Using WCF

Working With Windows Communication Foundation

garth@ggilmour.com

Introducing WCF

- Most modern applications are n-tier
  - Your code lives on multiple machines
    - Not including the database layer
  - Code on these machines must interact
    - Objects in CLR 1 need to call methods of objects in CLR 2
    - Distributed method calls are inherently unreliable
- There are many different options for remote calls
  - Synchronous verses asynchronous
  - Binary verses text based messaging
  - XML, JSON and binary serialization
  - Proprietary verses open protocols

Introducing WCF

- Windows has supported many remoting technologies
  - DCOM, MSMQ, Remoting, ASP .NET Web Services etc...
  - However each of these has its own runtime and rules
- WCF unifies these different approaches
  - It provides a single platform for distributed systems
    - Covering all the common types of interaction
  - Developers can concentrate on what they want to do
    - As opposed to picking or learning a technology stack
    - Configuration can be done in code or via XML
  - WCF is available from .NET 3.0 onwards
    - Visual Studio 2008 provides project types and wizards

Introducing WCF

- WCF is based around the ‘ABC’ mnemonic:
  - The Address the service is available at
  - Bindings to different protocols and services
  - The Contract that defines what operations are available
- The framework makes assumptions about usage
  - The contract will normally be specified via WSDL
  - Pre-configured bindings are supplied for industry standards like WS-I compliant Web Services
- Client side development remains straightforward
  - Point a Visual Studio wizard at the address and it consumes the contract by creating a proxy class

Contracts in WCF

- WCF distinguishes three types of contract
  - A Service Contract maps methods to operations
  - A Data Contract maps CLR types to XML Schema types
  - A Message Contract specifies how encoded types are inserted into messages (such as SOAP Envelopes)
- Contracts are defined via attributes on classes
  - They are ultimately represented within the WSDL
    - The implementing class is marked with ‘ServiceContract’
    - Individual methods are marked with ‘OperationContract’
- The contract and the binding are independent
  - In general each type of contract works on each binding

Creating a WCF Application

- We will create a Pizza Shop application
  - Adding extra features with each version
- We will need to create four items:
  - An interface that defines the service contract
  - An implementation of the interface methods
  - A console program that will host the service
  - A client side application with a generated proxy
- The server-side object will be configured as per-call
  - This means that a new object is created for each request
  - In the same way that ASP .NET pages are managed

```
[ServiceContract]
interface IPizzaShop {
    [OperationContract]
    void OrderPizza(string pizza);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class PizzaShop : IPizzaShop {
    public PizzaShop() {
        Console.WriteLine("PizzaShop instance created...");
    }
    public void OrderPizza(string pizza) {
        Console.WriteLine("Received Order For: {0}", pizza);
    }
}
```

```
class Program {
        static void Main(string[] args) {
            //The endpoint of the service
            string location = "http://localhost:8123/PizzaShopOne";

            //A ServiceHost object contains the infrastructure to host a WCF service
            ServiceHost host = new ServiceHost(typeof(PizzaShop), new Uri(location));

            //Our binding will be SOAP'y with support for WS-*
            host.AddServiceEndpoint(typeof(IPizzaShop),new WSHttpBinding(),"");

            ServiceMetadataBehavior behavior = new ServiceMetadataBehavior();
            behavior.HttpGetEnabled = true;
            host.Description.Behaviors.Add(behavior);

            host.AddServiceEndpoint(typeof(IMetadataExchange),
                                        MetadataExchangeBindings.CreateMexHttpBinding(), "mex");

            host.Open();
            Console.WriteLine("Service Host running - hit enter to end");
            Console.ReadLine();
            host.Close();
            Console.WriteLine("All done - bye...");
        }
}
```

```
//Proxy auto-generated by Visual Studio
class Client {
    static void Main(string[] args) {
        PizzaShopClient client = new PizzaShopClient();

        client.OrderPizza("Pepperoni");
        client.OrderPizza("Quatro Stagioni");
        client.OrderPizza("Mighty Meaty");

        Console.WriteLine("Just ordered three pizzas...");
        client.Close();
    }
}
```

Metadata Exchange (MEX) in WCF

- When you create a new service meta-information about it is not automatically published
  - A client could use the service if they already knew the endpoint address, operation names and message formats
  - Normally you want this information to be published so proxies can be generated by other teams as required
- To support discovery a service should publish a Metadata Exchange (MEX) endpoint
  - You declare the endpoint in your code or XML configuration and the metadata is generated for you
    - You also have to explicitly allow access via HTTP GET
  - The metadata is published as a WSDL document

Introducing Behaviors

- WCF has a built in extension point called behaviours
  - Behaviors intercept events, such as the start-up and shutdown of the host or the arrival of a message
- There are three types of behaviours
  - Service behaviours run across all endpoints
  - Endpoint behaviours apply to a single service
  - Operation behaviours apply to a single operation
- Behaviors are used to implement core WCF functions
  - Including concurrency, security and transactions
  - Concurrency is an example of a service behavior
  - Transactions are an example of an operation behavior

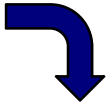
Instancing and Concurrency

- Instancing determines how many copies of the server side object are created as requests arrive
  - In ‘Single’ mode one instance handles all requests
  - In ‘PerCall’ mode an instance is created on each request
  - In ‘PerSession’ mode an instance is created for each client
- Concurrency determines how many threads are allowed to use at instance at once
  - In ‘Single’ mode only one thread is allowed
  - In ‘Reentrant’ mode again only one thread is allowed
    - But it can visit the instance many times
  - In ‘Multiple’ mode many threads can access the instance

The Per-Session WCF Pizza Shop

- Per-Session is the default instance value
  - As long as it is supported by the selected binding
  - Otherwise it degrades to per-call
- The WCF client needs to know a session is in use
  - Hence the interface needs to be marked as below:
  - '[ServiceContract(SessionMode=SessionMode.Required)]'
- The instance ends when the client calls 'Close'
  - If the client fails to do so then it times out after 10 min
    - Any further calls by the client will result in an exception
  - You can adjust this value by reconfiguring the binding

```
[ServiceContract(SessionMode=SessionMode.Required)]
interface IPizzaShop {
    [OperationContract]
    void OrderPizza(string pizza);
    [OperationContract]
    List<string> GetOrder();
    [OperationContract]
    double PriceOrder();
}
```



```
[ServiceBehavior(InstanceContextMode=
                        InstanceContextMode.PerSession)]
class PizzaShop : IPizzaShop {
    public PizzaShop() {
        pizzas = new List<string>();
        Console.WriteLine("PizzaShop instance created...");
    }
    public void OrderPizza(string pizza) {
        Console.WriteLine("Received Order For: {0}", pizza);
        pizzas.Add(pizza);
    }
    public List<string> GetOrder() {
        return pizzas;
    }
    public double PriceOrder() {
        return 12.50;
    }
    private List<string> pizzas;
}
```

```
class Program {
    static void Main(string[] args) {
        string location = "http://localhost:8123/PizzaShopTwo";
        ServiceHost host = new ServiceHost(typeof(PizzaShop), new Uri(location));

        WSHttpBinding binding = new WSHttpBinding();
        host.AddServiceEndpoint(typeof(IPizzaShop), binding, "");

        ServiceMetadataBehavior behavior = new ServiceMetadataBehavior();
        behavior.HttpGetEnabled = true;

        host.Description.Behaviors.Add(behavior);

        host.AddServiceEndpoint(typeof(IMetadataExchange),
                                        MetadataExchangeBindings.CreateMexHttpBinding(), "mex");

        host.Open();

        Console.WriteLine("Service Host running - hit enter to end");
        Console.ReadLine();

        host.Close();
        Console.WriteLine("All done - bye...");
    }
}
```

```
class Client {
    static void Main(string[] args) {
        PizzaShopClient client = new PizzaShopClient();

        client.OrderPizza("Pepperoni");
        client.OrderPizza("Quatro Stagioni");
        client.OrderPizza("Mighty Meaty");

        Console.WriteLine("Just placed order for:");
        foreach (string pizza in client.GetOrder()) {
                Console.WriteLine("\t{0}",pizza);
        }
        Console.WriteLine("At a total cost of: £{0}", client.PriceOrder());
        client.Close();
    }
}
```

