

WPF

Windows Presentation Foundation

© JMA 2012

INTRODUCCIÓN

© JMA 2012

¿Qué es WPF?

- WPF es una tecnología para desarrollar la siguiente generación de aplicaciones en Windows y en la web, utilizando toda la potencia del hardware y creando las mejores experiencias de usuario (UX).

© JMA 2012

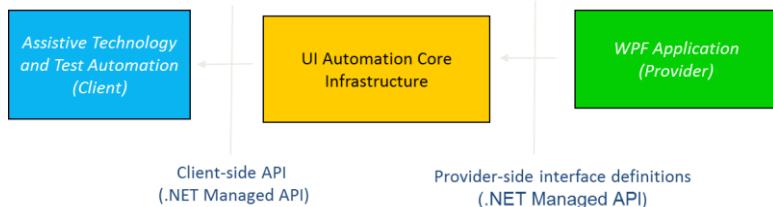
Windows Presentation Foundation

- **Tecnología IU estratégica de Microsoft**
 - Plataforma y motor
 - Lo mejor del Web y Windows
 - Uso de GPU para alto rendimiento
- **Unificación**
 - Formularios
 - 2D / 3D
 - Video / imágenes
 - Tipografía / Documentos
 - Animaciones
 - Speech
- **Silverlight**
 - Subconjunto multiplataforma
 - Multinavegador



© JMA 2012

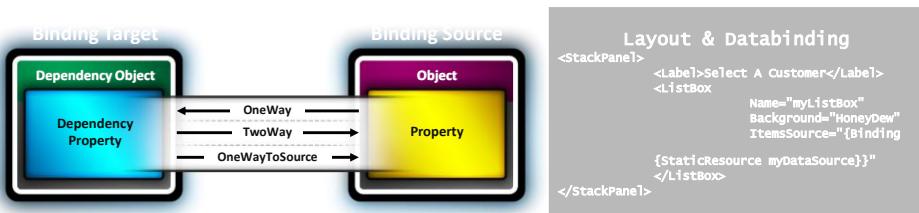
Accesibilidad



- Mejores prácticas de accesibilidad
 - Habilita la accesibilidad mediante programación, que es apoyado por los controles comunes de WPF a través de la interfaz de usuario de API de automatización
 - Apoyo de acceso mediante teclado, tales como teclas de acceso y el orden lógico de tabulación
 - Soporte de métricas del sistema para la configuración del usuario, tales como un alto contraste y visualización de DPI.
 - Crea un interfaz multi-modal, no depender de efectos visuales para transmitir información por sí sola

© JMA 2012

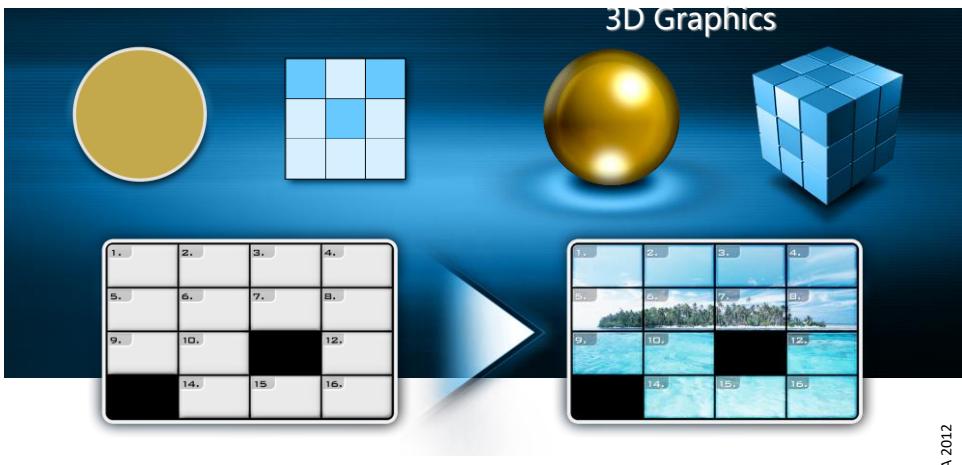
Data Binding



- La interfaz de usuario se puede enlazar a objetos CLR y XML
- Las propiedades de dependencia también se pueden enlazar a ADO.NET y objetos asociados con los servicios web y las propiedades Web
- La ordenación, filtrado y agrupamiento de vista se pueden generar en la parte superior de los datos
- Plantillas de datos se pueden aplicar a los datos

© JMA 2012

2D Graphics, 3D Graphics, Imaging



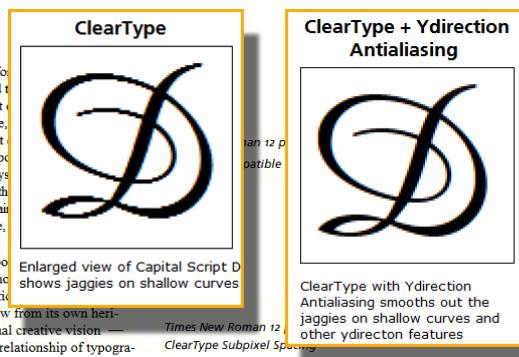
© JMA 2012

ClearType & Antialiasing

Sub-pixel positioning & natural widths

I want to conclude this article by suggesting why support for typography belongs among the important things, and not relegated to typography is not to prettify text, but to articulate it. That it's an aesthetic way — utilizing all the art it can draw from its own heritage, script traditions, and individual creative vision — should not disguise the expressive and organizational relationship of typography to text. A typographic culture, such as the one in which you engage as you read this article, is a system of visual indicators that helps readers navigate text and helps writers express their ideas. In the 550 years since Gutenberg developed metal type casting at Mainz, the printed Latin script has developed a particularly rich typographic culture, using romans, italics, bold

I want to conclude this article by suggesting why support for typography belongs among the important things, and not relegated to typography is not to prettify text, but to articulate it. That it's an aesthetic way — utilizing all the art it can draw from its own heritage, the heritage of manuscript tradition, and individual creative vision — should not disguise the expressive and organizational relationship of typography to text. A typographic culture, such as the one in which you engage as you read this article, is a system of visual indicators that helps readers navigate text and helps writers express their ideas. In the 550 years since Gutenberg developed metal type casting at Mainz, the printed Latin script has developed a particularly rich typographic culture, using romans, italics, bold



© JMA 2012

Nuevas Fonts para WPF

Calibri
Candara
Cambria
Constantia
Corbel
Consolas

Consolas

```
<TextPanel ID="root"
    xmlns="http://schemas.microsoft.com/2003/xaml
    xmlns:def="Definition"
    FontFamily="Calibri">
```

© JMA 2012

Audio & Video

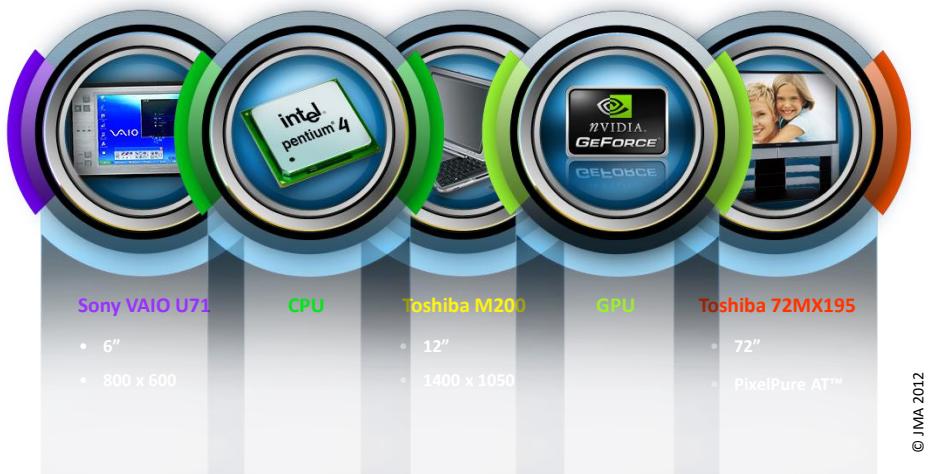


```
<Border Width="400"
    BorderBrush="Green"
    BorderThickness="9">
<StackPanel>
    <MediaElement Source="aero.wmv" />
    <Button>Hello</Button>
</StackPanel>
</Border>
```

- Formatos: WMV, MPEG y algunos AVIs
- Puede ser sincronizado con animaciones
- Windows Media Foundation utiliza para crear instancias de máquinas de reproducción gráficos DirectShow

© JMA 2012

CPU y aceleración por Hardware



Nuevas tecnologías de Documentos

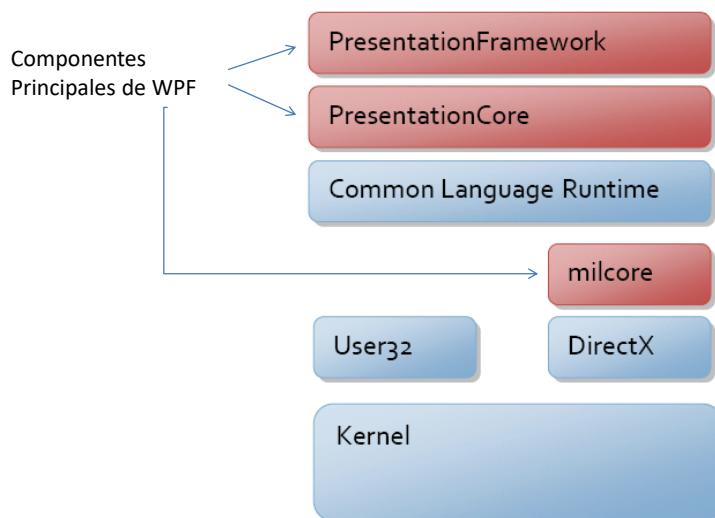


WPF Avanzado

- Elementos componibles
- Motor de diseño flexible
- Potente arquitectura de enlace de datos
- Capacidades de impresión, fuentes y documentos
- Controles lookless
- Estilos y plantillas
- La plena integración de todas las disciplinas de la interfaz de usuario

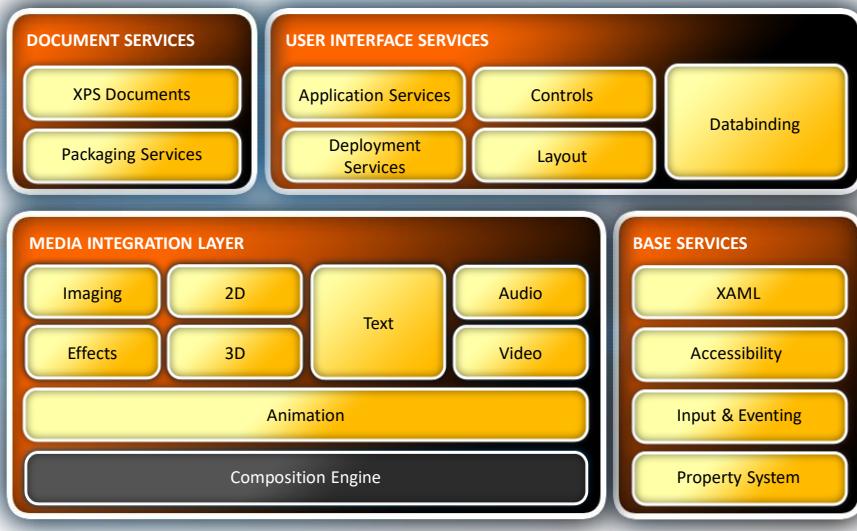
© JMA 2012

Arquitectura WPF



© JMA 2012

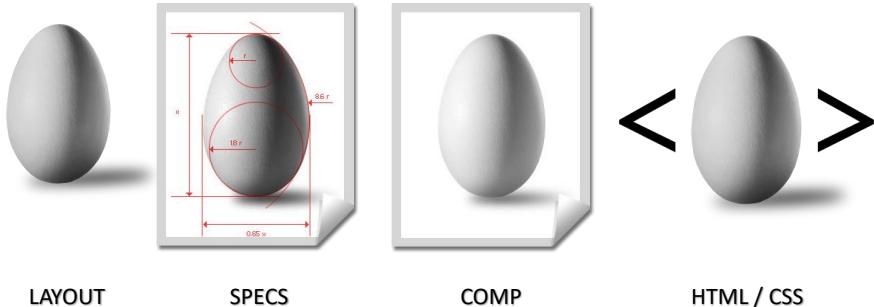
Arquitectura WPF



Roles

- Developers
 - Programador
- Designer
 - Diseñador gráfico
- UX Engineer
 - Especialista en usabilidad

Workflow Designer x Developers



© JMA 2012

Workflow Designer x Developers

PRODUCTION



© JMA 2012

Workflow Designer x Developers

FINAL RESULT



© JMA 2012

Colaboración diseñador-programador



¿Qué es XAML?

XAML = Extensive Application Markup Language

- Lenguaje declarativo
- Código y diseño separados
- Fácilmente editable desde herramientas



XAML

- XAML: Extensible Application Markup Language
- Está basado en XML
- Código mas compacto
- Jerárquico
- Soporte para listas
- Así es como se define la UI de las aplicaciones
 - WPF, Silverlight, Workflow Foundation
- Es un mapeo entre XML y los tipos del .NET Framework
 - Nodos -> Tipos
 - Atributos -> CLR Property o Dependency Property

Sus ventajas

- Se reducen los costos de programación y mantenimiento
- La programación es más eficaz
- Se pueden usar varias herramientas de diseño para implementar y compartir el marcado XAML
- La globalización y localización de las aplicaciones WPF se ha simplificado en gran medida.

