultproperty);
}
Set
{
Setvalue(Button.isdefaultproperty, value);
}
}
// A property changed callback(optional)
Private static void onisdefaultchanged(_dependencyobject o,
dependencypropertychangedeventar
{
//...
}
//...
}
```

Dependency properties

Dependency properties are represented by System.Windows.dependencyproperty. By convention, public static and Property suffix are used. Its required by several infrastructure pieces e.g. Localization tools, xaml loading. Optionally (via different overloads of Register), you can pass metadata that customizes how the property is treated by WPF, as well as callbacks for handling property value changes, coercing values, and validating values. .net prop wrapper optional. It helps setting from xaml and isnatural. Otherwise getvalue and setvalue are to be used. Getvalue and setvalue methods inherited from System.Windows.dependencyobject. Getvalue setvalue does not support generic. Dependency properties were introduced before generic support was added in C#.

Visual Studio has a snippet called propdp that automatically expands into a definition of a dependency property, which makes defining one much faster than doing all the typing yourself! .NET property wrappers are bypassed at runtime when setting dependency prop- erties in XAML! Although the XAML compiler depends on the property wrapper at compile time, WPF calls the underlying getvalue and setvalue methods directly at runtime! Therefore, to maintain parity between setting a property in XAML and procedural code, its crucial that property wrappers not contain any logic in addition to the getvalue/setvalue calls. If you want to add custom logic, thats what the registered callbacks are for. All of wpfs built-in property wrappers abide by this rule, so this warning is for anyone writing a custom class with its own dependency properties.

On the surface, it too much code but saves per-instance cost. Only static field and an efficient sparse storage system. If all were .net props with backing store, would consume lot more space e.g. 111

fields for button, 104 for label but 89 and 82 are dependency properties. Also code to check thread access, prompt containing element to be re-rendered.

Dependency properties support change notification. Its based on metadata at register time. Actions can be re-rendering the appropriate elements, updating the current layout, refreshing data bindings, and much property triggers - imagine you want color change on hovering.

It does not effect status bar. Not every dependency property participates in inheritance (Actually they can opt-in). May be other higher priority sources setting property value. Some controls like status bar internally set their values to system defaults. Property value inheritance in other places like to triggers inside a definition.

WPF contains many powerful mechanisms that independently attempt to set the value of dependency properties. Dependency properties were designed to depend on these providers in a consistent and orderly manner. It's a 5 step process

- Step 1: determine base value. Most providers factor into base value determination. Highest to lowest precedence. Ten providers that can set value for most dep.
  Props
  - Local valuedependencyobject.setvalue, prop assignment in xaml
  - Parent template trigger
  - Parent template
  - Style triggers
  - Template triggers
  - Style setters
  - Theme style triggers
  - Theme style setters
  - Property value inheritancealready seen
  - Default valueinitial value registered with the property.

  Thats why status bar did not get font propagated. Use dependencypropertyhelper.getvaluesource to find which was the source used.

- Step 2: evaluate. Expressions need evaluation. It is used in data binding.
- Step 3: apply animation. Animations can alter value from step 2 or replace it.
- Step 4: coerce. The almost final value passed to coercevaluecallback delegate if one is registered.

- Step 5: validate. Passed to validatevaluecallback delegate. If one was registered. Returning false causes exception canceling the entire process. Wpf 4 adds new value in dependencyobject called setcurrentvalue. It updates current value without changing value source.

Attached properties

They are special dependency properties that can be attached to arbitrary objects. Sounds strange but there are many applications for it. There is new xaml syntax for attached props.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"76
@ Copyright Virtual University of Pakistan
Visual Programming - CS411 VU
Title="About WPF4 Unleashed" sizetocontent="widthandheight"
Background="orangered">
<stackpanel>
<Label fontweight="Bold" fontsize="20" Foreground="White">
WPF4 Unleashed
</Label >
<Label> 2010 SAMS Publishing</Label>
<Label>Installed Chapters:</Label>
<listbox>
<listboxitem>Chapter 1</listboxitem>
<listboxitem>Chapter 2</listboxitem>
</listbox >
<stackpanel textelement.fontsize="30" textelement.fontstyle="Italic"  Orientation="Horizontal">
<Button minwidth="75" Margin="10">Help</Button>
<Button minwidth="75" Margin="10">OK</Button>
</stackpanel >
<statusbar>You have successfully registered this product.</statusbar> </stackpanel >
</Window >
```