The Singleton WCF Pizza Shop

- When a service is configured as a singleton only one instance is created and used by all clients
  - Even if the service is published to multiple endpoints
  - It is very important to select the right threading behavior
- The singleton instance lives forever
  - It is only destroyed when the host is shut down
  - If the contract of the service requires a session one will be created, and will never expire unless explicitly closed
- WCF lets you create the singleton yourself
  - The constructor of ‘ServiceHost’ is overloaded to accept a service-side object configured as a singleton

```
[ServiceContract]
interface IPizzaShop {
    [OperationContract]
    void CreateAccount(string clientID);

    [OperationContract]
    void CloseAccount(string clientID);

    [OperationContract]
    void OrderPizza(string clientID, string pizza);

    [OperationContract]
    List<string> GetOrder(string clientID);

    [OperationContract]
    double PriceOrder(string clientID);
}
```

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class PizzaShop : IPizzaShop {
    public PizzaShop() {
        orders = new Dictionary<string, List<string>>();
        Console.WriteLine("PizzaShop instance created...");
    }
    public void CreateAccount(string clientID) {
        orders.Add(clientID, new List<string>());
    }
    public void CloseAccount(string clientID) {
        orders.Remove(clientID);
    }
    public void OrderPizza(string clientID,string pizza) {
        Console.WriteLine("Received Order From {0} For: {1}", clientID, pizza);
        orders[clientID].Add(pizza);
    }
    public List<string> GetOrder(string clientID) {
        return orders[clientID];
    }
    public double PriceOrder(string clientID) {
        return 12.50;
    }
    private Dictionary<string, List<string>> orders;
}
```

```
class Program {
    static void Main(string[] args) {
        string location = "http://localhost:8123/PizzaShopThree";
        ServiceHost host = new ServiceHost(typeof(PizzaShop), new Uri(location));

        WSHttpBinding binding = new WSHttpBinding();
        host.AddServiceEndpoint(typeof(IPizzaShop), binding, "");

        ServiceMetadataBehavior behavior = new ServiceMetadataBehavior();
        behavior.HttpGetEnabled = true;

        host.Description.Behaviors.Add(behavior);

        host.AddServiceEndpoint(typeof(IMetadataExchange),
                                    MetadataExchangeBindings.CreateMexHttpBinding(), "mex");

        host.Open();

        Console.WriteLine("Service Host running - hit enter to end");
        Console.ReadLine();

        host.Close();
        Console.WriteLine("All done - bye...");
    }
}
```

```
class Client {
    static void Main(string[] args) {
        PizzaShopClient client = new PizzaShopClient();
        client.CreateAccount("AB12");
        client.CreateAccount("CD34");

        client.OrderPizza("AB12", "Capricciosa");
        client.OrderPizza("CD34", "American Hot");
        client.OrderPizza("AB12", "Siciliana");
        client.OrderPizza("CD34", "Diavolo");
        client.OrderPizza("AB12", "Quattro Formaggi");
        client.OrderPizza("CD34", "Quattro Stagioni");

        Console.WriteLine("Pizzas For AB12 Are:");
        foreach (string pizza in client.GetOrder("AB12")) {  Console.WriteLine("\t{0}", pizza); }
        Console.WriteLine("At a total cost of: £{0}", client.PriceOrder("AB12"));

        Console.WriteLine("Pizzas For CD34 Are:");
        foreach (string pizza in client.GetOrder("CD34")) { Console.WriteLine("\t{0}", pizza); }
        Console.WriteLine("At a total cost of: £{0}", client.PriceOrder("CD34"));

        client.CloseAccount("AB12");
        client.CloseAccount("CD34");
        client.Close();
    }
}
```

WCF and Architecture

- Using singletons creates chokepoints in your code
  - The single instance will be heavily threaded and the synchronization required will degrade performance
- The per-session option is very attractive
  - But as each client requires a dedicated instance there is a limit on the number that can be supported
- Large scale distributed applications are stateless
  - Or rather they push state management away from the font end of the application and onto the database layer
  - In WCF this means that server objects will be per-call
    - Clients will pass a unique identifier with each message
    - Special tables will be used to cache temporary data in the DB

```
[ServiceContract]
interface IPizzaShop {
    [OperationContract]
    void CreateAccount(string clientID);

    [OperationContract]
    void CloseAccount(string clientID);

    [OperationContract]
    void OrderPizza(string clientID, string pizza);

    [OperationContract]
    List<string> GetOrder(string clientID);

    [OperationContract]
    double PriceOrder(string clientID);
}
```

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class PizzaShop : IPizzaShop {
    public PizzaShop() {
        Console.WriteLine("PizzaShop instance created...");
    }
    public void CreateAccount(string clientID) {
        GetManager().AddAccount(clientID);
    }
    public void CloseAccount(string clientID) {
        GetManager().RemoveAccount(clientID);
    }
    public void OrderPizza(string clientID, string pizza) {
        Console.WriteLine("Received Order From {0} For: {1}", clientID, pizza);
        GetManager().AddToOrder(clientID, pizza);
    }
    public List<string> GetOrder(string clientID) {
        return GetManager().GetOrder(clientID);
    }
    public double PriceOrder(string clientID) {
        return 12.50;
    }
    private CurrentOrdersManager GetManager() {
        return Program.Manager;
    }
}
```

```
public class CurrentOrdersManager {
    private Dictionary<string, List<string>> orders;
    public CurrentOrdersManager() {
        orders = new Dictionary<string, List<string>>();
    }
    public void AddAccount(string clientID) {
        orders.Add(clientID, new List<string>());
    }
    public void RemoveAccount(string clientID) {
        orders.Remove(clientID);
    }
    public void AddToOrder(string clientID, string pizza) {
        orders[clientID].Add(pizza);
    }
    public List<string> GetOrder(string clientID) {
        return orders[clientID];
    }
}
```

```
public class Program {
    private static CurrentOrdersManager manager;
    public static Program() {
        manager = new CurrentOrdersManager();
    }
    public static CurrentOrdersManager Manager {
        get { return manager; }
    }
    //continued...
```

```
    //class 'Program' continued…

    static void Main(string[] args) {
        string location = "http://localhost:8123/PizzaShopFour";
        ServiceHost host = new ServiceHost(typeof(PizzaShop), new Uri(location));

        WSHttpBinding binding = new WSHttpBinding();
        host.AddServiceEndpoint(typeof(IPizzaShop), binding, "");

        ServiceMetadataBehavior behavior = new ServiceMetadataBehavior();
        behavior.HttpGetEnabled = true;

        host.Description.Behaviors.Add(behavior);

        host.AddServiceEndpoint(typeof(IMetadataExchange),
                                MetadataExchangeBindings.CreateMexHttpBinding(), "mex");

        host.Open();
        Console.WriteLine("Service Host running - hit enter to end");
        Console.ReadLine();
        host.Close();
        Console.WriteLine("All done - bye...");
    }
}
```

```
class Client {
    static void Main(string[] args) {
        PizzaShopClient client = new PizzaShopClient();
        client.CreateAccount("AB12");
        client.CreateAccount("CD34");

        client.OrderPizza("AB12", "Capricciosa");
        client.OrderPizza("CD34", "American Hot");
        client.OrderPizza("AB12", "Siciliana");
        client.OrderPizza("CD34", "Diavolo");
        client.OrderPizza("AB12", "Quattro Formaggi");
        client.OrderPizza("CD34", "Quattro Stagioni");

        Console.WriteLine("Pizzas For AB12 Are:");
        foreach (string pizza in client.GetOrder("AB12")) { Console.WriteLine("\t{0}", pizza); }
        Console.WriteLine("At a total cost of: £{0}", client.PriceOrder("AB12"));

        Console.WriteLine("Pizzas For CD34 Are:");
        foreach (string pizza in client.GetOrder("CD34")) { Console.WriteLine("\t{0}", pizza); }
        Console.WriteLine("At a total cost of: £{0}", client.PriceOrder("CD34"));

        client.CloseAccount("AB12");
        client.CloseAccount("CD34");
        client.Close();
    }
}
```

Using the App Configuration File

- Endpoints can be declared in the configuration file
  - Either ‘Web.config’ or ‘App.config’ depending on whether you code will be hosted in WAS or IIS
  - This makes it possible to change the endpoint address, bindings and behaviours at deployment time
- WCF has a very flexible architecture:
  - A contract can be referenced by many endpoints
    - And each endpoint could have a different binding
    - E.g. TCP/binary for speed and HTTP/SOAP for interoperability
  - Several endpoints, with the same binding but different contracts, can be located at the same address
    - Thereby appearing to the client as the same entity