© JMA 2012

Ejemplo

```
<Canvas  
    xmlns="http://schemas.microsoft.com  
        /client/2007">  
    <TextBlock FontSize="32"  
        Text="Hello world" />  
</Canvas>
```

Hello world

© JMA 2012

Markup ⇔ CLR

```
<TextBlock FontSize="32"  
          Text="Hello world" />
```

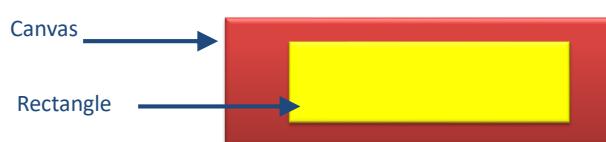
=

```
TextBlock t = new TextBlock();  
t.FontSize = 32;  
t.Text = "Hello world";
```

© JMA 2012

Composición

```
<Canvas Width="250" Height="200">  
  <Rectangle  
    Canvas.Top="25" Canvas.Left="25"  
    Width="200" Height="150"  
    Fill="Yellow" />  
</Canvas>
```

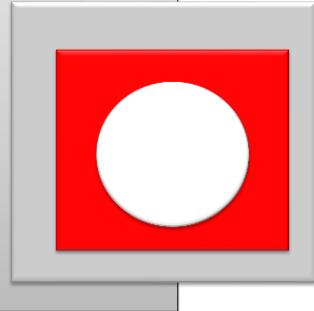


© JMA 2012

Composición Relativa

```
<Canvas Background="Light Gray">
    <Canvas Canvas.Top="25" Canvas.Left="25"
        Width="150" Height="100"
        Background="Red">

        <Ellipse Canvas.Top="25"
            Canvas.Left="25"
            Width="150"
            Height="75"
            Fill="White" />
    </Canvas>
</Canvas>
```



© JMA 2012

Transformaciones

- Todos los elementos las soportan
- Tipos
 - <RotateTransform />
 - <ScaleTransform />
 - <SkewTransform />
 - <TranslateTransform />
 - Moves
 - <MatrixTransform />
 - Scale, Skew and Translate Combined

© JMA 2012

Transformaciones

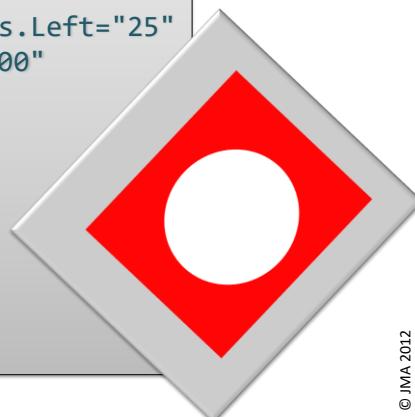
```
<TextBlock Text="Hello World">
    <TextBlock.RenderTransform>
        <RotateTransform Angle="-45" />
    </TextBlock.RenderTransform>
</TextBlock>
```



© JMA 2012

Composición y Transformación

```
<Canvas Background="Light Gray">
    <Canvas.RenderTransform>
        <RotateTransform Angle="-45" />
    </Canvas.RenderTransform>
    <Canvas Canvas.Top="25" Canvas.Left="25"
        Width="150" Height="100"
        Background="Red">
        <Ellipse Canvas.Top="25"
            Canvas.Left="25"
            Width="150"
            Height="75"
            Fill="White" />
    </Canvas>
</Canvas>
```



© JMA 2012

XAML

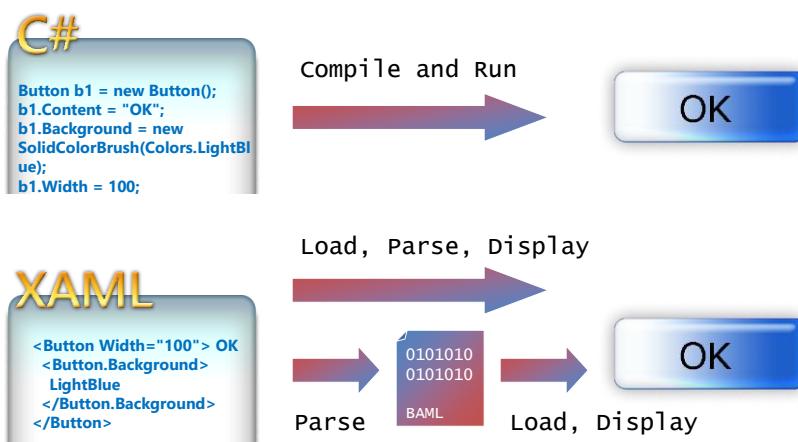
```

10   <!-- XAML basics -->
11
12   <!-- TextBox element maps to System.Windows.Controls.TextBox class
13   Text (Property Attribute) maps to Text property -->
14   <TextBox Text='WPF Demystified' />
15
16   <!-- MouseEnter (Event Attribute) maps to the MouseEnter event.
17       requires a C# or VB code file with MouseEnter procedure. -->
18   <TextBox MouseEnter='TextBox_MouseEnter' />
19
20   <!-- Property Elements are another way to specify property value -->
21   <TextBox>
22     <TextBox.Text>Another example</TextBox.Text>
23   </TextBox>
24
25   <!-- Databinding, Styles, Templates are some of the features
26       enabled with Markup Extensions -->
27
28   <TextBox Text='{Binding AuthorName}' />
29

```

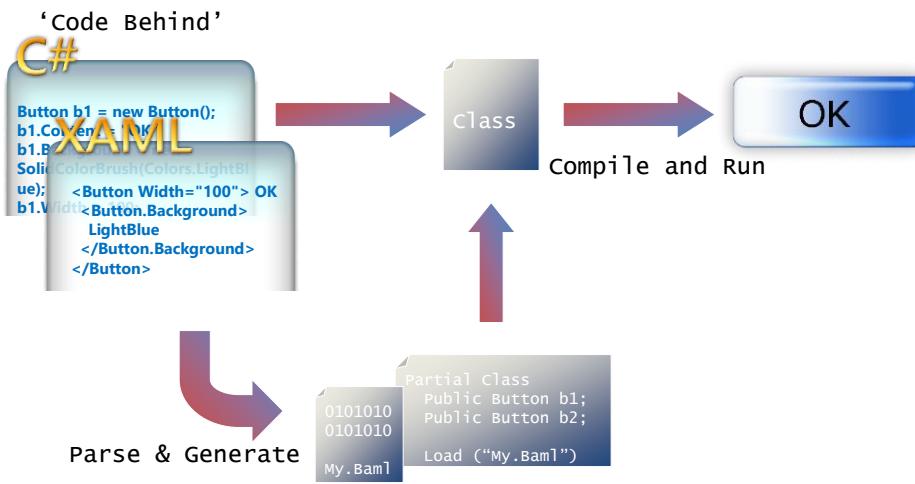
© JMA 2012

¿XAML o Código?



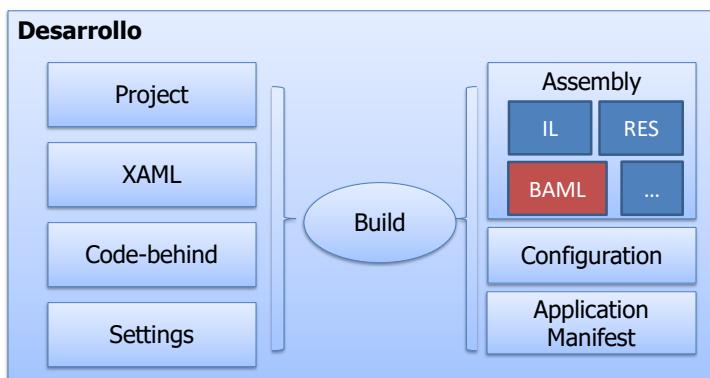
© JMA 2012

¿XAML o Código?



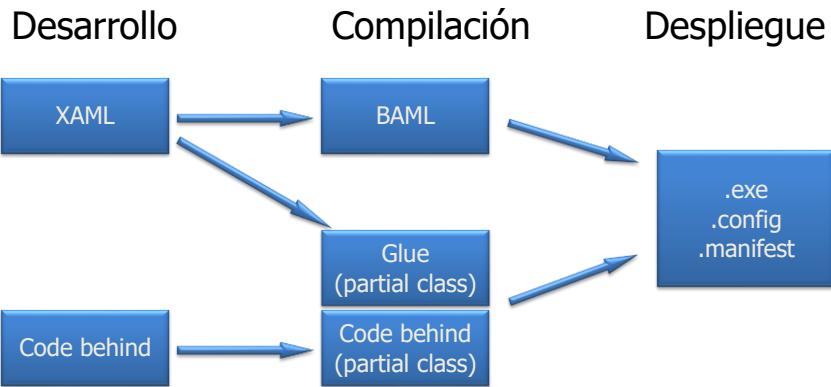
© JMA 2012

Anatomía de una aplicación WPF



© JMA 2012

Ciclo de vida



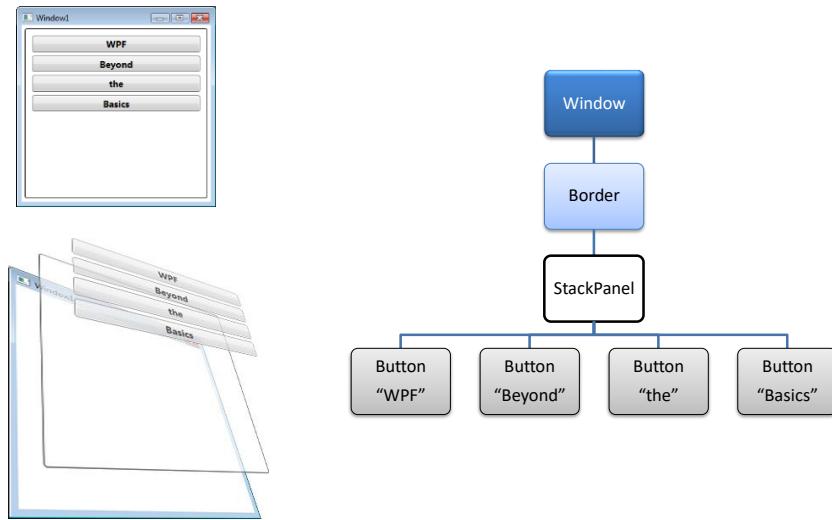
© JMA 2012

Ejecución

- Arboles lógicos, visuales y composición
- Rendering basado en capacidades
 - ‘Hardware’ o ‘software’
 - Puede ser detectado usando RenderCapability
- Controla todo el espacio aéreo visual

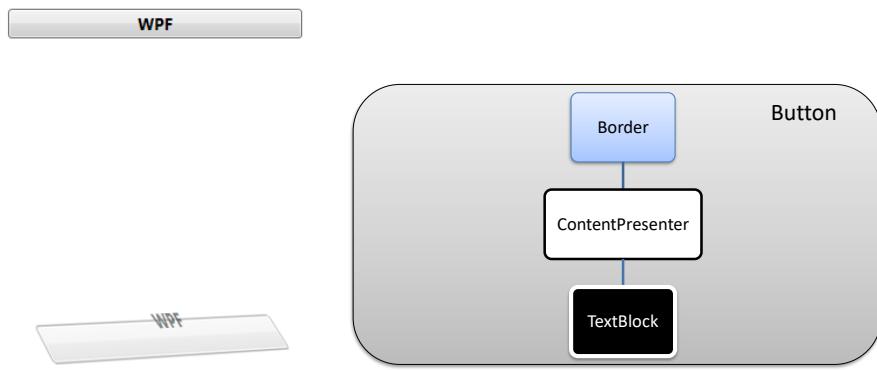
© JMA 2012

Árbol Lógico



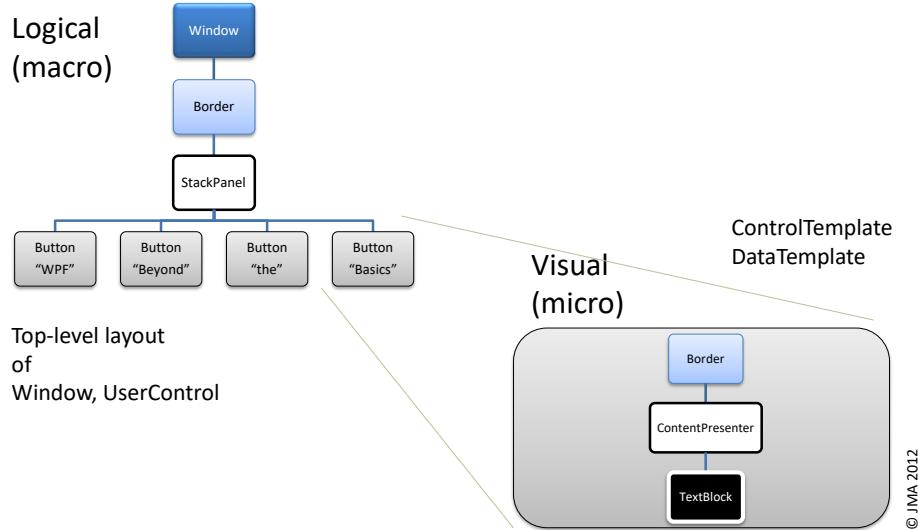
© JMA 2012

Árbol Visual

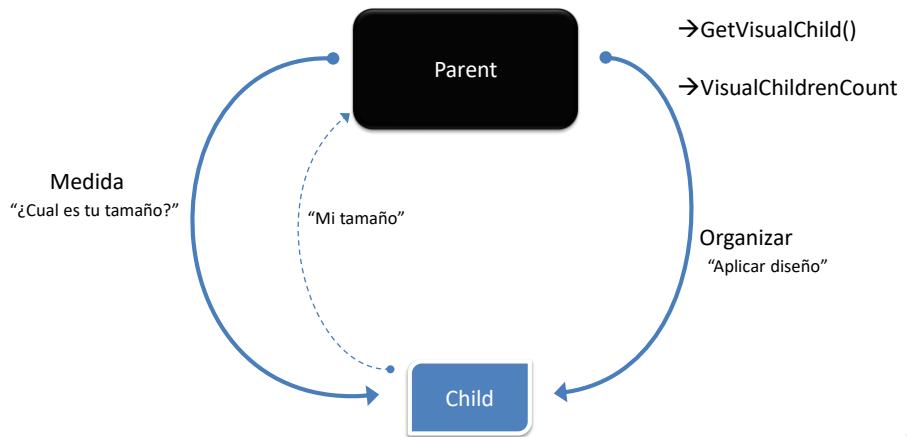


© JMA 2012

Logical + Visual



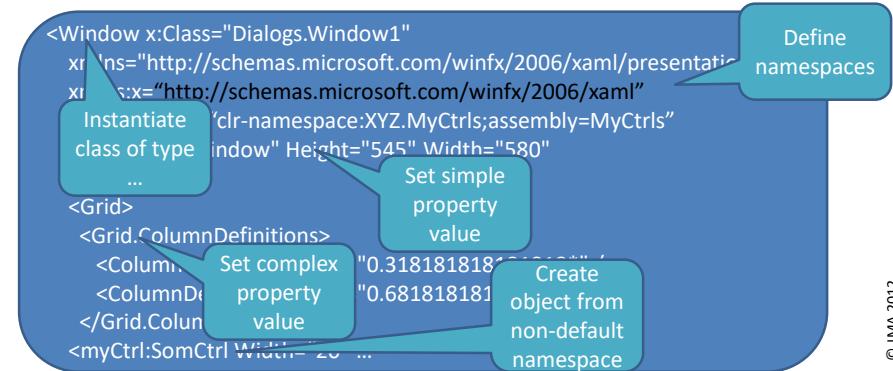
Layout



© JMA 2012

XAML Dinámico

- Se puede leer y guardar XAML desde código
 - XamlReader.Load, XamlReader.Parse
 - XamlWriter.Save



Sintaxis XAML

- Documentos XML bien formados y validos
- Vocabularios propios
- Vocabulario ampliable a través de los espacios de nombres
- Soporte de extensiones de marcado
- Conversión implícita de cadena al tipo de destino:
 - Valor de propiedad
 - Nombre del controlador de eventos

Espacios de nombres

- Básicos:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
mc:Ignorable="d"
```

- Para utilizar otros tipos:

```
xmlns:Prefix="clr-namespace:Namespace;assembly=AssemblyName"
```

© JMA 2012

Propiedades y eventos

- Son los atributos de la etiqueta
- Propiedades Simples
`<nombreDeTipo ... propiedad="valor" ... />`
- Propiedades Complejas/Colecciones
`<nombreDeTipo ... >`
 `<nombreDeTipo.propiedad>`
 `...`
 `<nombreDeTipo.propiedad>`
 `...`
`</nombreDeTipo>`
- Propiedades Asociadas
`<nombreDeTipo ... padre.propiedad="valor" ... />`
`<nombreDeTipo ... hijos.propiedad="valor" ... />`
- Eventos
`<nombreDeTipo ... evento="NombreControlador" ... />`

© JMA 2012

Extensiones de enlazado

- Las llaves {} y } indican el uso de una extensión de marcado que se aparta del tratamiento general de valores de atributo.
 - **Binding**: proporciona un valor enlazado a datos para una propiedad, utilizando el contexto de datos que se aplica al objeto primario en tiempo de ejecución.
 - **TemplateBinding**: permite que una plantilla de control utilice valores para propiedades con plantilla procedentes de propiedades definidas por el modelo de objetos de la clase que utilizará la plantilla.
 - **RelativeSource**: proporciona información de origen para un objeto Binding que puede navegar por varias posibles relaciones en el árbol de objetos en tiempo de ejecución.
 - **StaticResource**: proporciona un valor para una propiedad sustituyendo el valor de un recurso ya definido.
 - **DynamicResource**: proporciona un valor para una propiedad aplazando ese valor para que sea una referencia a un recurso en tiempo de ejecución.
 - **ColorConvertedBitmap**: admite un escenario de creación de imágenes relativamente avanzado.
 - **ComponentResourceKey** y **ThemeDictionary**: admiten aspectos de la búsqueda de recursos y temas que se empaquetan con controles personalizados.

© JMA 2012

Extensiones de marcado

- **x:name**
 - Nombre de referencia disponible para todas las etiquetas
 - Utilizado como nombre de las instancias CLR generadas
 - Búsquedas en el árbol: FindName("control")
- **x:key**
 - Identifica de forma exclusiva los elementos que se crean y a los que se hace referencia en un diccionario de recursos.
- **x:class**
 - Enlace al código subyacente
- **x:type**
 - Construye una referencia Type basada en un nombre de tipo.
- **{x:Static prefix:typeName.staticMemberName}**
 - Referencia a cualquier entidad de código estática por valor definida conforme a Common Language Specification (CLS).
- **{x:null}**
 - Especifica null como valor para una propiedad.

© JMA 2012

Extensiones de Design-Time

- d:DesignHeight and d:DesignWidth
 - d:DesignHeight="300" d:DesignWidth="400"
- d:DataContext
 - d:DataContext="{d:DesignInstance Type=local:Customer}">
- d:DesignInstance and d:IsDesignTimeCreatable
 - <Grid d:DataContext="{d:DesignInstance local:Customer, IsDesignTimeCreatable=True}">
- d:DesignData
 - d:DataContext="{d:DesignData Source=./DesignData/SampleCustomer.xaml}">
- d:DesignSource d:CreateList
 - <CollectionViewSource x:Key="CustomerViewSource"
 - d:DesignSource="{d:DesignInstance local:Customer, CreateList=True}" />
- d>Type

© JMA 2012

XML y XAML

- Caracteres especiales
 - Less than (<) <
 - Greater than (>) >
 - Ampersand (&) &
 - Quotation mark ("") "
- No preserva el espacio en blanco (sp, br, tab, ...)
- Para preservar el espacio en blanco:

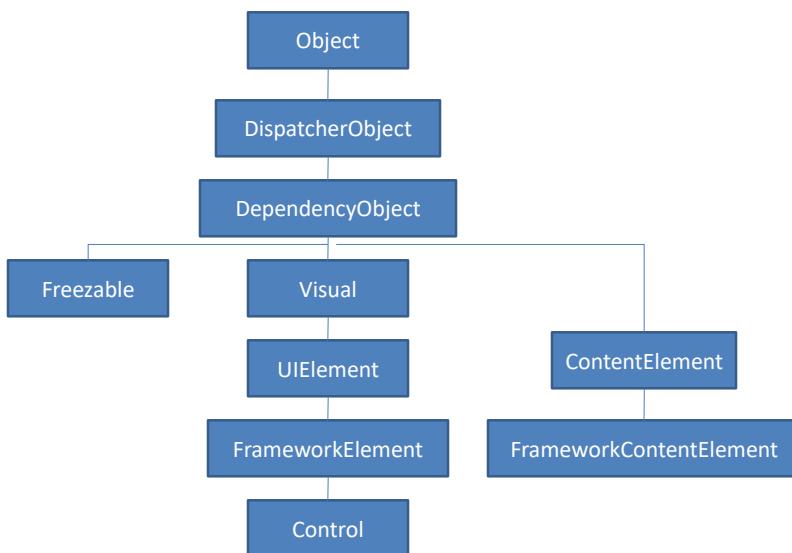

```
<nombredetipo xml:space="preserve" ... >
...
</nombredetipo>
```

© JMA 2012

DISEÑO DE APLICACIONES Y FORMULARIOS

© JMA 2012

Core classes



© JMA 2012

Modelo de contenido

- Clases que incluyen contenido arbitrario
- Clases que contienen una colección de objetos UIElement
- Clases que afectan a la apariencia de un objeto UIElement
- Clases que proporcionan comentarios visuales sobre un objeto UIElement
- Clases que permiten a los usuarios introducir texto
- Clases que muestran su texto
- Clases que dan formato al texto

© JMA 2012

Clases que incluyen contenido arbitrario

- Algunos controles pueden contener un objeto de cualquier tipo, como una cadena, un objeto DateTime o un UIElement que es un contenedor para elementos adicionales. Por ejemplo, un Button puede contener una imagen y algún texto; o CheckBox puede contener el valor de DateTime.Now.
- WPF tiene cuatro clases que pueden incluir contenido arbitrario y heredan de Control.
 - ContentControl: Un único objeto arbitrario.
 - HeaderedContentControl: Un encabezado y un único elemento; los dos son objetos arbitrarios.
 - ItemsControl: Una colección de objetos arbitrarios.
 - HeaderedItemsControl: Un encabezado y una colección de elementos; todos son objetos arbitrarios.

© JMA 2012

Controles que contienen un único objeto arbitrario

- La clase ContentControl contiene una parte única de contenido arbitrario, la propiedad Content o el contenido de la etiqueta.

Button, ButtonBase, CheckBox, ComboBoxItem, ContentControl, Frame, GridViewColumnHeader, GroupItem, Label, ListItem, ListViewItem, NavigationWindow, RadioButton, RepeatButton, ScrollViewer, StatusBarItem, ToggleButton, ToolTip, UserControl, Window

© JMA 2012

Controles que contienen un encabezado y contenido

- La clase HeaderedContentControl hereda de ContentControl y muestra contenido con un encabezado.
- Hereda la propiedad de contenido, Content, de ContentControl y define la propiedad Header que es del tipo Object; en consecuencia, ambos pueden ser un objeto arbitrario.
 - Expander
 - GroupBox
 - TabItem

© JMA 2012

Controles que contienen una colección de objetos arbitrarios

- La clase `ItemsControl` hereda de `Control` y puede contener varios elementos, como cadenas, objetos u otros elementos.
- Sus propiedades de contenido son `ItemsSource` y `Items`. `ItemsSource` se suele usar para llenar `ItemsControl` con una recolección de datos.
- Si no desea emplear una colección para llenar el `ItemsControl`, puede agregar elementos mediante la propiedad `Items`.
 - `Menu`, `MenuBase`, `ContextMenu`
 - `ComboBox`
 - `ListBox`
 - `ListView`
 - `TabControl`
 - `TreeView`
 - `Selector`
 - `StatusBar`

© JMA 2012

Clases que contienen una colección de objetos `UIElement`

- La clase `Panel` coloca y organiza los objetos `UIElement` secundarios. Su propiedad de contenido es `Children`.
 - `Canvas`
 - `DockPanel`
 - `Grid`
 - `TabPanel`
 - `ToolBarOverflowPanel`
 - `ToolBarPanel`
 - `UniformGrid`
 - `StackPanel`
 - `VirtualizingPanel`
 - `VirtualizingStackPanel`
 - `WrapPanel`

© JMA 2012

Clases que afectan a la apariencia de un objeto UIElement

- La clase **Decorator** aplica efectos visuales a un único objeto **UIElement** secundario o alrededor del mismo. Su propiedad de contenido es **Child**.
 - **AdornerDecorator**
 - **Border**
 - **BulletDecorator**
 - **ButtonChrome**
 - **ClassicBorderDecorator**
 - **InkPresenter**
 - **ListBoxChrome**
 - **SystemDropShadowChrome**
 - **Viewbox**
- Clases que proporcionan comentarios visuales sobre un objeto **UIElement**
 - La clase **Adorner** proporciona indicaciones visuales a un usuario.

© JMA 2012

Clases que contienen texto

- Clases que muestran su texto
 - Se pueden usar varias clases para mostrar texto sin formato o con formato.
 - Puede emplear **TextBlock** para mostrar cantidades pequeñas de texto.
 - Si desea mostrar grandes cantidades de texto, use los controles **FlowDocumentReader**, **FlowDocumentScrollView** o **FlowDocumentPageViewer**.
- Clases que permiten a los usuarios introducir texto
 - WPF proporciona tres controles primarios que permiten a los usuarios introducir texto. Cada control muestra el texto de manera diferente.
 - **TextBox**: Texto sin formato
 - **RichTextBox**: Texto con formato
 - **PasswordBox**: Texto oculto (se enmascaran los caracteres)
- Clases que dan formato al texto
 - **TextElement** y sus clases relacionadas le permiten dar formato al texto de los objetos **FlowDocument** y **TextBlock**.

© JMA 2012

Diseño

- **Canvas:**
 - los controles secundarios proporcionan su propio diseño.
- **DockPanel:**
 - los controles secundarios se alinean con los bordes del panel.
- **Grid:**
 - los controles secundarios se sitúan por filas y columnas.
- **StackPanel:**
 - los controles secundarios se apilan vertical u horizontalmente.
- **VirtualizingStackPanel:**
 - los controles secundarios se organizan en una vista "virtual" de una sola línea en sentido horizontal o vertical.
- **WrapPanel:**
 - los controles secundarios se sitúan por orden de izquierda a derecha y se ajustan a la línea siguiente cuando hay más controles de los que caben en la línea actual.

© JMA 2012

Propiedades asociadas

- Uno de los propósitos de una propiedad asociada es permitir que los diferentes elementos secundarios especifiquen valores únicos para una propiedad que en realidad está definida en un elemento primario.
- Una aplicación concreta de este escenario es hacer que los elementos secundarios informen al elemento primario de cómo se presentarán en la user interface (UI).
- La propiedad DockPanel.Dock se crea como una propiedad asociada porque se ha diseñado para establecerse en elementos contenidos dentro de un objeto DockPanel, en lugar de en el propio objeto DockPanel. La clase DockPanel define el campo estático DependencyProperty denominado DockProperty y, a continuación, proporciona los métodos GetDock y SetDock como descriptores de acceso públicos para la propiedad asociada.

© JMA 2012

Canvas

- El elemento Canvas permite la colocación del contenido de acuerdo con unas coordenadas x e y absolutas.
- Los elementos se pueden dibujar en una ubicación única; o, si ocupan las mismas coordenadas, el orden en que aparecen en el marcado determina el orden en el que se dibujan.
- Canvas proporciona la compatibilidad de diseño más flexible de todos los elementos Panel.
- Las propiedades Height y Width se utilizan para definir el área del lienzo, y a los elementos que contiene se les asignan coordenadas absolutas relativas al área del elemento Canvas primario.
- Cuatro propiedades adjuntas, Canvas.Left, Canvas.Top, Canvas.Right y Canvas.Bottom, permiten obtener un control muy preciso de la posición de los objetos dentro de un elemento Canvas, y gracias a ellas el programador puede colocar y organizar con precisión los elementos en la pantalla.

© JMA 2012

DockPanel

- El elemento DockPanel utiliza la propiedad adjunta DockPanel.Dock tal y como está establecida en los elementos de contenido secundarios para colocar el contenido a lo largo de los bordes de un contenedor.
- Cuando DockPanel.Dock se establece en Top o Bottom, coloca los elementos secundarios por encima o por debajo de los demás.
- Cuando la propiedad DockPanel.Dock está establecida en Left o en Right, coloca los elementos secundarios a la izquierda o a la derecha de los demás.
- La propiedad LastChildFill determina la posición del último elemento agregado como un elemento secundario de un elemento DockPanel.
- Puede utilizar DockPanel para colocar un grupo de controles relacionados, como un conjunto de botones. También puede utilizarlo para crear una UI "con paneles", similar a la de Microsoft Outlook.

© JMA 2012

Grid

- Define un área de cuadrícula flexible que está compuesta de columnas y filas.
- El elemento Grid combina la funcionalidad de una posición absoluta y de un control de datos tabular.
- Grid permite colocar con facilidad elementos y aplicarles estilo.
- Con Grid podrá definir agrupaciones flexibles de filas y columnas, e incluso dispone de un mecanismo para compartir información de tamaño entre varios elementos Grid.
- Grid agrega elementos basándose en un índice de fila y columna.
- Permite la disposición en capas del contenido secundario, lo que permite que haya más de un elemento dentro de una sola "celda".
- Los elementos secundarios de Grid se pueden colocar de manera absoluta en relación con el área de los límites de la "celda". Grid.Column, Grid.ColumnSpan, Grid.Row y Grid.RowSpan permiten indicar la celda y el número de celdas ocupadas.
- RowDefinitions y ColumnDefinitions controlan el número y tamaño de las filas y columnas.

© JMA 2012

StackPanel

- Un elemento StackPanel le permite "apilar" los elementos en una dirección asignada.
- De forma predeterminada, los elementos se apilan en dirección vertical.
- Se puede utilizar la propiedad Orientation para controlar el flujo del contenido en dirección horizontal.

© JMA 2012

WrapPanel

- WrapPanel se utiliza para colocar de izquierda a derecha los elementos secundarios en posición secuencial, y traslada el contenido a la línea siguiente cuando alcanza el borde de su contenedor primario.
- El contenido se puede orientar en sentido horizontal o vertical. WrapPanel resulta útil para los escenarios de user interface (UI) de flujo sencillo.
- También se puede utilizar para aplicar un tamaño uniforme a todos sus elementos secundarios.

© JMA 2012

Alineación, Márgenes y Relleno

- Los valores explícitos de las propiedades Width y Height tienen prioridad al fijar el ancho y alto de los elementos.
- Las propiedades HorizontalAlignment (Left, Right, Center o Stretch) y VerticalAlignment (Top, Center, Bottom o Stretch) describen cómo debería colocarse un elemento secundario dentro del espacio de diseño asignado del elemento primario.
- La propiedad Margin describe la distancia entre un elemento y su elemento secundario o del mismo nivel.
- La propiedad Padding describe la distancia entre un elemento y su contenido.
- Las propiedades MaxWidth, MinWidth, MaxHeight y MinHeight fijan el ancho y alto máximo y mínimo de los elementos.

© JMA 2012

Controles de WPF por función

- **Presentación y selección de fechas:**
 - Calendar y DatePicker.
- **Presentación de datos:**
 - DataGrid, ListView y TreeView.
- **Cuadros de diálogo:**
 - OpenFileDialog, PrintDialog y SaveFileDialog.
- **Información para el usuario:**
 - AccessText, Label, Popup, ProgressBar, StatusBar, TextBlock y ToolTip.
- **Documentos:**
 - DocumentViewer, FlowDocumentPageViewer, FlowDocumentReader, FlowDocumentScrollView y StickyNoteControl.
- **Entradas de lápiz digitales:**
 - InkCanvas y InkPresenter.
- **Multimedia:**
 - Image, MediaElement y SoundPlayerAction.

© JMA 2012

Controles de WPF por función

- **Diseño:**
 - Border, BulletDecorator, Canvas, DockPanel, Expander, Grid, GridView, GridSplitter, GroupBox, Panel, ResizeGrip, Separator, ScrollBar, ScrollViewer, StackPanel, Thumb, Viewbox, VirtualizingStackPanel, Window y WrapPanel.
- **Navegación:**
 - Frame, Hyperlink, Page, NavigationWindow y TabControl.
- **Botones:**
 - Button y RepeatButton.
- **Menús:**
 - ContextMenu, Menu yToolBar.
- **Entrada:**
 - TextBox, RichTextBox y PasswordBox.
- **Selección:**
 - CheckBox, ComboBox, ListBox, RadioButton y Slider.

© JMA 2012

Ribbon (4.5)

- Requiere agregar manualmente la referencia System.Windows.Controls.Ribbon
- La cinta de opciones es una barra de comandos que organiza las características de una aplicación en una serie de fichas en la parte superior de la ventana de la aplicación.
- La cinta reemplaza la barra de menús y las barras de herramientas tradicionales (algunos componentes de la cinta se dibujan en la barra de título de la ventana por lo que hay que heredar de RibbonWindow en lugar de Window).
- Cada cinta tiene un menú de aplicación, una barra de herramientas de acceso rápido, fichas, grupos y controles.
- Las fichas de cinta contienen grupos y cada grupo contiene controles. Los controles relacionados se pueden combinar aún más en grupos de controles.
- Los controles de cinta incluyen controles sencillos como botones, casillas y cuadros de texto; y controles de menú como cuadros combinados, botones de expansión y botones de menú.
- Además de los componentes necesarios, una cinta también puede incluir componentes opcionales, como fichas contextuales, información sobre herramientas mejorada y galerías.

© JMA 2012

Despliegue WPF

- Ensamblado .NET
 - Ejecutable tradicional Setup, ClickOnce
- Aplicación XBAP
 - Dentro del navegador
 - Modelo de navegación integrado con browser
- Loose XAML
 - Renderización directa en browser
 - Opciones interesantes: ASP.NET / XML + XSL
- Documento
 - Formato de documento XPS = Subset XAML

© JMA 2012

Modelos de GUI

- SDI: interfaz de documento único
 - Ventana principal (Window):
 - Controles (UserControl)
 - Ventanas emergentes (PopUP)
- MDI: interfaz de múltiples documentos
 - Soportado por herramientas de terceros.
- Navegación: tipo web
 - Ventana de navegación (NavigationWindow).
 - Páginas de contenido (Page) e hipervínculos (Hyperlink)
 - Marcos (Frame) y motor de navegación (NavigationService)

© JMA 2012

Ventanas secundarias

- Modeless, que no se cierran con la principal
`Window2 win = new Window2();
win.Show();`
- Modeless, que se cierran con la principal
`win = new Window2();
win.Owner = this;
win.Show();`
- Modal (Cuadro de diálogo)
`Window2 win = new Window2();
win.Owner = this;
if(win.ShowDialog() == true) // Aceptado (bool?)`

© JMA 2012

Cuadros de diálogo

- Heredan de la clase Window
- Configurar el aspecto de la ventana:
 - Mostrar Barra de título, ícono y menú Sistema para minimizar, maximizar, restaurar y cerrar el cuadro de diálogo.
 - Mostrar botones Cerrar, Minimizar, Maximizar y Restaurar
 - ResizeMode="NoResize"
WindowStartupLocation="CenterScreen"
ShowInTaskbar="False"
- Mostrar un botón Aceptar (IsDefault='true') y un botón Cancelar (IsCancel='true').
- Para Aceptar (this.DialogResult = true) o Cancelar (this.DialogResult = false).

© JMA 2012

Control de usuario

- La manera más sencilla de crear un control es crear una clase que deriva de UserControl.
- Es un modelo conveniente si desea generar el control agregando elementos existentes en él, similar a cómo crear una ventana, y si no necesita utilizar personalización compleja.
- El control no admitirá plantillas y, por tanto, no admite la personalización compleja.
- Un UserControl es un ContentControl, lo que significa que puede contener un único objeto de cualquier tipo (como una cadena, una imagen o un panel).
- Contenedor de controles:
`<ContentControl x:Name="UCHost" />`
- Abrir control como componente:
`UCHost.Content = new UserControl1();`
- Cerrar el control componente:
`UCHost.Content = null;`

© JMA 2012

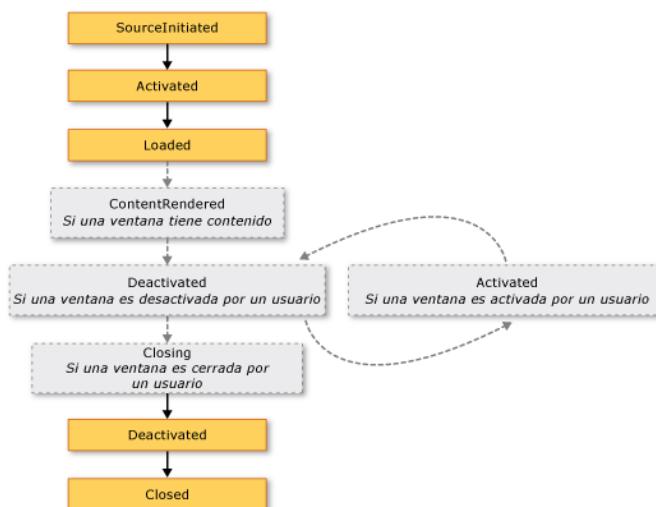
Popup

- El control Popup proporciona una manera de mostrar contenido en una ventana independiente que flota sobre la ventana de la aplicación actual de manera relativa a un elemento designado o a una coordenada de la pantalla.
- Un control Popup muestra su contenido cuando la propiedad IsOpen está establecida en true y permanece abierto hasta que la propiedad IsOpen se establece en false.
- Sin embargo, puede cambiar el comportamiento predeterminado estableciendo la propiedad StaysOpen en false, la ventana se cierra cuando un evento del mouse se produce fuera de la ventana de Popup.