Chapter 17

Lecture 17

Enumeration values

Enumeration values such as fontstyles.Italic, Orientation.Horizontal, and horizontalalignment.Center were pre-viously specified in XAML simply as Italic, Horizontal, and Center, respectively. This is possible thanks to the enumconverter type converter C#. C# code shows no real magic, no .net property involved internally. There are just calls to dependencyobject.setvalue and getvalue like normal property wrappers must not go anything else.

```
Public static void setfontsize(dependencyobject element, double value) {

Element.setvalue(textelement.fontsizeproperty, value);

}

Public static double getfontsize(dependencyobject element)

{

Return(double)element.getvalue(textelement.fontsizeproperty);

}
```

We are setting the fontsize property by unrelated class textelement. We could also have used textblock but textelement.fontsizeproperty is a separate dependencyproperty field from Control.fontsizeproperty.

```
Textelement.fontsizeproperty= dependencyproperty.registerattached(

"fontsize", typeof(double), typeof(textelement), new frameworkpropertymetadata(
Systemfonts.messagefontsize, frameworkpropertymetadataoptions.Inherits |
Frameworkpropertymetadataoptions.affectsrender |

Frameworkpropertymetadataoptions.affectsmeasure),

New validatevaluecallback(textelement.isvalidfontsize));

Using registerattached us optimized for attached property metadata. Control on the other hand just calls  addowner.

Control.fontsizeproperty = textelement.fontsizeproperty.addowner(

Typeof(Control), new frameworkpropertymetadata(systemfonts.messagefontsize,
Frameworkpropertymetadataoptions.Inherits));
```

These font related properties all controls inherit are from textelement in most cases, the class exposing the attached property is the same that defines the normal dependency propery. Many dependency properties have a Tag property for storing arbitrary custom data. A great mechanism for even extending "sealed" classes. In procedural code, you can actually use "any" property by calling setvalue.

```
// Attach an unrelated property to a Button and set its value to true:
Okbutton.setvalue(itemscontrol.istextsearchenabledproperty, true);

We can use this property any way we want but its better way to use Tag.
```

```
Geometrymodel3d model= new geometrymodel3d();
Model.setvalue(frameworkelement.tagproperty, "my custom data");
```

Its most commonly used for layout of UI elements. Various panel derived classes define attached properties designed to be attached to their children. This way each panel can define custom behaviour without burdening all possible children with their own set of properties. It allows extensibility.

In summary, wpf could have exposed its features by api's but instead they focused on adding features to the core. Multiple types of properties, multiple tree, multiple ways to achieve the same thing. Hopefully you can appreciate the value in coming lectures. These concepts will fade in the background, sizing, positioning, and transforming elements. Sizing and positioning called "layout".

Chapter 18

Lecture 18

Layout transform vs render transform.

One is applied before rendering and one after. Pointconverter is used to specify the origin.

```
<Button rendertransformorigin="0.5,0.5" Background="Orange">
<Button.rendertransform>
<rotatetransform Angle="45"/>
</Button.rendertransform >
Rotated 45
</Button >
```

Builtin transforms are Rotate Transform, Scale Transform, Skew Transform, Translate Transform, and Matrix Transform. Rotatetransform has Angle, centerx, centery with defaults 0. Center useless for layout transform. Center is also useful when grouping rendertransforms.

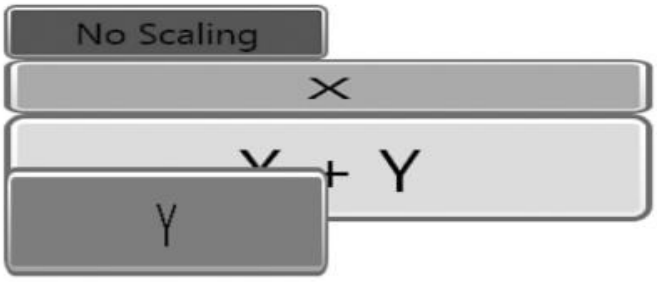
Transform can be on inner element.

```
<Button Background="Orange">
<textblock rendertransformorigin="0.5,0.5">
<textblock.rendertransform>
<rotatetransform Angle="45"/>
</textblock.rendertransform >
45
</textblock >
</Button >
```