The Pizza Shop Using ‘App.config’

```
public class Program {
    private static CurrentOrdersManager manager;
    static Program() {
        manager = new CurrentOrdersManager();
    }
    public static CurrentOrdersManager Manager {
        get {
                    return manager;
        }
    }
    static void Main(string[] args) {
        ServiceHost host = new ServiceHost(typeof(PizzaShop));

        host.Open();
        Console.WriteLine("Service Host running - hit enter to end");
        Console.ReadLine();
        host.Close();
        Console.WriteLine("All done - bye...");
    }
}
```

```
<configuration>
  <system.serviceModel>

    <services>
      <service name="PizzaShopV5.PizzaShop" behaviorConfiguration="behaviorOne">
        <host>
          <baseAddresses>
              <add baseAddress="http://localhost:8123/PizzaShopFive"/>
          </baseAddresses>
        </host>
        <endpoint address="" binding="wsHttpBinding" contract="PizzaShopV5.IPizzaShop"/>
        <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
      </service>
    </services>

    <behaviors>
      <serviceBehaviors>
        <behavior name="behaviorOne">
          <serviceMetadata httpGetEnabled="True"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>

  </system.serviceModel>
</configuration>
```

Controlling the Generated WSDL

- By default the symbols used in the generated WSDL are the same as those in your code
  - The port and service are named after your class
  - The operations are named after your methods
- Ideally the names should be independent
  - So your code can change without breaking compatibility
- Symbol names can be changed via attributes
  - ‘Name’, ‘Action’ and ‘ReplyAction’ on ‘OperationContract’
  - ‘Name’ and ‘Namespace’ on ‘ServiceContract’
  - The ‘ServiceBehavior’ attribute

Pizza Shop with Customized WSDL

- The following slides show:
  - The WSDL portType and SOAP bindings
    - For our existing Pizza Shop application
  - A new contract interface with extra information added
    - Via properties of the attributes
  - The altered WSDL portType and SOAP bindings
  - The proxy built by VS based on the changed WSDL
- Note the items that have been changed
  - The name and namespace URL of the port
  - The names of the operations in the port
  - The URL's used by WS-Addressing
    - We have only changed the URL's of input messages
    - We could have changed the output ones via 'ReplyAction'

```
<wsdl:portType name="IPizzaShop">
    <wsdl:operation name="OrderPizza">
        <wsdl:input wsaw:Action="http://tempuri.org/IPizzaShop/OrderPizza"
                    message="tns:IPizzaShop_OrderPizza_InputMessage"/>
        <wsdl:output wsaw:Action="http://tempuri.org/IPizzaShop/OrderPizzaResponse"
                    message="tns:IPizzaShop_OrderPizza_OutputMessage"/>
    </wsdl:operation>
    <wsdl:operation name="GetOrder">
        <wsdl:input wsaw:Action="http://tempuri.org/IPizzaShop/GetOrder"
                    message="tns:IPizzaShop_GetOrder_InputMessage"/>
        <wsdl:output wsaw:Action="http://tempuri.org/IPizzaShop/GetOrderResponse"
                    message="tns:IPizzaShop_GetOrder_OutputMessage"/>
    </wsdl:operation>
    <wsdl:operation name="PriceOrder">
        <wsdl:input wsaw:Action="http://tempuri.org/IPizzaShop/PriceOrder"
                    message="tns:IPizzaShop_PriceOrder_InputMessage"/>
        <wsdl:output wsaw:Action="http://tempuri.org/IPizzaShop/PriceOrderResponse"
                    message="tns:IPizzaShop_PriceOrder_OutputMessage"/>
    </wsdl:operation>
</wsdl:portType>
```

```
<wsdl:binding name="WSHttpBinding_IPizzaShop" type="tns:IPizzaShop">
    <wsp:PolicyReference URI="#WSHttpBinding_IPizzaShop_policy"/>
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="OrderPizza">
        <soap12:operation soapAction="http://tempuri.org/IPizzaShop/OrderPizza" style="document"/>
        <wsdl:input>
            <wsp:PolicyReference URI="#WSHttpBinding_IPizzaShop_OrderPizza_Input_policy"/>
            <soap12:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <wsp:PolicyReference URI="#WSHttpBinding_IPizzaShop_OrderPizza_output_policy"/>
            <soap12:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetOrder">
        <soap12:operation soapAction="http://tempuri.org/IPizzaShop/GetOrder" style="document"/>
        <wsdl:input>
            <wsp:PolicyReference URI="#WSHttpBinding_IPizzaShop_GetOrder_Input_policy"/>
            <soap12:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <wsp:PolicyReference URI="#WSHttpBinding_IPizzaShop_GetOrder_output_policy"/>
            <soap12:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

```
[ServiceContract(Name="CustomWsdlPizzaShop",Namespace="http://nowhere.com")]
interface IPizzaShop {

    [OperationContract(Name="create_account",Action="http://custom/create_account")]
    void CreateAccount(string clientID);

    [OperationContract(Name = "close_account", Action = "http://custom/close_account")]
    void CloseAccount(string clientID);

    [OperationContract(Name = "order_pizza", Action = "http://custom/order_pizza")]
    void OrderPizza(string clientID, string pizza);

    [OperationContract(Name = "get_order", Action = "http://custom/get_order")]
    List<string> GetOrder(string clientID);

    [OperationContract(Name = "price_order", Action = "http://custom/price_order")]
    double PriceOrder(string clientID);
 }
```

```
<wsdl:portType name="CustomWsdlPizzaShop">
    <wsdl:operation name="order_pizza">
        <wsdl:input wsaw:Action="http://custom/order_pizza"
                    message="tns:CustomWsdlPizzaShop_order_pizza_InputMessage"/>
        <wsdl:output wsaw:Action="http://nowhere.com/CustomWsdlPizzaShop/order_pizzaResponse"
                    message="tns:CustomWsdlPizzaShop_order_pizza_OutputMessage"/>
    </wsdl:operation>
    <wsdl:operation name="get_order">
        <wsdl:input wsaw:Action="http://custom/get_order"
                    message="tns:CustomWsdlPizzaShop_get_order_InputMessage"/>
        <wsdl:output wsaw:Action="http://nowhere.com/CustomWsdlPizzaShop/get_orderResponse"
                     message="tns:CustomWsdlPizzaShop_get_order_OutputMessage"/>
    </wsdl:operation>
    <wsdl:operation name="price_order">
        <wsdl:input wsaw:Action="http://custom/price_order"
                    message="tns:CustomWsdlPizzaShop_price_order_InputMessage"/>
        <wsdl:output wsaw:Action="http://nowhere.com/CustomWsdlPizzaShop/price_orderResponse"
                     message="tns:CustomWsdlPizzaShop_price_order_OutputMessage"/>
    </wsdl:operation>
</wsdl:portType>
```

```
<wsdl:binding name="WSHttpBinding_CustomWsdlPizzaShop" type="i0:CustomWsdlPizzaShop">
  <wsp:PolicyReference URI="#WSHttpBinding_CustomWsdlPizzaShop_policy"/>
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="order_pizza">
    <soap12:operation soapAction="http://custom/order_pizza" style="document"/>
    <wsdl:input>
      <wsp:PolicyReference URI="#WSHttpBinding_CustomWsdlPizzaShop_order_pizza_Input_policy"/>
      <soap12:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference URI="#WSHttpBinding_CustomWsdlPizzaShop_order_pizza_output_policy"/>
      <soap12:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="get_order">
    <soap12:operation soapAction="http://custom/get_order" style="document"/>
    <wsdl:input>
      <wsp:PolicyReference URI="#WSHttpBinding_CustomWsdlPizzaShop_get_order_Input_policy"/>
      <soap12:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference URI="#WSHttpBinding_CustomWsdlPizzaShop_get_order_output_policy"/>
      <soap12:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

```
class Program {
    static void Main(string[] args) {
        CustomWsdlPizzaShopClient client = new CustomWsdlPizzaShopClient();
        client.create_account("AB12");
        client.create_account("CD34");

        client.order_pizza("AB12", "Capricciosa");
        client.order_pizza("CD34", "American Hot");
        client.order_pizza("AB12", "Siciliana");
        client.order_pizza("CD34", "Diavolo");
        client.order_pizza("AB12", "Quattro Formaggi");
        client.order_pizza("CD34", "Quattro Stagioni");

        Console.WriteLine("Pizzas For AB12 Are:");
        foreach (string pizza in client.get_order("AB12")) { Console.WriteLine("\t{0}", pizza); }
        Console.WriteLine("At a total cost of: £{0}", client.get_order("AB12"));

        Console.WriteLine("Pizzas For CD34 Are:");
        foreach (string pizza in client.get_order("CD34")) { Console.WriteLine("\t{0}", pizza); }
        Console.WriteLine("At a total cost of: £{0}", client.get_order("CD34"));

        client.close_account("AB12");
        client.close_account("CD34");
        client.Close();
    }
}
```