```
<Popup x:Name="popDlg" StaysOpen="False" PopupAnimation="Slide" Placement="Right"
PlacementTarget="{Binding ElementName=btnAbrirPOP, Mode=OneWay}"
AllowsTransparency="True" Width="200" Height="100">
    <Border CornerRadius="10" Background="#FFFDFDD4" Padding="5">
        <Grid>
            <Button x:Name="btnCerrarPOP" Content="Cerrar"
HorizontalContentAlignment="Right" VerticalAlignment="Bottom" HorizontalAlignment="Right"
Click="btnCerrarPOP_Click"/>
        </Grid>
    </Border>
</Popup>
```

© JMA 2012

Eventos de duración de ventana



© JMA 2012

Modelo de Navegación

- Page: Encapsula una página de contenido a la que pueda navegar, se diseña de forma similar a las ventanas.
- Hyperlink: Permite que un usuario inicie la navegación en un objeto Page determinado.
- NavigationService: Encargado de la búsqueda y descarga de la página.
- Diario (Journal): Implementa un servicio del historial de navegación que almacena una entrada para cada fragmento de contenido al que se ha navegado previamente.
- NavigationWindow: Proporciona una ventana principal para aplicaciones independientes de navegación.
- Frame: Proporciona un contenedor de páginas para su uso en ventanas u otras páginas.

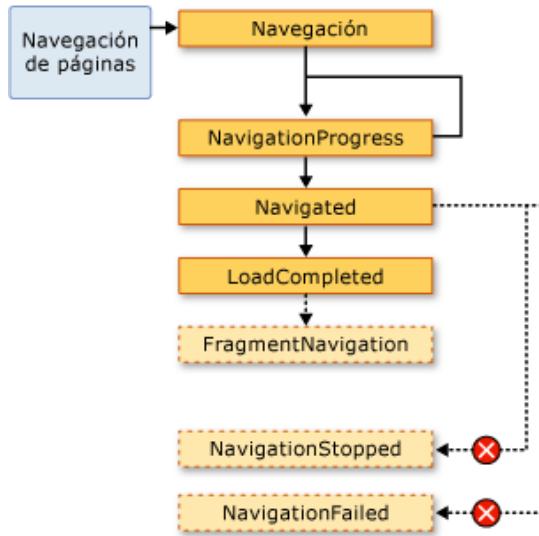
© JMA 2012

Navegación

- Navegación por hipervínculos
`<Hyperlink NavigateUri="pag2.xaml">...</Hyperlink>`
- Navegación por fragmentos
`<TextBlock Name="Fragmento1">...</TextBlock>`
`<Hyperlink
NavigateUri="pag2.xaml#Fragmento1">...</Hyperlink>`
- Navegación por código :
`ns = NavigationService.GetNavigationService(refPage);`
`ns = this.NavigationService;`
`ns.Navigate(new Page2());`
`ns.Navigate(new Uri("Page2.xaml", UriKind.Relative));`

© JMA 2012

Duración de la navegación



© JMA 2012

Navegación con el diario

- Dos pilas con el historial de las páginas: una pila de retroceso y una pila de avance.
- Barra de navegación
- Declarativa por comandos (NavigationCommands):


```

<Hyperlink Command="NavigationCommands.BrowseBack" ... >
<Hyperlink Command="NavigationCommands.BrowseForward"
... >
```
- Imperativa por código (NavigationService):
 - GoBack, GoForward
 - CanGoBack, CanGoForward

© JMA 2012

Estado de la página

- Activación:
`<Page ... KeepAlive="True" ...>`
- Solo se crean una vez, hay que mover la lógica de inicialización a la que hay que llamar cada vez que se navega a la página al controlador del evento Loaded.
- Controles que mantienen el estado:
 - CheckBox, ComboBox , Expander, Frame, ListBox, ListBoxItem, MenuItem, ProgressBar, RadioButton, Slider, TabControl, TabItem, TextBox
- Para que recuerde información adicional:
 - Crear propiedades de dependencia con el marcador de metadatos de FrameworkPropertyMetadata.Journal=true.
 - Implementar el interfaz IProvideCustomContentState.

© JMA 2012

Marcos

`<Frame Source="FramePage1.xaml" ... />`

- Dispone de su propio contenido, NavigationService, diario y barra de navegación.
- JournalOwnership=
 - OwnsJournal (Diario propio)
 - UsesParentJournal (Diario del contenedor)
- NavigationUIVisibility="Hidden"

© JMA 2012

Navegación estructurada

- Se denomina navegación estructurada cuando se necesita:
 - Pasar parámetros a la página llamada.
 - Devolver información de estado o datos a la página que llama.
 - Quitar del historial de navegación la página llamada cuando vuelve a la página que llama.
- `PageFunction<T>`, hereda de `Page` y representa un tipo especial de página que:
 - implementa los mecanismos básicos de navegación estructurada.
 - permite tratar la navegación a una página de forma similar a llamar a un método.

© JMA 2012

PageFunction<T>

```
<PageFunction ...
    xmlns:local="clr-namespace: ..."
    x:TypeArguments="local:ParamType"
    ... >

public partial class MyPageFunction : PageFunction<ParamType> {
    public MyPageFunction(ParamType param) {
        InitializeComponent();
        ...
        p = param;
        ...
    }
    ...
}
```

© JMA 2012

Devolución de resultado

- PageFunction<T> implementa el método OnReturn para volver al llamador, pasando un valor devuelto a través de un objeto ReturnEventArgs<T>.

```
void ok_Click(object sender, RoutedEventArgs e) {  
    OnReturn(new ReturnEventArgs<RsltType>(rslt));  
}  
void cancel_Click(object sender, RoutedEventArgs e) {  
    OnReturn(null);  
}
```

© JMA 2012

Pasar parámetros y devolver el resultado

```
void pageFunctionHyperlink_Click(object sender, RoutedEventArgs e) {  
    var param = new ParamType(...);  
    MyPageFunction page = new MyPageFunction(param);  
    page.Return += pageFunction_Return;  
    this.NavigationService.Navigate(page);  
}  
  
void pageFunction_Return(object sender, ReturnEventArgs<string> e) {  
    estado = (e != null ? "Accepted" : "Canceled");  
    if (e != null) {  
        rslt = e.Result;  
        ...  
    }  
}
```

© JMA 2012

Quitar la página del diario

- La navegación estructurada suele ser una actividad aislada; cuando vuelve la página llamada, la página que llama debe crear y navegar a una nueva página que llama para capturar más datos.
- De forma predeterminada, una función de página se quita automáticamente del historial cuando se llama a OnReturn, porque RemoveFromJournal está a true.
- Para mantener una función de página en el historial de navegación después de llamar a OnReturn, hay que establecer RemoveFromJournal a false.
`<PageFunction ... RemoveFromJournal="False" ... >`

© JMA 2012

Objeto Application

- Crear y administrar la infraestructura común de las aplicaciones.
- Realizar el seguimiento e interactuar con la duración de la aplicación.
- Recuperar y procesar los parámetros de la línea de comandos.
- Compartir propiedades y recursos del ámbito de la aplicación.
- Detectar y responder a las excepciones no controladas.
- Devolver códigos de salida.
- Administrar las ventanas en las aplicaciones independientes (vea Información general sobre ventanas de WPF).
- Realizar el seguimiento y administrar la navegación (vea Información general sobre navegación).
- Se define en:

App.xaml

App.cs