Text rotation around the top-left corner

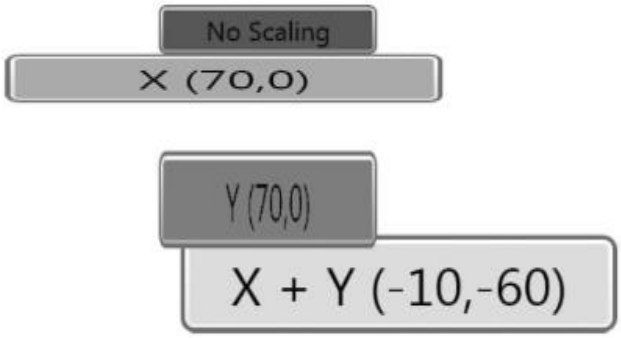
Text rotation around the middle

```
Scale Transform has scalex, scaley, centerx, centery.
<stackpanel Width="100">
 <Button Background="Red">No Scaling</Button>
 <Button Background="Orange">
<Button.rendertransform>
<scaletransform scalex="2"/>
</Button.rendertransform >
X
</Button >
 <Button Background="Yellow">
<Button.rendertransform>
<scaletransform scalex="2" scaley="2"/>
</Button.rendertransform >
X + Y
 </Button >
 <Button Background="Lime">
<Button.rendertransform>
<scaletransform scaley="2"/>
</Button.rendertransform >
Y
 </Button> 21
</stackpanel >
```

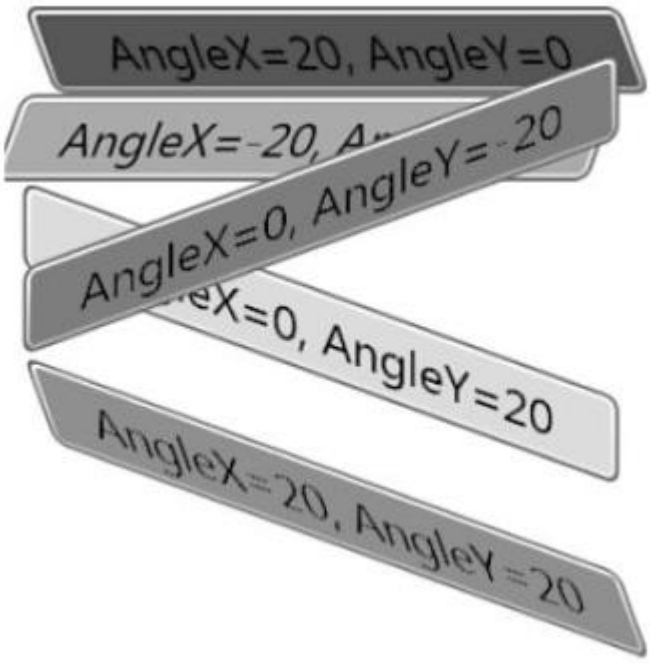


Stretch and scaletransform only effect if more than stretch. Padding is scaled but margin is not. Does not affect actualhieght actualwidth rendersize etc.

Skew Transform has anglex — Amount of horizontal skew (default value = 0), angley — Amount of vertical skew



(default value = 0), centerx — Origin for horizontal skew (default value = 0), centery — Origin for vertical skew (default value = 0). It is applied as a render transform here.



Translate Transform has X — Amount to move horizontally (default value = 0), Y — Amount to move vertically (default value = 0). It has no effect as a layout transform.

Matrix Transform has a single Matrix property (of type System.Windows.Media.Matrix) representing a

3x3 affine transformation matrix.

<Button rendertransform=1,0,0,1,10,20/>

$$\begin{bmatrix} M11 & M12 & 0 \\ M21 & M22 & 0 \\ OffsetX & OffsetY & 1 \end{bmatrix}$$

Its the only transform with a type converter to convert a string. Combining transforms can be done. We can apply both layout and render transforms. Can figure out a combined matrix and apply matrix transform or use transformgroup a Transform derived class and let it be calculated for you.

```
<Button>
 <Button.rendertransform>
<transformgroup>
<rotatetransform Angle="45"/>
<scaletransform scalex="5" scaley="1"/>
<skewtransform anglex="30"/>
</transformgroup >
 </Button.rendertransform >
 OK
</Button >
```

Not all elements can be transformed.

We learned about layout of individual elements. The layout is finally decided by parent. We have used stackpanel. Let's see this and other panels for layout. Static pixel based placement and sizes results in limited but different screen sizes. Builtin panels that make it easy.

5 main builtin panels in System.Windows.Controls:

- Canvas,
- stackpanel,

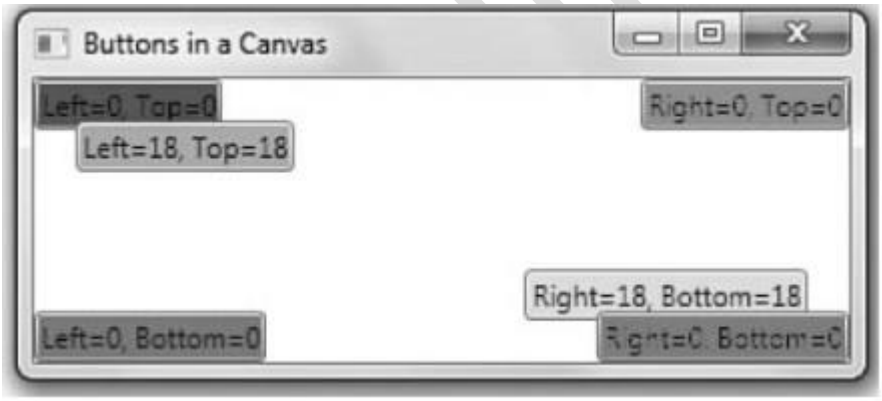
- wrappanel,
- dockpanel,
- Grid

Content overflow is when parents and children can't agree on the use of available space.

Canvas is the most basic. Probably never used. "classic" notion of explicit coordinates but device independent pixels and relative to any corner of the window. We can position elements in a Canvas by using

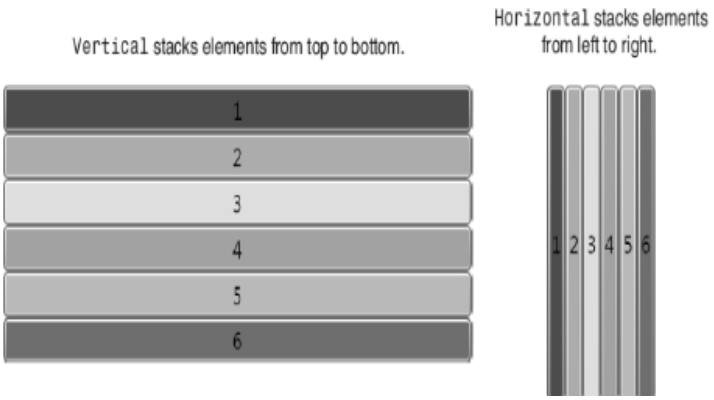
its attached properties: Left, Top, Right, and Bottom. Default offsets are from top left.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
Title="Buttons in a Canvas">
 <Canvas>
<Button Background="Red">Left=0, Top=0</Button>
<Button Canvas.Left="18" Canvas.Top="18"
Background="Orange"> Left=18, Top=18</Button>
<Button Canvas.Right="18" Canvas.Bottom="18"
Background="Yellow">Right=18, Bottom=18</Button>
<Button Canvas.Right="0" Canvas.Bottom="0"
Background="Lime">Right=0, Bottom=0</Button>
<Button Canvas.Right="0" Canvas.Top="0"
Background="Aqua">Right=0, Top=0</Button>
<Button Canvas.Left="0" Canvas.Bottom="0"
Background="Magenta">Left=0, Bottom=0</Button>
 </Canvas ></Window >
```

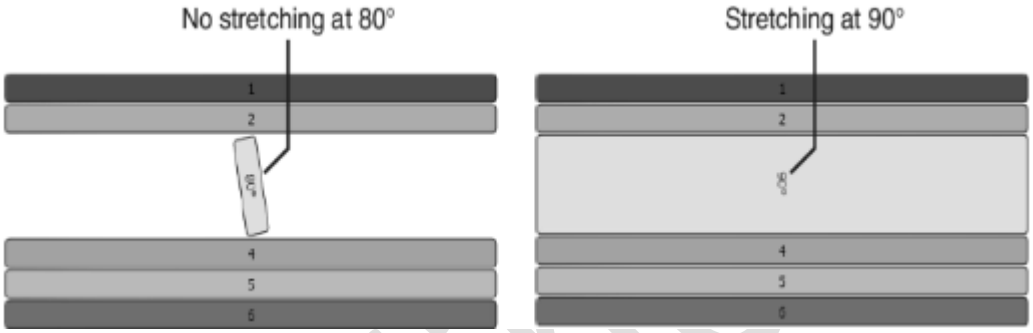


If left and right, right is ignored, top and bottom, bottom is ignored. Whats z order. Z-order decides which element to show in overlapping elements.

```
<Canvas>

 <Button Canvas.zindex="1" Background="Red">On Top!</Button>

 <Button Background="Orange">On Bottom with a Default zindex=0</Button>

</Canvas >
```



Stack Panel is popular, simple, useful, stacks sequentially. It is one of the few panels that dont even define an attached property. Orientation can be horizontal or vertical (which is default). Default horizontal direction is basedon flowdirection.



Virtualizing panels e.g. Virtualizingstackpanel save space for offscreen content when data binding e.g. Listbox uses it Wrap panel wraps to additional rows or columns when not enough space. It has no attached properties. Orientation is by default horizontal. Itemheight, itemwidth are uniform for all children and not set by default.

Best of luck