One-Way Messaging

- By default a WCF service is synchronous
  - Client code must block and wait whilst the request is processed on the server and a response returned
  - The request may have been sent in a separate thread, but this is a client-side implementation detail
- An operation can be marked as one-way
  - Via '[OperationContract(IsOneWay = true)]'
  - The client receives an acknowledgement that the request has arrived at the server and then carries on
    - It does not wait for processing to occur on the server
  - The binding will be optimized for this case

Duplex Service Contracts

- A duplex contract is similar to request-response
  - Except that a response will not directly follow a request
  - There may be a significant delay or a single response may be sent in reply to a sequence of requests
- The contract allows messages to flow each way
  - Without being solicited by the other participant
- The client contract is determined by the service
  - The developer of the client generates a proxy as usual
- Each party will need its own address
  - For some bindings this may require a second channel

Duplex Contract in the Pizza Shop

- Assume the Pizza Shop required a Duplex Contract
  - Clients order any number of pizzas via a one-way method
  - Once the order is complete they are ‘cooked’ in a separate thread and the client is notified when they are ready
- In order to accomplish this:
  - The binding needs to change to ‘wsDualHttpBinding’
  - The service contract needs to specify a callback interface
    - We need to declare this with one or more callback methods
  - Start a separate thread to communicate with the client
    - A proxy for calling client methods can be obtained by calling ‘OperationContext.GetCallbackChannel<TYPE>()’

```
[ServiceContract(CallbackContract=typeof(IPizzaShopCallback))]
interface IPizzaShop {
    [OperationContract]
    void CreateAccount(string clientID);
    [OperationContract]
    void CloseAccount(string clientID);
    [OperationContract(IsOneWay=true)]
    void OrderPizza(string clientID, string pizza);
    [OperationContract(IsOneWay = true)]
    void OrderComplete(string clientID);
    [OperationContract]
    List<string> GetOrder(string clientID);
    [OperationContract]
    double PriceOrder(string clientID);
}
interface IPizzaShopCallback {
    [OperationContract(IsOneWay = true)]
    void OrderReadyForCollection();
}
```

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class PizzaShop : IPizzaShop {
    /* Existing methods as before... */
    public void OrderComplete(string clientID) {
        List<string> order = GetManager().GetOrder(clientID);
        IPizzaShopCallback callback =
                OperationContext.Current.GetCallbackChannel<IPizzaShopCallback>();
        PizzaCooker cooker = new PizzaCooker(callback, order);

        Thread t = new Thread(new ThreadStart(cooker.CookPizzas));
        t.IsBackground = true;
        t.Start();
    }
}
```

```
class PizzaCooker {
    public PizzaCooker(IPizzaShopCallback callback, List<string> order) {
        this.callback = callback;
        this.order = order;
    }
    public void CookPizzas() {
        Thread.Sleep(3000);
        Console.WriteLine("Cooking Done!");
        callback.OrderReadyForCollection();
    }
    private IPizzaShopCallback callback;
    private List<string> order;
}
```

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="PizzaShopV7.PizzaShop" behaviorConfiguration="behaviorOne">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8123/PizzaShopSeven"/>
          </baseAddresses>
        </host>
        <endpoint address="" binding="wsDualHttpBinding" contract="PizzaShopV7.IPizzaShop"/>
        <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="behaviorOne">
          <serviceMetadata httpGetEnabled="True"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

```
class Program {
    static void Main(string[] args) {
        InstanceContext ic = new InstanceContext(new CallbackHandler());
        PizzaShopClient client = new PizzaShopClient(ic);
        client.CreateAccount("AB12");

        client.OrderPizza("AB12", "Capricciosa");
        client.OrderPizza("AB12", "American Hot");
        client.OrderPizza("AB12", "Siciliana");
        client.OrderPizza("AB12", "Diavolo");

        Console.WriteLine("Just ordered the following:");
        foreach (string pizza in client.GetOrder("AB12")) { Console.WriteLine("\t{0}", pizza); }
        Console.WriteLine("At a total cost of: £{0}", client.PriceOrder("AB12"));
        client.OrderComplete("AB12");

        Console.WriteLine("Hit return after pizzas cooked");
        Console.ReadLine();

        client.CloseAccount("AB12");
        client.Close();
    }
}
```

```
class CallbackHandler : IPizzaShopCallback {
    public void OrderReadyForCollection() {
        Console.WriteLine("The pizzas are cooked!!");
    }
}
```

Data Contracts

- A Data Contract specifies how instances of types are marshalled into requests and responses
  - Marshalling to XML is done via a ‘DataContractSerializer’
- Once again this is done using attributes
  - The class is annotated with ‘DataContract’
  - Fields are annotated with ‘DataMember’
  - Classes and fields which are not marked with attributes will not be present in the generated WSDL
- As with service contracts you have fine grained control
  - E.g. ‘Name’, ‘Order’ and ‘IsRequired’ on ‘DataMember’

```
[ServiceContract]
interface IPizzaShop {
    [OperationContract]
    void CreateAccount(string clientID);

    [OperationContract]
    void CloseAccount(string clientID);

    [OperationContract]
    void OrderPizza(string clientID, Pizza pizza);

    [OperationContract]
    Order GetOrder(string clientID);

    [OperationContract]
    double PriceOrder(string clientID);
}
```

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class PizzaShop : IPizzaShop {
    /* Other methods as before...  */

    public void OrderPizza(string clientID, Pizza pizza) {
        Console.WriteLine("Received Order From {0} For: {1}", clientID, pizza);
        GetManager().AddToOrder(clientID, pizza);
    }
    public Order GetOrder(string clientID) {
        List<Pizza> pizzas = GetManager().GetOrder(clientID);
        Order order = new Order();
        order.ClientID = clientID;
        order.Pizzas = pizzas;

        return order;
    }
}
```

```
[DataContract(Namespace="http://valueTypes/NS")]
public class Order {
    [DataMember(Name = "AccountNumber", IsRequired = true)]
    public string ClientID {
        get {
                return clientID;
        }
        set {
                clientID = value;
        }
    }
    [DataMember(Name = "OrderContents", IsRequired = true)]
    public List<Pizza> Pizzas {
        get {
                return pizzas;
        }
        set {
                pizzas = value;
        }
    }
    private string clientID;
    private List<Pizza> pizzas;
}
```

```
public enum PizzaSize { SMALL, MEDIUM, LARGE }

[DataContract(Namespace = "http://valueTypes/NS")]
public class Pizza {
    [DataMember(Name = "Choice", IsRequired = true)]
    public string Name {
        get {
                return name;
        }
        set {
                name = value;
        }
    }
    [DataMember(Name = "Serving", IsRequired = true)]
    public PizzaSize Size {
        get {
                return size;
        }
        set {
                size = value;
        }
    }
    private string name;
    private PizzaSize size;
```

```
<xs:schema elementFormDefault="qualified" targetNamespace="http://valueTypes/NS"
                xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://valueTypes/NS">
    <xs:complexType name="Pizza">
        <xs:sequence>
            <xs:element name="Choice" nillable="true" type="xs:string"/>
            <xs:element name="Serving" type="q1:PizzaSize"
                        xmlns:q1="http://schemas.datacontract.org/2004/07/PizzaShopV8"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Pizza" nillable="true" type="tns:Pizza"/>
    <xs:complexType name="Order">
        <xs:sequence>
            <xs:element name="AccountNumber" nillable="true" type="xs:string"/>
            <xs:element name="OrderContents" nillable="true" type="tns:ArrayOfPizza"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Order" nillable="true" type="tns:Order"/>
    <xs:complexType name="ArrayOfPizza">
        <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded"
                        name="Pizza" nillable="true" type="tns:Pizza"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="ArrayOfPizza" nillable="true" type="tns:ArrayOfPizza"/>
</xs:schema>
```

```
static void Main(string[] args) {
    PizzaShopClient client = new PizzaShopClient();
    client.CreateAccount("AB12");

    client.OrderPizza("AB12", new Pizza() { Choice="Capricciosa", Serving=PizzaSize.SMALL });
    client.OrderPizza("AB12", new Pizza() { Choice="American Hot", Serving=PizzaSize.MEDIUM });
    client.OrderPizza("AB12", new Pizza() { Choice="Siciliana", Serving=PizzaSize.LARGE });
    client.OrderPizza("AB12", new Pizza() { Choice="Diavolo", Serving=PizzaSize.SMALL });
    client.OrderPizza("AB12", new Pizza() { Choice="Quattro Formaggi", Serving=PizzaSize.MEDIUM });
    client.OrderPizza("AB12", new Pizza() { Choice="Quattro Stagioni", Serving=PizzaSize.LARGE });

    Order order = client.GetOrder("AB12");

    Console.WriteLine("Just ordered the following on account {0}:", order.AccountNumber);
    foreach (Pizza pizza in order.OrderContents) {
        Console.WriteLine("\t{0} of size {1}", pizza.Choice, pizza.Serving);
    }

    Console.WriteLine("At a total cost of: £{0}", client.PriceOrder("AB12"));

    client.CloseAccount("AB12");
    client.Close();
}
```