```
public partial class App : Application { }
```

© JMA 2012

Objeto inicial

- Pagina inicial:
`<Application ... StartupUri=" MainPage.xaml" ... />`
- Ventana principal:
`<Application ... StartupUri=" MainWindow.xaml" ... />`
Application.MainWindow
- Evento:
`<Application ... Startup="app_Startup" ... />`
- SplashScreen
 - Imagen WIC (BMP, GIF, JPEG, PNG o TIFF)
 - Propiedades → Acción de compilación=SplashScreen

© JMA 2012

Cierre de la aplicación

- Para facilitar la administración del cierre de la aplicación, Application proporciona el método Shutdown, la propiedad ShutdownMode y los eventos SessionEnding y Exit.
- Modo de apagado (ShutdownMode):
 - OnLastWindowClose
 - OnMainWindowClose
 - OnExplicitShutdown

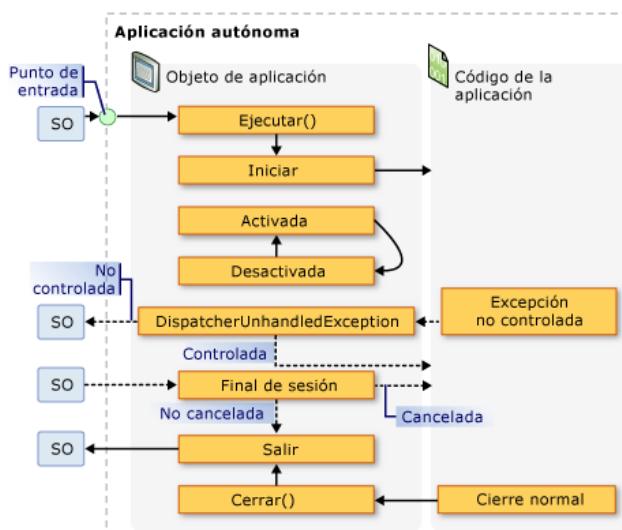
© JMA 2012

Eventos

- LoadCompleted
 - Se produce cuando se ha cargado y analizado el contenido
- Activated
 - Se inicia y muestra un objeto Window.
 - Cuando el usuario vuelve a la aplicación.
- Deactivated
 - Un usuario cambia a otra aplicación desde la actual.
 - Cuando se cierra la aplicación.
- SessionEnding
 - Se produce cuando el usuario finaliza la sesión de Windows cerrando sesión o apagando el sistema operativo.
- Exit
 - Se produce justo antes de que se cierre una aplicación y no se puede cancelar.
- DispatcherUnhandledException
 - Cuando llega una excepción que no controlada al objeto aplicación.

© JMA 2012

Eventos de duración de la aplicación



© JMA 2012

Servicios de Aplicación

- App.Current: objeto Application actual (singleton).
- App.Current.Properties: Diccionario con las propiedades globales del ámbito de la aplicación.
- Recursos globales:
 - <Application.Resources>...</Application.Resources>
 - Application.Current.Resources["ApplicationScopeResource"]
- Administración de ventanas:
 - App.Current.MainWindow: ventana principal de la aplicación.
 - App.Current.Windows: Colección de ventanas instanciadas de una aplicación.
- Administración de la navegación:
 - Intercepta los eventos de Navigating, Navigated, NavigationProgress, NavigationFailed, NavigationStopped, LoadCompleted, FragmentNavigation

© JMA 2012

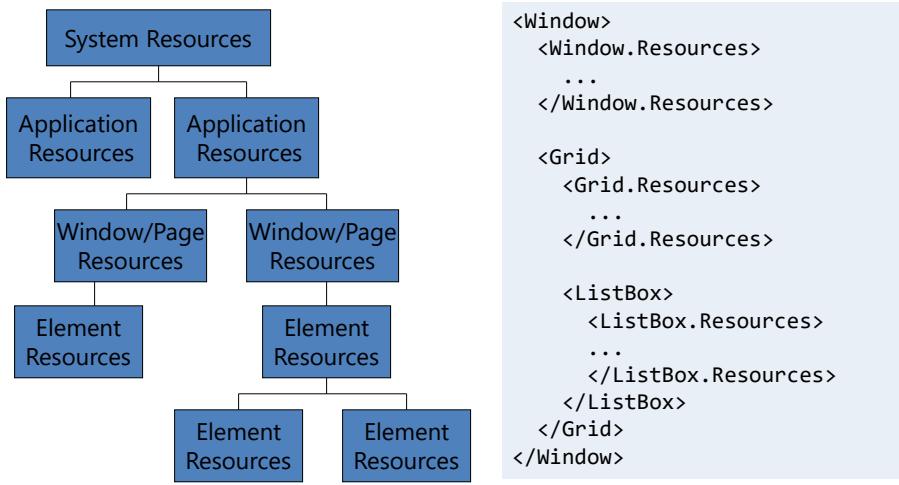
Áreas Resources

- El área Resources expone un almacén para los recursos compartidos.
- En el área de recursos pueden definirse fuentes de datos, estilos, plantillas, ... que posteriormente podrán ser enlazados.
- Si los recursos cambian, el sistema de recursos asegura que las propiedades de elemento enlazadas a esos recursos se actualizan automáticamente para reflejar el cambio.
- Todos los UIElement puede definir su propiedad recursos.
- Todos los recursos deben estar identificados por una clave x:Key para su referencia.
- Las referencias se establecen como:
 - StaticResource: recuperar solamente
 - DynamicResource: la instancia puede cambiar

```
<Window.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
</Window.Resources>
...
<Rectangle Fill="{StaticResource MyBrush}" ...>
```

© JMA 2012

Jerarquía de recursos



© JMA 2012

SplashScreen

- La SplashScreen muestra una imagen en una ventana de inicio o pantalla, cuando se inicia una aplicación de WPF.
- La SplashScreen clase puede mostrar cualquier formato de imagen compatible con Windows Imaging Component (WIC).
- Por ejemplo, puede utilizar el formato BMP, GIF, JPEG, PNG o TIFF.
- Si la imagen es un archivo PNG e incluye un canal alfa, la imagen se representa utilizando la transparencia definida en el canal alfa.
- Para agregar una imagen existente como una pantalla de presentación
 1. Cree o busque la imagen que desea utilizar para la pantalla de presentación. Puede utilizar cualquier formato de imagen compatible con Windows Imaging Component (WIC). Por ejemplo, se puede utilizar el formato BMP, GIF, JPEG, PNG o TIFF.
 2. Agregue el archivo de imagen al proyecto de aplicación de WPF.
 3. En el Explorador de soluciones, seleccione la imagen.
 4. En la ventana Propiedades, haga clic en la flecha de la lista desplegable de la propiedad Acción de compilación.
 5. Seleccione SplashScreen en la lista desplegable.

© JMA 2012

Empaquetar URI en WPF

- En Windows Presentation Foundation (WPF), se utilizan uniform resource identifiers (URIs) para identificar y cargar archivos de muchas maneras, incluidas las que figuran a continuación:
 - Especificando la user interface (UI) que se va a mostrar cuando se inicie una aplicación por primera vez.
 - Cargando imágenes.
 - Navegando a páginas.
 - Cargando archivos de datos no ejecutables.
- Además, se pueden usar URIs para identificar y cargar archivos desde diversas ubicaciones, como las que figuran a continuación:
 - El ensamblado actual.
 - Un ensamblado al que se hace referencia.
 - Una ubicación relativa a un ensamblado.
 - El sitio de origen de la aplicación.

© JMA 2012

Empaquetar URI en WPF

- El pack URI para un archivo de recursos compilado en un ensamblado al que se hace referencia utiliza la siguiente autoridad y ruta de acceso:
- Autoridad: application:///.
- Ruta de acceso: nombre de un archivo de recursos compilado en un ensamblado al que se hace referencia. La ruta de acceso debe tener el formato siguiente:
nombreCortoDeEnsamblado[;Versión][;clavePública];component/rutaDeAcceso
 - nombreCortoDeEnsamblado: nombre corto del ensamblado al que se hace referencia.
 - ;Versión [opcional]: versión del ensamblado al que se hace referencia y que contiene el archivo de recursos. Se utiliza cuando hay cargados dos o más ensamblados a los que se hace referencia con el mismo nombre corto.
 - ;clavePública [opcional]: clave pública utilizada para firmar el ensamblado al que se hace referencia. Se utiliza cuando hay cargados dos o más ensamblados a los que se hace referencia con el mismo nombre corto.
 - ;component: especifica que la referencia al ensamblado se hace desde el ensamblado local.
 - /rutaDeAcceso: nombre del archivo de recursos, incluida su ruta de acceso relativa a la carpeta raíz del proyecto del ensamblado al que se hace referencia

```
// Absolute URI (default)
Uri absoluteUri = new Uri("pack://application:,,,/File.xaml", UriKind.Absolute);
// Relative URI
Uri relativeUri = new Uri("/File.xaml", UriKind.Relative);
```

© JMA 2012

PROPIEDADES, EVENTOS Y COMANDOS

© JMA 2012

Propiedades de dependencia

- WPF proporciona un conjunto de servicios que se pueden utilizar para extender la funcionalidad de una propiedad de CLR: las propiedades de dependencia.
- El propósito de las propiedades de dependencia es proporcionar una manera de calcular el valor de una propiedad en función del valor de otras entradas.
- Estas otras entradas pueden incluir propiedades del sistema tales como temas y preferencias del usuario, mecanismos de determinación de propiedad Just-In-Time tales como el enlace de datos y las animaciones o guiones gráficos, plantillas del uso múltiple tales como recursos y estilos, o valores conocidos a través de relaciones de elementos primarios-secundarios con otros elementos del árbol de elementos.
- Además, una propiedad de dependencia se puede implementar para que proporcione validación autónoma, valores predeterminados, devoluciones de llamada que supervisen los cambios de otras propiedades y un sistema que pueda forzar valores de la propiedad en función de información que puede estar disponible en tiempo de ejecución.
- Las clases derivadas también pueden cambiar algunas características concretas de una propiedad existente invalidando metadatos de propiedades de dependencia, en lugar de reemplazar la implementación real de propiedades existentes o crear propiedades nuevas.

© JMA 2012

Respaldo de propiedades de dependencia

- WPF extienden la funcionalidad de propiedad CLR proporcionando un tipo que respalda una propiedad, como implementación alternativa al modelo estándar de respaldar la propiedad con un campo privado.
- El nombre de este tipo es DependencyProperty.
- El otro tipo importante que define WPF es DependencyObject.DependencyObject que define la clase base que puede registrar y poseer una propiedad de dependencia.
- Pasos:
 1. Opcionalmente, preparar metadatos.
 2. Crear y registrar la propiedad de dependencia.
 3. Crear el contenedor CLR de la propiedad.

© JMA 2012

Propiedad de dependencia

- Una propiedad de dependencia es un atributo (campo) público, de clase, de solo lectura y de tipo DependencyProperty.
- El nombre de la propiedad de dependencia debe utilizar el notación Pascal con el sufijo Property.


```
public static readonly DependencyProperty
MyProperty = DependencyProperty.Register(
    "MyProperty", // Nombre sin sufijo
    typeof(int), // Tipo de datos de la propiedad
    typeof(ownerclass), // Clase a la que pertenece la clase
    new PropertyMetadata(0) // Metadatos adicionales
);
```

© JMA 2012

Contenedor CLR

- Implementación como propiedad de instancia de la propiedad de dependencia.
- Utiliza el identificador de propiedad de dependencia sin el sufijo Property, utilizando las llamadas a GetValue y SetValue para la lectura y escritura de la propiedad, proporcionando así el respaldo para la propiedad utilizando el sistema de propiedades de WPF.
- La propiedad debe ser de tipo declarado a registrar la propiedad de dependencia.

```
public int MyProperty {  
    get { return (int)GetValue(MyPropertyProperty); }  
    set { SetValue(MyPropertyProperty, value); }  
}
```

© JMA 2012

Metadatos

- Se crean como una instancia de PropertyMetadata o de la más especializada FrameworkPropertyMetadata.
- Permiten establecer:
 - valor predeterminado de la propiedad de dependencia.
 - que afectan al diseño del elemento (AffectsArrange, AffectsMeasure, AffectsRender).
 - que afectan al diseño del elemento contenedor (AffectsParentArrange, AffectsParentMeasure).
 - modo de enlace de datos (IsNotDataBindable, BindsTwoWayByDefault)
 - métodos de devolución de llamada para propagar cambios o convertir el valor.

© JMA 2012

Métodos de devolución de llamada

- **PropertyChangedCallback:** La devolución de llamada de cambio de propiedad para el valor Current se utiliza para reenviar el cambio a otras propiedades dependientes, invocando explícitamente las devoluciones de llamada de forzado de valores registradas para esas otras propiedades:

```
private static void OnCurrentReadingChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e) {
    d.CoerceValue(MinReadingProperty);
    d.CoerceValue(MaxReadingProperty);
}
```

- **CoerceValueCallback:** La devolución de llamada de forzado de valores comprueba los valores de las propiedades de las que depende potencialmente la propiedad actual y fuerza el valor actual si es necesario:

```
private static object CoerceCurrentReading(DependencyObject d, object value) {
    Gauge g = (Gauge)d;
    double current = (double)value;
    if (current < g.MinReading) current = g.MinReading;
    if (current > g.MaxReading) current = g.MaxReading;
    return current;
}
```

© JMA 2012

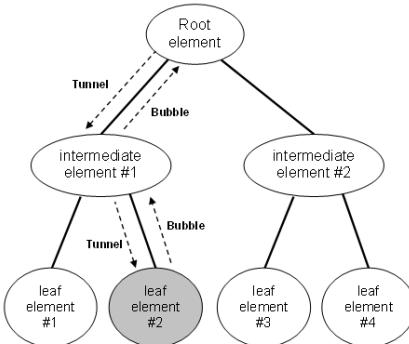
Eventos enrutados

- **Definición funcional:** un evento enrulado es un tipo de evento que puede invocar controladores o varios agentes de escucha en un árbol de elementos, en lugar de simplemente en el objeto que lo desencadenó.
- **Definición de implementación:** Un evento enrulado es un evento CLR que está respaldado por una instancia de la clase RoutedEvent y que es procesado por el sistema de eventos de Windows Presentation Foundation (WPF).
- Una aplicación típica de WPF contiene muchos elementos, estos elementos se relacionan entre sí a través de un árbol de elementos. En función de la definición del evento, la ruta de eventos puede viajar en cualquiera de las dos direcciones, pero generalmente viaja partiendo del elemento de origen y, a continuación, "se propaga" en sentido ascendente por el árbol de elementos hasta que llega a la raíz (normalmente una página o una ventana).

© JMA 2012

Enrutamiento de eventos

- Directo (Direct): sólo el propio elemento de origen tiene la oportunidad de invocar controladores como respuesta. Esto es análogo al "enrutamiento" utilizado por Windows Forms para los eventos.
- Propagación (Bubbled): se invocan los controladores de eventos en el origen del evento. A continuación, el evento enrulado va pasando por los elementos primarios sucesivos hasta alcanzar la raíz del árbol de elementos.
- Túnel (Tunneled): inicialmente, se invocan los controladores de eventos en la raíz de árbol de elementos. A continuación, el evento enrulado viaja a través de los elementos secundarios sucesivos a lo largo de la ruta, hacia el elemento del nodo que es el origen del evento enrulado (el elemento que desencadenó el evento enrulado).



© JMA 2012

Definir Eventos enrutados

- Al igual que en las propiedades de dependencia hay seguir los pasos:
 1. Opcionalmente, preparar metadatos.
 2. Crear y registrar el evento enrulado.
 3. Crear el contenedor CLR del evento.