Message Contracts

- A message contract controls how marshalled types are placed into the request and/or response
  - For SOAP messages data can be placed in headers or in one or more parts within the body
- The ‘MessageContract’ attribute is used to control the structure of SOAP messages
  - ‘MessageHeader’ and ‘MessageBodyMember’ determine where marshalled types are placed
  - With headers extra SOAP specific data can be added

Understanding Channels

- A channel is a layer through which messages pass
  - WCF arranges channels into channel stacks
  - A preconfigured channel stack is called a binding
- Channels are divided into transports and protocols
  - Transport channels represent protocols
    - Such as TCP, HTTP and MSMQ
  - Protocol channels implement services
    - Such as security and transactions

Understanding Channels

- A channel must support each type of service contract
  - These being request-response, one-way and duplex
- This is achieved using channel shapes
  - A channel shape is just an interface
- There are 10 possible shapes:
  - IInputChannel and IInputSessionChannel
  - IOutputChannel and IOutputSessionChannel
  - IRequestChannel and IRequestSessionChannel
  - IReplyChannel and IReplySessionChannel
  - IDuplexChannel and IDuplexSessionChannel

Understanding Bindings

- A binding is a preconfigured channel stack
  - There are 10 bindings in .NET V3.0 and 12 in V3.5
- Bindings can be set in code or XML
  - A Binding is a collection of Binding Elements
  - You can iterate over these to view their details
- One binding applies only to apps in the same machine
  - A ‘netNamedPipeBinding’ uses pipes to carry messages between two applications on the local machine
  - Pipes are a form of Inter Process Communication (IPC)
    - Which has been used on both UNIX and Windows for decades

WCF Bindings in .NET 3.5

Binding	Description
basicHttpBinding	WS-I Basic Profile Web Services
wsHttpBinding	As above plus WS-* Standards
wsDualHttpBinding	Web Services with support for Duplex Contracts
webHttpBinding	REST / POX style Web Services
netTcpBinding	Interaction between CLR's
netNamedPipeBinding	Interaction between CLR's on same machine
netMsmqBinding	Asynchronous communication using MSMQ
netPeerTcpBinding	For peer to peer applications
msmqIntegrationBinding	Communication using MSMQ and Queues
wsFederationHttpBinding	WS-* Web Services with Federated Identity
ws2007HttpBinding (3.5)	Updated version of wsHttpBinding
ws2007FederationHttpBinding (3.5)	Updated version of wsFederationHttpBinding

Serializing Types

- WCF serializes data into requests and responses
  - It is converted from an object tree to an XML Infoset
    - This is an abstract description of an XML document
    - Based on a tree of nodes of different types
  - It is then encoded as a stream of bytes
- Five serialization types are supported:
  - XML Schema types (DataContractSerializer)
  - XSD types plus type info (NetDataContractSerializer)
  - XML without a related XSD (XmlSerializer)
  - JSON (DataContractJsonSerializer)
    - JSON uses JavaScript Object Notation

Hosting a WCF Based Application

- There are many options for hosting a WCF app:
  - A conventional .NET application started manually
    - The ‘ServiceHost’ class lets you start up WCF yourself
  - A .NET application managed via WAS
    - The Windows Process Activation Service
  - A Virtual Application running inside IIS 7
- WAS and IIS will be the most common choices
  - Since lifecycle management is provided for you
  - But equally you may wish your application to be self-hosting so that you have complete control
    - Any number of services can live within the same process

Using WPF

Working With Windows Presentation Foundation

garth@ggilmour.com

Introducing WPF

- Pre .NET GUI development was polarized
  - VB 6 was easy but hid the underlying code
  - MFC showed the code but was ‘write only’
- Windows Forms provided the best of both worlds
  - The underlying code was exposed and easy to read
- WPF expands the notion of what a GUI is
  - Apps can be run on the desktop or in a browser
  - Multimedia content can be embedded directly
  - Support for typography, vector graphics and animations
  - 3D drawing via the Microsoft DirectX API

WPF and Silverlight

- Silverlight is a cross-platform browser plug-in
  - Which enables you to download and run WPF apps
- The Silverlight download includes:
  - The Common Language Runtime
  - A minimal set of the framework libraries
  - A XAML parser and Silverlight specific controls
- Silverlight is not synonymous with WPF
  - It only provides features that work on all platforms
  - E.g. there is no 3D support in Silverlight
- Silverlight is one competitor for ‘Web 2.0’
  - The others include Flash and JavaFX

Developing WPF Applications

- WPF supports two development models
  - Using a standard C# object oriented API
  - Using Extensible Application Markup Language (XAML)
    - This is the format used by Visual Studio projects
- Often the WPF code will be written for you
  - But the best education is to write it manually
- The next slides show:
  - A WPF application written in code
  - A WPF application written in XAML
    - The XML is dynamically loaded rather than precompiled

```
class Program {
    private static Label label;
    private static Button button;
    private static TextBox textBox;

    static void Clicked(object sender, RoutedEventArgs args) {
        textBox.Text = "Modified content";
    }

    [STAThread]
    static void Main(string[] args) {
        //An application hosts one or more windows
        Application application = new Application();

        //We need at least one window
        Window mainWindow = new Window();
        mainWindow.SizeToContent = SizeToContent.WidthAndHeight;
        mainWindow.Title = "Simple WPF Application";

        //Register an event handler for button clicks
        mainWindow.AddHandler(Button.ClickEvent, new RoutedEventHandler(Clicked));
```

```
        //Controls will be grouped in a vertical panel holding:
        // 1) A label
        // 2) A horizontal panel holding:
        //      2a) A button
        //      2b) A text box
        StackPanel panel1 = new StackPanel();
        panel1.Orientation = Orientation.Vertical;
        label = new Label();
        label.Content = "A Simple Demo of WPF";
        panel1.Children.Add(label);

        StackPanel panel2 = new StackPanel();
        panel2.Orientation = Orientation.Horizontal;

        button = new Button();
        button.Content = "Push Me";
        textBox = new TextBox();
        textBox.Text = "Initial content";

        panel2.Children.Add(button);
        panel2.Children.Add(textBox);
        panel1.Children.Add(panel2);
        mainWindow.Content = panel1;
        application.Run(mainWindow);
    }
}
```

```
class Program {
    private static TextBox textBox;
    [STAThread]
    static void Main(string[] args) {
        Application application = new Application();
        Uri uri = new Uri("pack://application:,,,/MyXaml.xml");
        Stream stream = Application.GetResourceStream(uri).Stream;
        FrameworkElement element = XamlReader.Load(stream) as FrameworkElement;

        Window mainWindow = new Window();
        mainWindow.SizeToContent = SizeToContent.WidthAndHeight;
        mainWindow.Title = "Simple WPF Application";

        mainWindow.AddHandler(Button.ClickEvent, new RoutedEventHandler(Clicked));

        mainWindow.Content = element;
        textBox = element.FindName("textBox") as TextBox;

        application.Run(mainWindow);
    }
    static void Clicked(object sender, RoutedEventArgs args) {
        textBox.Text = "Modified content";
    }
}
```

```
<StackPanel Orientation="Vertical"
                xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Label>
        A Simple Demo of WPF
    </Label>
    <StackPanel Orientation="Horizontal">
        <Button Name="button">
            Push Me
        </Button>
        <TextBox Name="textBox">
            Initial content
        </TextBox>
    </StackPanel>
</StackPanel>
```

WCF Basics

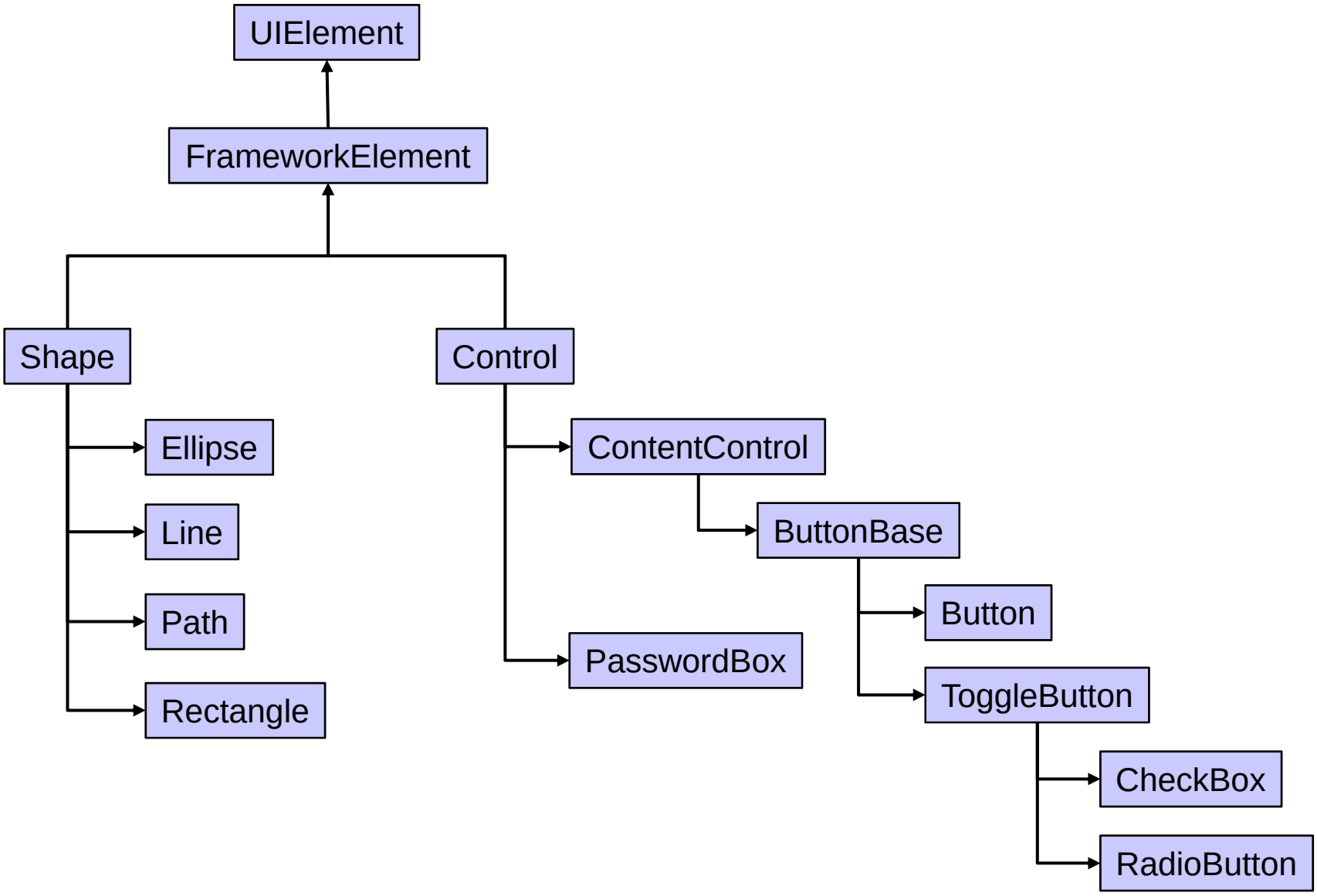
- The main object in WCF is of type ‘Application’
  - It manages one or more ‘Window’ objects
  - There must be a single main window
- The size of the window is in ‘logical pixels’
  - As are all dimensions used in WPF
  - These are also known as ‘device independent pixels’ and measure 1/96 of an inch on any display
- Each window has a ‘Content’ property
  - This can only be set to a single item
  - It can be a string, bitmap, drawing or control

Controls in WPF

- Controls in WPF inherit from ‘FrameworkElement’
  - Which in turn inherits from ‘UIElement’
  - This declares the crucial ‘OnRender’ method
- Its subclasses act as bases for types of control
  - The ‘Shape’ class is the base for controls that perform drawing using 2D vector graphics (Ellipse, Line etc...)
  - The ‘TextElement’ class is the base for controls that contain inline or block based text (TextBlock etc...)
  - The ‘Control’ class is the base for controls that receive and respond to input from the user (aka ‘user interactive’)
    - Unlike other GUI libraries WPF reserves the term ‘control’ for interactive components, as opposed to all visual components

Controls in WPF

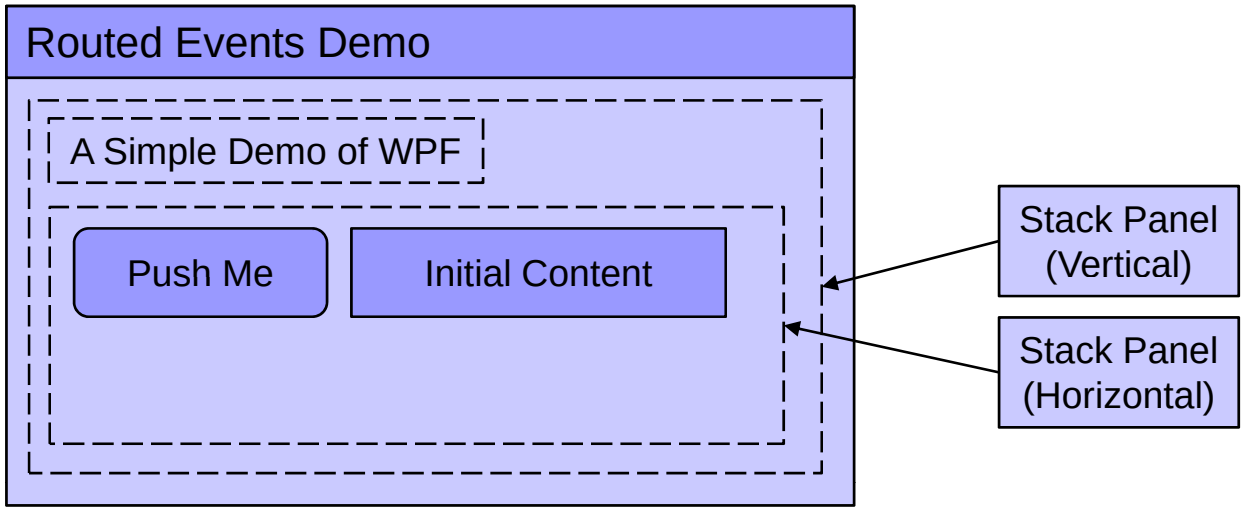
- The ‘ContentControl’ class inherits from ‘Control’
  - It defines the ‘Content’ property used in a window
  - Most WPF controls inherit from ‘ContentControl’
    - This allows controls to contain other controls as required
- Multiple controls are grouped using panels
  - The ‘Panel’ class inherits from ‘FrameworkElement’
  - Subclasses include ‘Canvas’, ‘Grid’, and ‘StackPanel’
- Panels position their children dynamically
  - They use layout strategies rather then absolute positioning
    - E.g. ‘StackPanel’ adds items to a horizontal or vertical stack
  - This can be adjusted via alignments, padding and margins