```
public static readonly RoutedEvent TapEvent =
EventManager.RegisterRoutedEvent(
    "Tap", RoutingStrategy.Bubble, typeof(RoutedEventHandler),
    typeof(MyButtonSimple));
```

```
public event RoutedEventHandler Tap {
    add { AddHandler(TapEvent, value); }
    remove { RemoveHandler(TapEvent, value); }
}
```

© JMA 2012

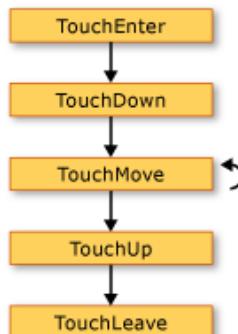
Entrada táctil y manipulación

- El nuevo hardware y API del sistema operativo Windows 7 proporcionan a las aplicaciones la capacidad de recibir varias entradas táctiles simultáneamente.
- WPF habilita las aplicaciones para detectar y responder a la entrada táctil de una manera similar a otra entrada, como el mouse o teclado, y generar los eventos cuando se produce la entrada táctil.
- WPF expone dos tipos de eventos cuando se produce un toque o entrada táctil: los eventos de toque y eventos de manipulación.
- Los eventos de toque proporcionan los datos sin procesar de cada dedo en una pantalla táctil y su movimiento.
- Los eventos de manipulación interpretan la entrada como acciones.

© JMA 2012

Eventos Touch

- Las clases base, UIElement, UIElement3D y ContentElement, definen los eventos a los que se puede suscribir para que la aplicación responda a la entrada táctil.
- Los eventos Touch son útiles cuando la aplicación interpreta la entrada táctil como algo distinto de la manipulación de un objeto.
 - TouchDown
 - TouchMove
 - TouchUp
 - TouchEnter
 - TouchLeave
 - PreviewTouchDown
 - PreviewTouchMove
 - PreviewTouchUp
 - GotTouchCapture
 - LostTouchCapture



© JMA 2012

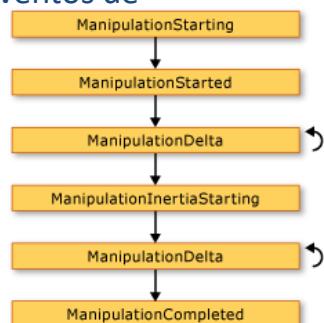
Eventos de manipulación

- A diferencia de los eventos Touch que simplemente notifican la posición de la entrada táctil, los eventos de manipulación notifican cómo se puede interpretar la entrada.
- Hay tres tipos de manipulaciones:
 - Traducción: Coloque un dedo en un objeto y mueva el dedo por la pantalla táctil para invocar una manipulación de traducción. Esto normalmente mueve el objeto.
 - Expansión: Coloque dos dedos en un objeto y acerque y aleje los dedos uno de otro para invocar una manipulación de expansión. Esto normalmente cambia el tamaño del objeto.
 - Rotación: Coloque dos dedos en un objeto y rote los dedos para invocar una manipulación de rotación. Esto normalmente gira el objeto.
- Se puede producir más de un tipo de manipulación simultáneamente.

© JMA 2012

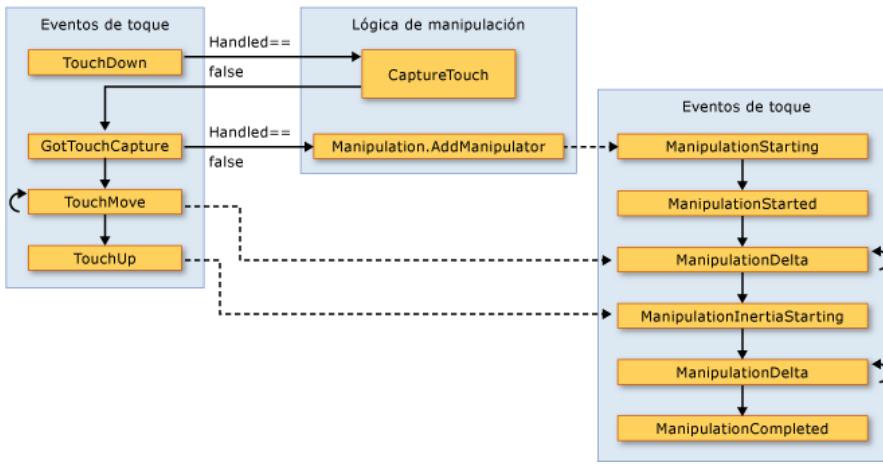
Eventos de manipulación

- `UIElement` define los siguientes eventos de manipulación.
 - `ManipulationStarting`
 - `ManipulationStarted`
 - `ManipulationDelta`
 - `ManipulationInertiaStarting`
 - `ManipulationCompleted`
 - `ManipulationBoundaryFeedback`
- La propiedad `ManipulationContainer` establece el contenedor al que son relativos todos los cálculos y eventos de manipulación.



© JMA 2012

Relación entre los eventos de manipulación y toque



© JMA 2012

Comandos

- Los comandos habilitan el control de entrada en un nivel más semántico que la entrada del dispositivo por eventos.
- Los comandos son directivas simples, como Cut, Copy, Paste u Open.
- Los comandos son útiles para centralizar la lógica de comandos, se podría obtener acceso al mismo comando desde un control Menu, unToolBar, a través de un método abreviado de teclado, ...
- Los comandos también proporcionan un mecanismo para deshabilitar los controles cuando el comando deja de estar disponible.
- WPF proporciona una biblioteca de comandos comunes que consta de ApplicationCommands, MediaCommands, ComponentCommands, NavigationCommands y EditingCommands; o también puede definir la suya propia.

© JMA 2012

RoutedCommand

- RoutedCommand es la implementación WPF de la interfaz ICommand.
- Cuando se ejecuta RoutedCommand, se desencadenan los eventos PreviewExecuted y Executed en el destino del comando, que se tunelizan y se propagan por el árbol de elementos al igual que cualquier otra entrada.
- Si no se establece ningún destino de comando, el elemento que tenga el foco de teclado será el destino del comando.
- La lógica que ejecuta el comando está asociada a un elemento CommandBinding. Cuando un evento Executed alcanza un elemento CommandBinding para ese comando en concreto, se llama a ExecutedRoutedEventHandler en CommandBinding. Este controlador ejecuta la acción del comando.

© JMA 2012

Comando enrulado

- El modelo de comando enrulado de WPF se puede descomponer en cuatro conceptos básicos:
 - El comando: es la acción que se va a ejecutar.
 - El origen del comando: es el objeto que invoca el comando.
 - El destino del comando: es el objeto en el que se ejecuta el comando.
 - El enlace del comando: es el objeto que asigna la lógica de comando al comando.

© JMA 2012

El comando

- Los comandos de WPF se crean mediante la implementación de la interfaz `ICommand`.
- `ICommand` expone dos métodos, `Execute` y `CanExecute`, y un evento, `CanExecuteChanged`.
 - `Execute` realiza las acciones que están asociadas al comando.
 - `CanExecute` determina si el comando se puede ejecutar en el destino del comando actual.
 - El evento `CanExecuteChanged` se produce si el administrador de comandos que centraliza las operaciones de comandos detecta un cambio en el origen del comando que podría invalidar un comando que se ha iniciado pero que el enlace de comando aún no ha ejecutado.
- La implementación de WPF de `ICommand` es la clase `RoutedCommand` y el centro de interés de esta información general.

```
public interface ICommand {
    void Execute(object parameter);
    bool CanExecute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

© JMA 2012

Orígenes de comando

- Un origen de comando es el objeto que invoca el comando.
- Los orígenes de comando en WPF implementan generalmente la interfaz `ICommandSource`.
- `ICommandSource` expone tres propiedades:
 - `Command` es el comando a ejecutar cuando se invoca el origen de comando.
 - `CommandTarget` es el objeto en el que se ejecuta el comando. Hay que destacar que en WPF la propiedad `CommandTarget` de `ICommandSource` solamente es aplicable cuando la interfaz `ICommand` es un objeto `RoutedCommand`. Si se establece la propiedad `CommandTarget` en un objeto `ICommandSource` y el comando correspondiente no es de tipo `RoutedCommand`, se omite el destino del comando. Si no se establece `CommandTarget`, el destino del comando será el elemento que tenga el foco del teclado.
 - `CommandParameter` es un tipo de datos definido por el usuario utilizado para pasar información a los controladores que implementan el comando.

© JMA 2012

Enlace del comando

- Un objeto `CommandBinding` asocia un comando con los controladores de eventos que implementan el comando.
- La clase `CommandBinding` contiene una propiedad `Command`, y eventos `PreviewExecuted`, `Executed`, `PreviewCanExecute` y `CanExecute`.

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
        Executed="OpenCmdExecuted"
        CanExecute="OpenCmdCanExecute"/>
</Window.CommandBindings>
```

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e) { ... }
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}
```

© JMA 2012

Destino de comando

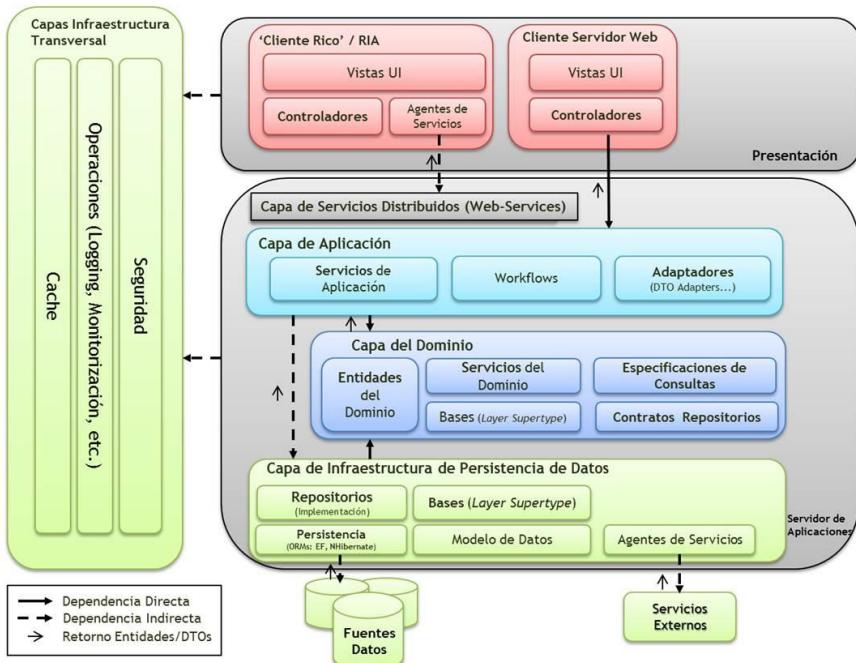
- El destino de comando es el elemento en el que se ejecuta el comando.
- En lo referente a un objeto `RoutedCommand`, el destino de comando es el elemento en el que se inicia el enrutado de `CanExecute` y `Executed`.
- La propiedad `CommandTarget` de `ICommandSource` solamente se aplica cuando `ICommand` es `RoutedCommand`.
- Si se establece la propiedad `CommandTarget` en un objeto `ICommandSource` y el comando correspondiente no es de tipo `RoutedCommand`, se omite el destino del comando.

© JMA 2012

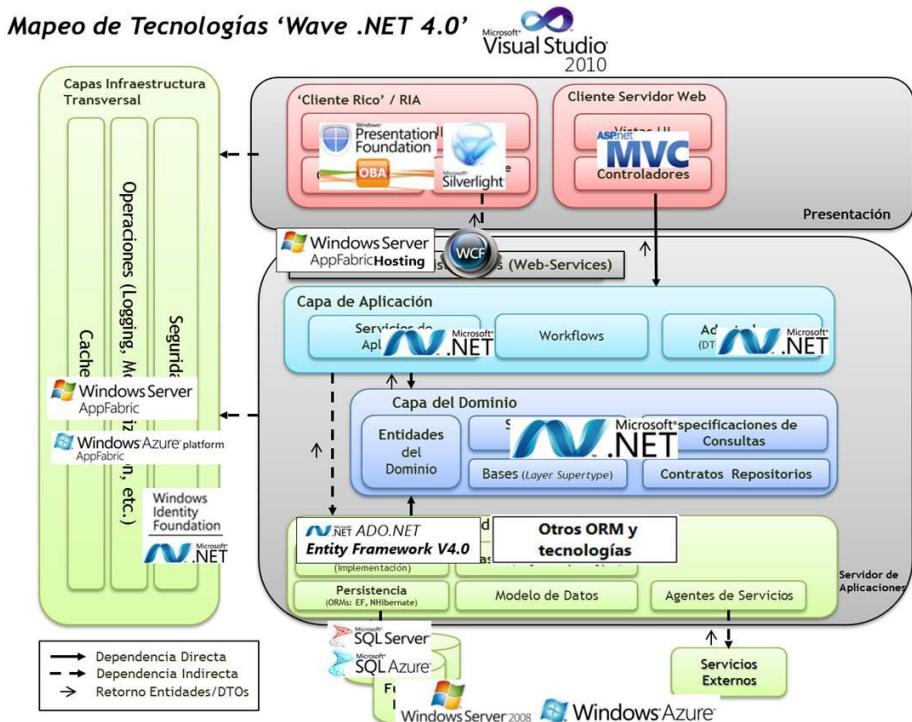
ENLACE A DATOS

© IMA 2012

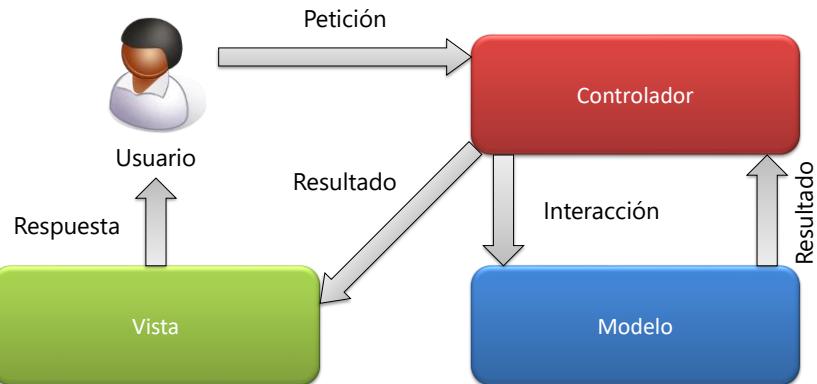
Arquitectura N-Capas con Orientación al Dominio



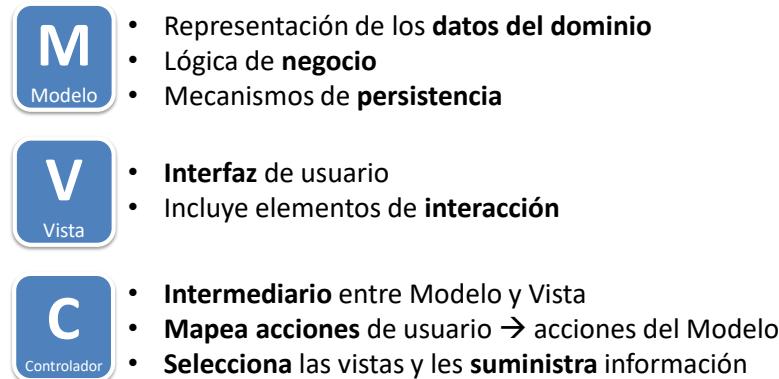
© IMA 2012



El patrón MVC en ASP.NET



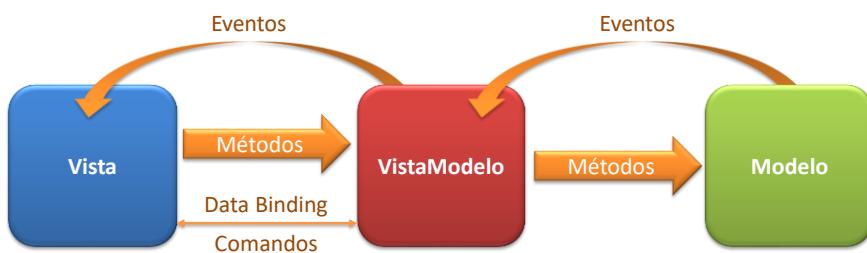
El patrón MVC



© JMA 2012

Patrón de Diseño Model View ViewModel (MVVM)

- El **Modelo** es la entidad que representa el concepto de negocio.
- La **Vista** es la representación gráfica del control o un conjunto de controles que muestran el Modelo de datos en pantalla.
- La **VistaViewModel** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



© JMA 2012

Características MVVM

- La vista y el modelo de vista se comunican mediante enlaces de datos, métodos, propiedades, eventos y mensajes.
- El modelo de vista expone propiedades y comandos además de modelos.
- La vista se encarga de sus propios eventos relacionados con la interface al usuario y los pasa al modelo de vista mediante comandos.
- Los modelos y propiedades en el modelo de vista son actualizados desde la vista usando enlaces de datos bidireccionales.