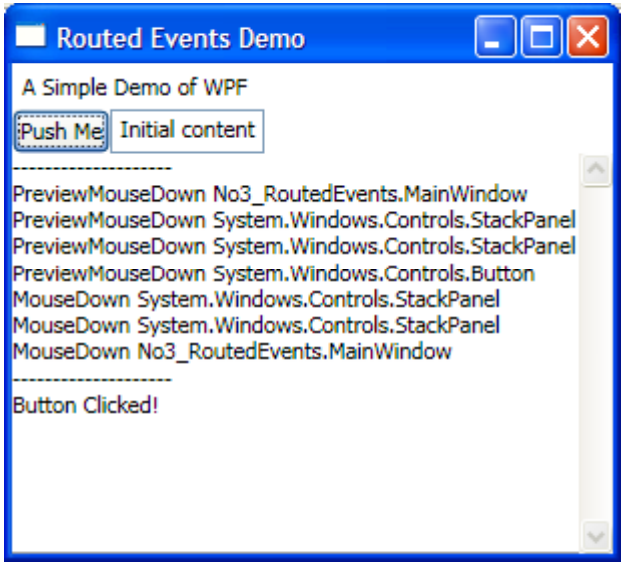
Event Handling in WPF

- In most GUI libraries an event is sent directly to the in-focus control that the mouse is over
  - This is not as simple in WPF because controls can contain other controls, as well as images and text blocks
- WPF introduces two special kinds of events
  - A ‘tunnelling event’ starts at the top of the control tree and works its way down to the control with focus
  - A ‘bubbling event’ starts at the control with focus and works it way up to the top of the control tree
- Any control can register a handler for any event type
  - E.g. ‘win.AddHandler(Control.MouseDownEvent, handler)’

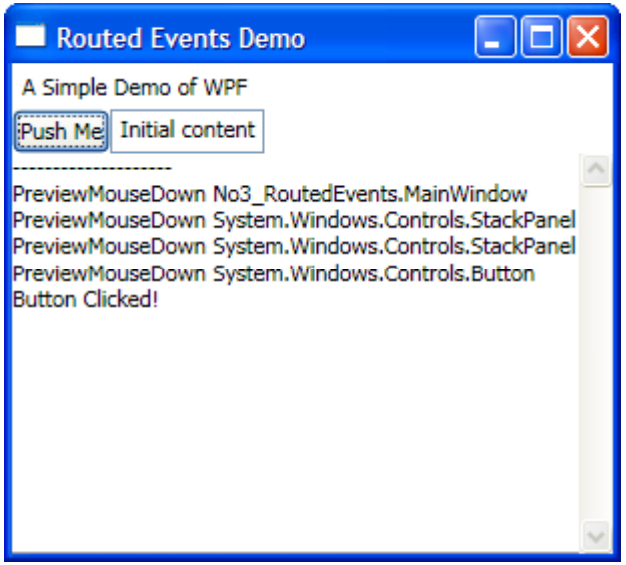
```
private void AddEventHandlers() {
    this.AddHandler(Control.PreviewMouseDownEvent, new RoutedEventHandler(MyHandler));
    panel1.AddHandler(Control.PreviewMouseDownEvent, new RoutedEventHandler(MyHandler));
    panel2.AddHandler(Control.PreviewMouseDownEvent, new RoutedEventHandler(MyHandler));
    button.PreviewMouseDown += MyHandler;

    button.Click += (object sender, RoutedEventArgs args) => addMessageToOutput("Button Clicked!");

    this.AddHandler(Control.MouseDownEvent, new RoutedEventHandler(MyHandler), true);
    panel1.AddHandler(Control.MouseDownEvent, new RoutedEventHandler(MyHandler), true);
    panel2.AddHandler(Control.MouseDownEvent, new RoutedEventHandler(MyHandler), true);
    button.MouseDown += MyHandler;
}
private void MyHandler(object sender, RoutedEventArgs args) {
    if (sender == this && args.RoutedEvent.Name.Equals("PreviewMouseDown") {
        addMessageToOutput("--------------------");
    }
    string msg = args.RoutedEvent.Name + " " + sender.GetType().ToString();
    addMessageToOutput(msg);

    if (sender == this && args.RoutedEvent.Name.Equals("MouseDown")) {
        addMessageToOutput("--------------------");
    }
    viewer.ScrollToBottom();
}
```



With flag set



Without flag set



Event Handling in WPF

- Many events come in bubbling / tunnelling pairs
  - For each event there is a corresponding preview event
  - E.g. ‘MouseDown’ and ‘PreviewMouseDown’
- The preview event fires first by tunnelling down
  - It travels from the root down to the control with focus
  - E.g. If a button is in a panel in a window, then the preview event will fire on the window, panel and button controls
- Then the expected event fires by bubbling up
  - E.g. the ‘MouseDown’ event fires on the button first, then on the panel and finally the window

Event Handling in WPF

- The bubbling of the event can be cancelled
  - A control can signal that event handling is complete
  - By setting the ‘Handled’ property of the event args object
- The bubbling is not actually cancelled
  - It still occurs but your event handlers are not triggered
  - Unless the handler was added with a special flag set
- It is possible for a control to capture the mouse
  - This means it continues to receive ‘MouseMove’ and ‘MouseUp’ events even if the mouse leaves the control
  - Windows can override this when required

Examining XAML

- XAML is a declarative way of specifying a control
  - The control need not necessarily be a window
- The declarative form is easier to read than code
  - It displays the hierarchical relationships better
  - It is frequently shorter and less verbose
  - Of course code is required for event handlers etc...
- XAML can be processed in two ways
  - It can be read in and interpreted at runtime
    - This is useful for rapid prototyping
  - It can be precompiled and merged with other IL
    - Obviously this will be the most common choice

XAML and Markup Extensions

- XAML is an XML based language
  - But it defines an alternative notation for concisely providing arbitrary information inside attributes
- This feature is known as Markup Extensions
  - Extensions specify resources, state data bindings etc...
- A markup extension is placed in braces
  - E.g. ‘<MyControl Text=“{MyExtension STUFF}”>’
  - The STUFF part may be a sequence of name/value pairs
- If you need to place braces inside attributes as part of static text then they need to be escaped
  - E.g. ‘<MyControl Text=“{ }{ STUFF }”>’

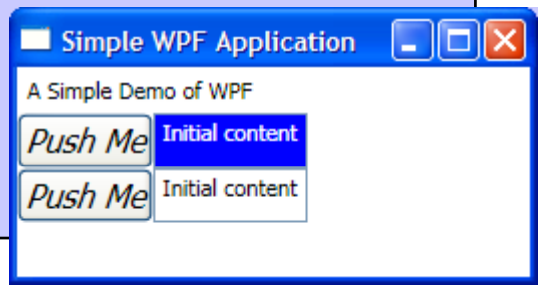
Specifying Resources

- Many elements are able to have a resources section
  - E.g. inside a ‘StackPanel’ element you can have a ‘StackPanel.Resources’ element as the first child
  - Resources are available to all child controls
    - This is a convenient way of specifying ‘magic numbers’
- There are three ways of accessing static resources
  - Via a markup extension called ‘StaticResource’
    - ‘Size=“{StaticResource mySize}”’
  - Via a conventional XML element:
    - ‘<StaticResource ResourceKey=“mySize”/>’
  - In code via the ‘Resources’ property of the control
    - ‘myStack.Resources.Add(“Key”, value)’

Resources and Styles

- A common use of resources is to hold ‘Style’ elements
  - A style is a set of properties to be applied to many controls
    - This helps add a consistent ‘look and feel’
  - Properties set directly on a control override style settings
- There are several ways of applying a style
  - The style may have a key, which is referenced by controls
  - The style may use the ‘TargetType’ attribute
    - Which specifies that the style applies to all controls of a type
    - E.g. ‘TargetType={x:Type Label}’
  - The mechanisms can be combined
    - In which case a style is only used if the control is of the correct type and also references the correct key

```
<StackPanel Orientation="Vertical"
            xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel.Resources>
    <Style TargetType="Button">
      <Setter Property="FontSize" Value="16"/>
      <Setter Property="FontStyle" Value="Italic"/>
    </Style>
    <Style x:Key="myStyle">
      <Setter Property="Control.Foreground" Value="White"/>
      <Setter Property="Control.Background" Value="Blue"/>
    </Style>
  </StackPanel.Resources>
  <Label>A Simple Demo of WPF</Label>

  <StackPanel Orientation="Horizontal">
    <Button Name="button1">Push Me</Button>
    <TextBox Name="textBox1" Style="{StaticResource myStyle}">Initial content</TextBox>
  </StackPanel>

  <StackPanel Orientation="Horizontal">
    <Button Name="button2">Push Me</Button>
    <TextBox Name="textBox2">Initial content</TextBox>
  </StackPanel>
</StackPanel>
```

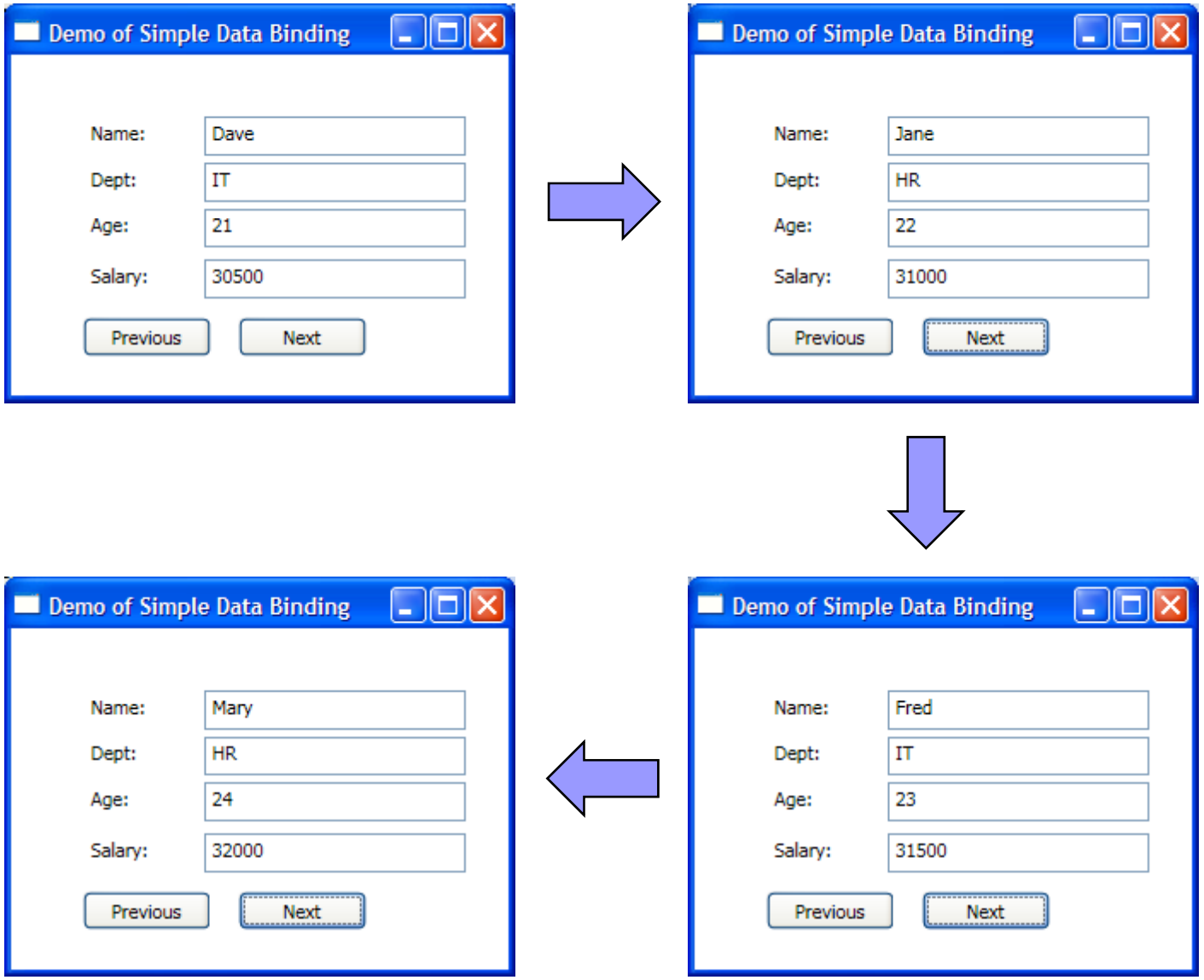


Styles and Templates

- Every control in WPF has a ‘Template’ property
  - This references a ‘ControlTemplate’ object that completely defines the appearance of the control
  - There may be further properties for controls with children
    - Such as ‘DataTemplate’ and ‘ItemTemplate’
- Templates and styles are not the same thing
  - Styles are layered on top of the current template
  - A control usually has no default styling
- You can define your own templates in the XAML file
  - As with styles they can be based on keys or types

XAML and Data Binding

- XAML provides a markup extension for data binding
  - There is also a more verbose but pure XML alternative
- Data binding is specified on arbitrary properties
  - E.g. ‘Content=“{Binding ElementName=abc Path=def}”’
  - Note that the values are never quoted
- The binding notation can have a ‘Mode=X’ pair
  - ‘OneWay’ means the data source is NOT updated
  - ‘TwoWay’ means that the source is updated
  - ‘OneTime’ means binding occurs only on startup
  - ‘OneWayToSource’ signifies write only access



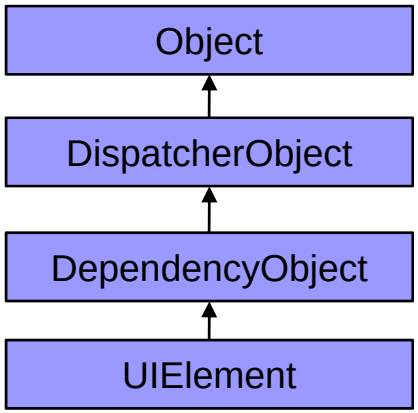
```
<!-- Layout Properties of Controls Omitted -->
<Grid Width="272" Height="196">
    <Label Name="label1">Name:</Label>
    <Label Name="label2">Dept:</Label>
    <Label Name="label3">Age:</Label>
    <Label Name="label4">Salary:</Label>

    <TextBox Name="textBox1" Text="{Binding Path=Name}" />
    <TextBox Name="textBox2" Text="{Binding Path=Dept}"/>
    <TextBox Name="textBox3" Text="{Binding Path=Age}"/>
    <TextBox Name="textBox4" Text="{Binding Path=Salary}"/>

    <Button Name="button1" Click="button1_Click">Previous</Button>
    <Button Name="button2" Click="button2_Click">Next</Button>
</Grid>
```

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }
    private void Window_Loaded(object sender, RoutedEventArgs e) {
        sampleData = BuildData();
        DataContext = sampleData[count];
    }
    private static List<Employee> BuildData() {
        var data = new List<Employee>();
        data.Add(new Employee("Dave", "IT", 21, 30500.0));
        // Many other employees added to list...
        return data;
    }
    private void button1_Click(object sender, RoutedEventArgs e) {
        if (count > 0) {
            count--;
        }
        DataContext = sampleData[count];
        updateBindings();
    }
```

```
    private void button2_Click(object sender, RoutedEventArgs e) {
        if (count < (sampleData.Count - 1)) {
                    count++;
        }
        DataContext = sampleData[count];
        updateBindings();
    }
    private void updateBindings() {
        textBox1.GetBindingExpression(TextBox.TextProperty).UpdateTarget();
        textBox2.GetBindingExpression(TextBox.TextProperty).UpdateTarget();
        textBox3.GetBindingExpression(TextBox.TextProperty).UpdateTarget();
        textBox4.GetBindingExpression(TextBox.TextProperty).UpdateTarget();
    }
    private List<Employee> sampleData;
    private int count;
}
```

The WPF Class Hierarchy

- So far we have assumed that the ‘UIElement’ class is at the top of the WPF class hierarchy
  - In fact ‘UIElement’ inherits from ‘DependencyObject’
  - Which in turn inherits from ‘DispatcherObject’
- Each of these supports a single feature:
  - Dependency Object supports dependency properties
  - Dispatcher Object supports multithreading



Dependency Properties

- A Dependency Property is a property where value changes cascade through the controls children
  - E.g. Change the ‘FontSize’ of a window and any descendents with the same property change as well
    - Unless their property value had been explicitly set
- In order to create a dependent property:
  - Call ‘DependencyProperty.Register(...)’
    - This registers the property with the WPF internals
    - The object returned should be stored in a static field
  - Declare the property as usual
    - But use the ‘SetValue’ and ‘GetValue’ methods inherited from ‘DependencyObject’ to store and load the value

WPF and Multithreading

- By default a WPF application has two threads
  - A rendering thread that runs in the background
  - An event-dispatching thread called the ‘UI Thread’
    - This is managed by an instance of the ‘Dispatcher’ class
- The ‘DispatcherObject’ class has two functions
  - To provide access to the ‘Dispatcher’ instance
  - To provide methods which allow a control to check that it is only ever being used within the UI Thread
    - These methods are ‘CheckAccess’ and ‘VerifyAccess’
- Exceptions occur if you use a control in another thread
  - You can schedule jobs of work onto the UI Thread