© JMA 2012

¿Cuáles son los beneficios del patrón MVVM?

- *Separación de vista / presentación.*
- *Permite las pruebas unitarias:* como la lógica de presentación está separada de la vista, podemos realizar pruebas unitarias sobre la VistaModelo.
- *Mejora la reutilización de código.*
- *Soporte para manejar datos en tiempo de diseño.*
- *Múltiples vistas:* la VistaModelo puede ser presentada en múltiples vistas, dependiendo del rol del usuario por ejemplo.

© JMA 2012

Patrones

- Observable
 - IObservable
 - IObservableCollection
 - INotifyPropertyChanged
 - ObservableCollection
- Command
 - ICommand
- Editable
 - IEditableObject

© JMA 2012

Observable Class

```
public class Entidad : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(PropertyChangedEventArgs e) {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }
    private string propiedad;
    public string Propiedad {
        get { return propiedad; }
        set {
            if (propiedad != value) {
                propiedad = value;
                OnPropertyChanged("Propiedad");
            }
        }
    }
}
```

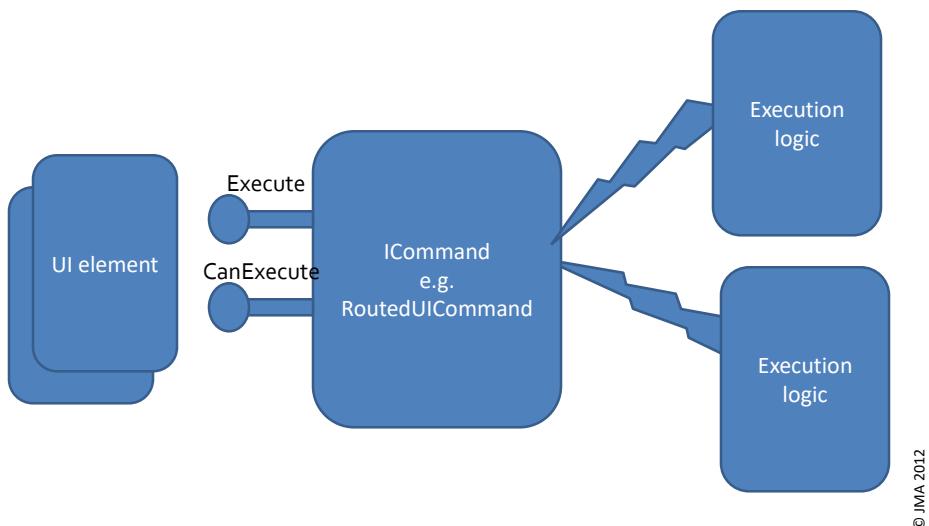
© JMA 2012

Observable Class (v 4.5)

```
public abstract class ObservableBase : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void NotifyPropertyChanged([CallerMemberName] string propertyName = null) {
        Debug.Assert(
            string.IsNullOrEmpty(propertyName) ||
            (GetType().GetProperty(propertyName) != null),
            "Check that the property name exists for this instance.");
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
public class Entidad : ObservableBase {
    private string propiedad;
    public string Propiedad {
        get { return propiedad; }
        set {
            if (propiedad != value) {
                propiedad = value;
                NotifyPropertyChanged();
            }
        }
    }
}
```

© JMA 2012

Patrón Command



© JMA 2012

DelegateCommand

```
public class DelegateCommand : ICommand {  
    public DelegateCommand(Action<object> ExecuteHandler,  
        Func<object,bool> CanExecuteHandler) { ... }  
    public DelegateCommand(Action<object> ExecuteHandler) { }  
}  
  
public DelegateCommand Comando {  
    get {  
        return new DelegateCommand(  
            cmd => { ... },  
            cmd => { return true; }  
        );  
    }  
}
```

<https://www.wpftutorial.net/DelegateCommand.html>

© JMA 2012

Frameworks de MVVM

- Prism – Microsoft
- MVVM Light
- Simple MVVM
- Caliburn

© JMA 2012

Enlace de datos (DataBinding)

- *El enlace de datos es el proceso que establece una conexión entre la UI de la aplicación y la lógica del negocio*

Los objetos controles tienen funciones integradas que permiten definir de forma flexible elementos de datos individuales o colecciones de elementos de datos.

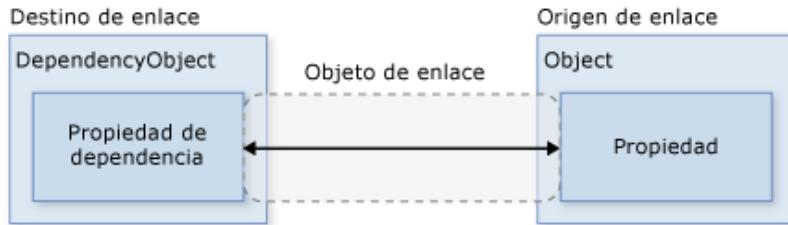
© JMA 2012

Ventajas del DataBinding

- Un mayor número de propiedades que admiten de forma inherente el enlace de datos.
- Una representación flexible de los datos en la UI.
- La separación bien definida de la lógica del negocio de la UI.

© JMA 2012

... Enlace de Datos

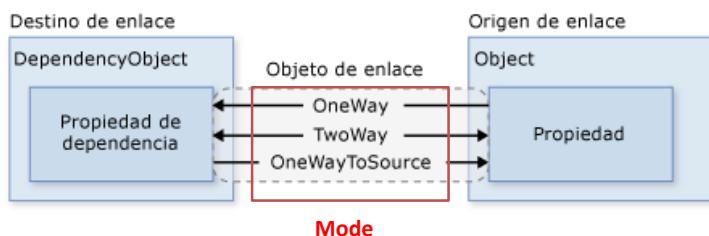


El enlace de datos es esencialmente el puente entre el destino del enlace y el origen del enlace.

© JMA 2012

Dirección del flujo de datos

Se controla este comportamiento estableciendo la propiedad Mode del objeto Binding



© JMA 2012

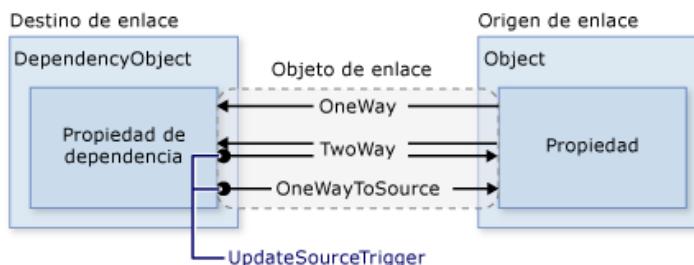
Dirección del flujo de datos

- El enlace **OneWay** permite que los cambios en la propiedad de origen actualicen automáticamente la propiedad de destino, pero los cambios en la propiedad de destino no se propagan de nuevo a la propiedad de origen.
- El enlace **TwoWay** permite que los cambios realizados en la propiedad de origen o en la de destino se actualicen automáticamente en el otro.
- **OneWayToSource** es el enlace inverso de OneWay; actualiza la propiedad de origen cuando cambia la propiedad de destino.
- El enlace **OneTime** permite que la propiedad de origen inicialice la propiedad de destino, pero los demás cambios no se propagan.

© JMA 2012

¿Qué desencadena la actualización del origen?

La propiedad [UpdateSourceTrigger](#) del enlace determina qué desencadena la actualización del origen.



- Si el valor de [UpdateSourceTrigger](#) es `PropertyChanged`, el valor se actualizará en cuanto cambie la propiedad de destino y viceversa.
- Si el valor de [UpdateSourceTrigger](#) es `LostFocus` (valor por defecto), ese valor sólo se actualizará con el nuevo valor cuando la propiedad de destino pierda el foco.

© JMA 2012

Orígenes de enlaces

- Recursos:
 - Objetos del CLR
 - ObjectDataProvider
 - XmlDataProvider
 - CollectionViewSource
- Propiedad DataContext del contenedor.

© JMA 2012

ObjectDataProvider

- Ajusta y crea un objeto que puede utilizar como origen de enlace.
- Permite pasar parámetros al constructor de la clase.
- Permite ejecutar un método con parámetros del objeto como fuente de datos.
- La propiedad Data expone el objeto de datos subyacente.

```
<Window.Resources>
    <ObjectDataProvider ObjectType="{x:Type local:TemperatureScale}"
        MethodName="ConvertTemp" x:Key="convertTemp">
        <ObjectDataProvider.MethodParameters>
            <system:Double>0</system:Double>
            <local:TempType>Celsius</local:TempType>
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
```

© JMA 2012

XmlDataProvider

- Los datos subyacentes a los que se puede tener acceso mediante el enlace de datos en la aplicación pueden ser cualquier árbol de nodos XML.

- El referenciado de los nodos se realiza mediante consultas Xpath.

```
<XmlDataProvider x:Key="InventoryData" XPath="Inventory/Books">
    <x:XData>
        <Inventory xmlns="">
            ...
        </Inventory>
    </x:XData>
</XmlDataProvider>
<XmlDataProvider x:Key="BookData" Source="data\bookdata.xml"
    XPath="Books"/>
<XmlDataProvider x:Key="BookData" Source="http://MyUrl"
    XPath="Books"/>
```

© JMA 2012

CollectionViewSource

- Representa una vista para agrupar, ordenar, filtrar y navegar por una colección de datos.

- Requiere los xmlns:

```
– xmlns:scm="clr-
    namespace:System.ComponentModel;assembly=WindowsBase"
– xmlns:dat="clr-
    namespace:System.Windows.Data;assembly=PresentationFramework"
<CollectionViewSource Source="{StaticResource places}" x:Key="cvs">
    <CollectionViewSource.SortDescriptions>
        <scm:SortDescription PropertyName="CityName"/>
    </CollectionViewSource.SortDescriptions>
    <CollectionViewSource.GroupDescriptions>
        <dat:PropertyGroupDescription PropertyName="State"/>
    </CollectionViewSource.GroupDescriptions>
</CollectionViewSource>
```

© JMA 2012

Extensiones de enlazado

- Las llaves ({ y }) indican el uso de una extensión de marcado que se aparta del tratamiento general de valores de atributo.
 - **Binding**: proporciona un valor enlazado a datos para una propiedad, utilizando el contexto de datos que se aplica al objeto primario en tiempo de ejecución.
 - **RelativeSource**: proporciona información de origen para un objeto Binding que puede navegar por varias posibles relaciones en el árbol de objetos en tiempo de ejecución.
 - **TemplateBinding**: permite que una plantilla de control utilice valores para propiedades con plantilla procedentes de propiedades definidas por el modelo de objetos de la clase que utilizará la plantilla.
 - **StaticResource**: proporciona un valor para una propiedad sustituyendo el valor de un recurso ya definido.
 - **DynamicResource**: proporciona un valor para una propiedad aplazando ese valor para que sea una referencia a un recurso en tiempo de ejecución.
 - **ColorConvertedBitmap**: admite un escenario de creación de imágenes relativamente avanzado.
 - **ComponentResourceKey** y **ThemeDictionary**: admiten aspectos de la búsqueda de recursos y temas que se empaquetan con controles personalizados.

© JMA 2012

{Binding}

- Origen:
 - Source, RelativeSource
- Ruta de acceso:
 - Path, XPath, ElementName
- Dirección:
 - Mode, BindsDirectlyToSource
- Conversiones:
 - StringFormat, Converter, ConverterParameter, ConverterCulture
- Notificaciones:
 - NotifyOnSourceUpdated, NotifyOnTargetUpdated, NotifyOnValidationError, UpdateSourceTrigger
- Validaciones:
 - ValidationRules, ValidatesOnExceptions, ValidatesOnDataErrors

© JMA 2012

Nivel de vinculación

- Contextos de datos: DataContext
- Vinculación de propiedades
 - Propiedades de dependencia
- Vinculación de colecciones
 - ItemsControl
 - ItemsSource, DisplayMemberPath, SelectedValuePath
 - SelectedValue, SelectedItem

© JMA 2012

Convertidores

```
/// <summary>
/// Convierte un numero a cadena
/// </summary>
public class StringToNullConverter : IValueConverter {
    #region Miembros de IValueConverter

        public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture) {
            return value;
        }

        public object ConvertBack(object value, Type targetType, object
parameter, System.Globalization.CultureInfo culture) {
            if (value == null)
                return null;
            else if(value.ToString().EstaVacio())
                return null;
            return value;
        }
    }
```

© JMA 2012

Reglas de validación

```
public class RequiredValidator : ValidationRule {
    public override ValidationResult Validate(object value,
        System.Globalization.CultureInfo cultureInfo) {
        if (value == null || String.IsNullOrEmpty(value.ToString()))
            return new ValidationResult(false, "Requerido");
        else
            return ValidationResult.ValidResult;
    }
}
<TextBox.Text>
<Binding Path="Nombre" UpdateSourceTrigger="PropertyChanged" >
    <Binding.ValidationRules>
        <c:RequiredValidator />
    </Binding.ValidationRules>
</Binding>
</TextBox.Text>
```

© JMA 2012

Validación

- **IDataErrorInfo**
 - Proporciona la funcionalidad para facilitar información de error personalizada a la que puede enlazar una interfaz de usuario.
 - **Error** : Obtiene un mensaje de error que indica lo que le pasa a este objeto.
 - **Item[String]**: Obtiene el mensaje de error correspondiente a la propiedad con el nombre especificado.

© JMA 2012

Validación

- **INotifyDataErrorInfo (v 4.5)**
 - Permite que las clases de entidad de datos implementar reglas de validación personalizadas y exponer los resultados de validación de forma asíncrona, admitiendo objetos de error personalizados, varios errores por propiedad, errores entre propiedades (que afectan a varias propiedades, se pueden asociar con una o todas las propiedades afectadas) y errores de nivel de entidad (que afectan a varias propiedades o afectar a la entidad completa sin que afecte a una propiedad determinada).
 - **HasErrors:** Obtiene un valor que indica si la entidad tiene errores de validación.
 - **GetErrors(String):** Obtiene los errores de validación para una propiedad específica o para toda la entidad.
 - **ErrorsChanged:** Evento que se produce cuando los errores de validación han cambiado para una propiedad o para la toda la entidad.

© JMA 2012

Plantilla de Validación

```
<ControlTemplate x:Key="ValidationTemplate">
    <Grid DataContext="{Binding}" Name="CntrlContainer">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <AdornedElementPlaceholder Name="AdornedCntrl" />
        <Image x:Name="iconoErr" Stretch="None"
            Source="/Demos;component/Imagenes/16x16/exclamation.png"
            ToolTip="{Binding ElementName=AdornedCntrl,
Path=AdornedElement.(Validation.Errors)[0].ErrorContent}" Grid.Column="1" />
    </Grid>
</ControlTemplate>
<Style x:Key="CntrValidado" TargetType="{x:Type Control}">
    <Setter Property="Validation.ErrorTemplate" Value="{StaticResource ValidationTemplate}" />
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="ToolTip" Value="{Binding
Path=(Validation.Errors)[0].ErrorContent, RelativeSource={x:Static RelativeSource.Self}}"/>
            <Setter Property="Background" Value="{StaticResource
FondoErrorValidacion}" />
        </Trigger>
    </Style.Triggers>
</Style>
```

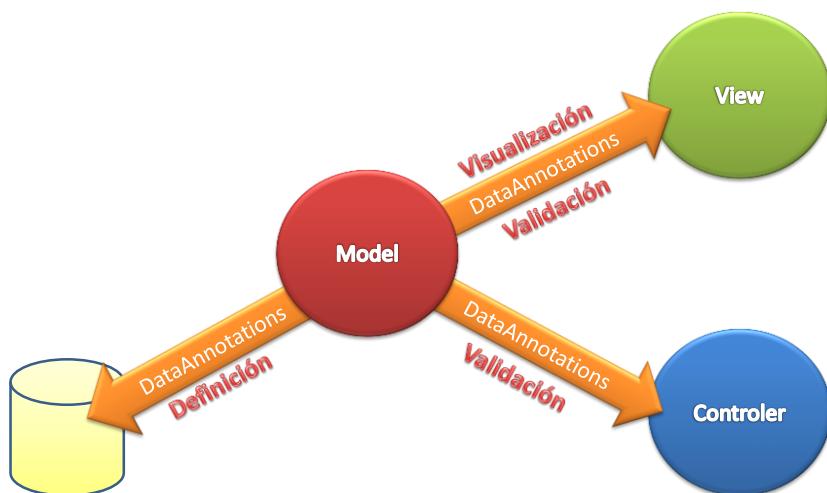
© JMA 2012

DataAnnotations

- System.ComponentModel.DataAnnotations;
- Permiten definir los metadatos de los modelos de datos para su uso por entidades externas.
- Cuenta con decoradores para:
 - Definición del modelo de datos: claves (PK), asociaciones (FK), simultaneidad, ...
 - Interfaz de usuario y localización: Título, descripción, formato, solo lectura, ...
 - Validaciones y errores personalizados: Obligatoriedad, longitudes, formatos, ...

© JMA 2012

Contexto Declarativo



© JMA 2012

Visualización

Decorador	Descripción
DataType	Especifica el nombre de un tipo adicional que debe asociarse a un campo de datos.
Display	Permite especificar las cadenas traducibles de los tipos y miembros de las clases parciales de entidad.
DisplayFormat	Especifica el modo en que los datos dinámicos de ASP.NET muestran y dan formato a los campos de datos.
ScaffoldColumn	Especifica si una clase o columna de datos usa la técnica scaffolding.
ScaffoldTable	Especifica si una clase o tabla de datos usa la técnica scaffolding.
UIHint	Especifica la plantilla o el control de usuario que los datos dinámicos usan para mostrar un campo de datos.
System.Web.Mvc	
HiddenInput	Indica que una propiedad se debería presentar como un elemento input oculto.

© JMA 2012

[Display]

- **Name:** valor que se usa para mostrarlo en la interfaz de usuario (Título, Etiqueta, ...).
- **Description:** valor que se usa para mostrar una descripción en la interfaz de usuario.
- **Prompt:** valor que se usará para establecer la marca de agua para los avisos en la interfaz de usuario.
- **ShortName:** valor que se usa para la etiqueta de columna de la cuadrícula.
- **GroupName:** valor que se usa para agrupar campos en la interfaz de usuario.
- **Order:** número del orden de la columna.

© JMA 2012

Tipos asociados

Asociado	Descripción
DateTime	Representa un instante de tiempo, expresado en forma de fecha y hora del día.
Date	Representa un valor de fecha.
Time	Representa un valor de hora.
Duration	Representa una cantidad de tiempo continua durante la que existe un objeto.
PhoneNumber	Representa un valor de número de teléfono.
Currency	Representa un valor de divisa.
Text	Representa texto que se muestra.
Html	Representa un archivo HTML.
MultilineText	Representa texto multilínea.
EmailAddress	Representa una dirección de correo electrónico.
Password	Representa un valor de contraseña.
Url	Representa un valor de dirección URL.
ImageUrl	Representa una URL en una imagen.
CreditCard	Representa un número de tarjeta de crédito.
PostalCode	Representa un código postal.
Upload	Representa el tipo de datos de la carga de archivos.

© JMA 2012

Validación

Decorador	Descripción
Required	Especifica que un campo de datos necesita un valor.
DataType	Especifica el nombre de un tipo adicional que debe asociarse a un campo de datos. Decoradores especializados: CreditCard, EmailAddress, EnumDataType, FileExtensions, Phone, Url
Compare	Compara dos propiedades.
Range	Especifica las restricciones de intervalo numérico para el valor de un campo de datos.
StringLength	Especifica la longitud mínima y máxima de caracteres que se permiten en un campo de datos. Decoradores especializados: MinLength, MaxLength
RegularExpression	Especifica que un valor de campo de datos debe coincidir con la expresión regular especificada.
CustomValidation	Especifica un método de validación personalizado que se utiliza para validar una propiedad o una instancia de clase.

© JMA 2012

Validación manual de objetos

- En `System.ComponentModel.DataAnnotations`, la clase estática `Validator` ofrece métodos que permiten realizar las comprobaciones de forma directa sobre objetos o propiedades concretas.

```
IEnumerable<ValidationResult> getValidationErrors(object obj) {  
    var validationResults = new List<ValidationResult>();  
    var context = new ValidationContext(obj, null, null);  
    Validator.TryValidateObject(obj,  
        context,  
        validationResults,  
        true);  
    return validationResults;  
}
```

© JMA 2012

IValidableObject

- Obliga a implementar un único método, llamado `Validate()`, que determina si el objeto especificado es válido.
- Será invocado automáticamente por `TryValidateObject()` siempre que no encuentre errores al comprobar las restricciones especificadas mediante anotaciones.
- Devolverá una lista de objetos `ValidationResult` con los resultados de las comprobaciones.
- En las clases prescriptivas (EF) se aplica con una partial class.
- Interfaces relacionados: `IDataErrorInfo`, `INotifyDataErrorInfo`

© JMA 2012

Anotar clases ya existentes

- Se requiere una clase auxiliar con las propiedades a decorar decoradas.
- Declarativa: [MetadataType(typeof(tipo))]
 - Se aplica con una partial class (en el mismo ensamblado que la clase a anotar).
- Imperativa: System.ComponentModel.TypeDescriptor y AssociatedMetadataTypeTypeDescriptionProvider:

```
var descriptionProvider = new
    AssociatedMetadataTypeTypeDescriptionProvider(
        typeof(Friend), typeof(FriendMetadata));
TypeDescriptor.AddProviderTransparent(
    descriptionProvider, typeof(Friend));
```

© JMA 2012

Repositorio

- Un repositorio separa la lógica empresarial de las interacciones con la base de datos subyacente y centra el acceso a datos en un área, lo que facilita su creación y mantenimiento.
- El repositorio pertenecen a la capa de infraestructura y devuelve los objetos del modelo de dominio.
- Deberían implementar el patrón de doble herencia.
- Forma parte de los Domain Driven Design patterns: Domain Entity, Value-Object, Aggregates, Repository, Unit of Work, Specification, Dependency Injection, Inversion of Control (IoC).

© JMA 2012

MultiBinding y PriorityBinding

- Asociar una colección de objetos Binding a una sola propiedad.

```
<TextBlock>
    <TextBlock.Text>
        <MultiBinding StringFormat="{1}, {0}">
            <Binding Path="FirstName"></Binding>
            <Binding Path="LastName"></Binding>
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>
```
- Asociar el primer enlace de la colección que genera un valor correctamente.

```
<TextBlock Background="Honeydew" Width="100" HorizontalAlignment="Center">
    <TextBlock.Text>
        <PriorityBinding FallbackValue="defaultvalue">
            <Binding Path="SlowestDP" IsAsync="True"/>
            <Binding Path="SlowerDP" IsAsync="True"/>
            <Binding Path="FastDP" />
        </PriorityBinding>
    </TextBlock.Text>
</TextBlock>
```

© JMA 2012

PLANTILLAS DE DATOS

© JMA 2012

Plantillas de datos

- El modelo de plantillas de datos de WPF proporciona gran flexibilidad para definir la presentación de los datos.
- Los controles WPF tienen funcionalidades integradas para admitir la personalización de la presentación de los datos.
- Para mostrar un objeto utiliza el método `ToString()` si no dispone de una plantilla de datos.
- Las plantillas se definen con la etiqueta `DataTemplate` y reciben como `DataContext` el objeto a dar formato.

```
<DataTemplate>
    <StackPanel>
        <TextBlock Text="{Binding Path=TaskName}" />
        <TextBlock Text="{Binding Path=Description}" />
        <TextBlock Text="{Binding Path=Priority}" />
    </StackPanel>
</DataTemplate>
```

© JMA 2012

Ámbito de las plantillas

- Local:


```
<ListBox ItemsSource="{Binding Source={StaticResource myList}}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            ...
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```
- Recurso:


```
<Window.Resources>
    ...
    <DataTemplate x:Key="myTaskTemplate">
        ...
    </DataTemplate>
    ...
</Window.Resources>
...
<ListBox ... ItemTemplate="{StaticResource myTaskTemplate}" />
```

© JMA 2012

La propiedad DataType

- La propiedad **DataType** permite asociar una plantilla a un determinado tipo de datos (clase).
- Si se ha establecido el **DataType** no es obligatorio asignar la **x:Key** cuando se definen en el área de recursos, pero en la misma área no puede haber dos plantillas diferentes asociadas al mismo **DataType**.
- En caso de no indicarse explícitamente mediante la **x:Key**, WPF recorre ascendenteamente el árbol de recursos buscando la primera plantilla que satisfaga el tipo.
- Especialmente indicado para colecciones y elementos multi-tipos.

```
<DataTemplate DataType="{x:Type local:Task}">
    <StackPanel>
        ...
    </StackPanel>
</DataTemplate>
```

© JMA 2012

DataTriggers

- Un desencadenador de datos establece propiedades o inicia acciones, como una animación, cuando cambia un valor de propiedad vinculada.

```
<DataTemplate x:Key="myTaskTemplate">
    ...
    <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding Path=TaskType}" Value="Home">
            <Setter TargetName="border" Property="BorderBrush" Value="Yellow"/>
            <Setter TargetName="border" Property="Background" Value="White" />
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>
```

- Se puede disparar por la concurrencia de varios valores:

```
<MultiDataTrigger>
    <MultiDataTrigger.Conditions>
        <Condition Binding="{Binding Path=Name}" Value="Portland" />
        <Condition Binding="{Binding Path=State}" Value="OR" />
    </MultiDataTrigger.Conditions>
    <Setter Property="Background" Value="Cyan" />
</MultiDataTrigger>
```

© JMA 2012

ESTILOS, PLANTILLAS Y CREACIÓN DE CONTROLES

© JMA 2012

Diccionarios de recursos

- El diccionario contiene recursos de WPF utilizados por los componentes y otros elementos de una aplicación WPF.
- Se crean como ficheros XAML sin código subyacente.

```
<ResourceDictionary  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:local="clr-namespace:MVVM" >
```
- Esta compuesto de recursos que debes estar identificados por x:Key o x>Type.
- No puede haber dos recursos con el mismo x:Key.
- No puede haber dos recursos con el mismo x>Type si no tienen asociado un x:Key dado que el tipo actúa de clave en estos casos.
- Todos los recursos reutilizables deberían pertenecer a un diccionario, de tal forma que los recursos se pueden compartir entre las aplicaciones y también quedan aislados de un modo más cómodo para su localización.

© JMA 2012

Diccionarios de recursos

- Los ficheros con diccionarios de recursos deben combinarse con recursos locales del elementos donde quieren utilizarse.
- Un elemento tiene disponibles todos los recursos de su contenedor.
- El app.xaml contiene los recursos globales para toda la aplicación.

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Theme/MiTema.xaml" />
            <ResourceDictionary Source="myresourcedictionary.xaml"/>
            <ResourceDictionary Source="myresourcedictionary2.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

© JMA 2012

Estilos

- Un Style es un conjunto de valores por defecto de las propiedades de cualquier elemento que derive de FrameworkElement o FrameworkContentElement.
- Los valores locales de las propiedades priman sobre los valores establecidos en el Style.
- El Style se puede aplicar a múltiples elementos por lo que se unifica el diseño y facilita la mantenibilidad manteniendo la coherencia estética de la aplicación.
- El estilo se establece con las propiedades Style y las propiedades con sufijo Style.

```
<Style TargetType="{x:Type TextBlock}">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="14"/>
</Style>
```

© JMA 2012

Elementos del estilo

- **TargetType:** tipo para el que está previsto este estilo, sin x:Key se aplica a todos los objetos de este tipo.
- **BasedOn:** estilo definido que es la base del estilo actual.
- **Resources:** colección de recursos que se pueden usar en el ámbito del estilo.
- **Setters:** colección de objetos Setter y EventSetter.
- **Triggers:** colección de objetos TriggerBase que aplican valores de propiedad basados en condiciones especificadas.

© JMA 2012

Setter

- Representa un establecedor que aplica un valor de propiedad.
 - **Property:** establece la propiedad a la que se va a aplicar el valor de la propiedad Value.
 - **Value:** establece el valor que se va aplicar a la propiedad especificada por Setter.
 - **TargetName:** establece el nombre del objeto al que va destinado el Setter cuando se define en un Trigger de plantilla.
- ```
<Setter Property="Background"
 Value="{DynamicResource WindowBackgroundBrush}" />
```

© JMA 2012

## Desencadenadores

- Un desencadenador establece propiedades o inicia acciones, como una animación, cuando cambia un valor de propiedad o cuando se genera un evento.
- Style , ControlTemplate y DataTemplate tienen una propiedad Triggers que puede contener un conjunto de desencadenadores.
- Hay varios tipos de desencadenadores:
  - Desencadenadores de propiedades
  - Desencadenadores de eventos
  - Desencadenadores de enlazados a datos

© JMA 2012

## Desencadenadores de propiedades

- Un objeto Trigger que establece valores de propiedad o inicia acciones según el valor de una propiedad se denomina desencadenador de propiedad.

```
<Style.Triggers>
 <Trigger Property="IsSelected" Value="True">
 <Setter Property="Opacity" Value="1.0" />
 <Setter Property="MaxHeight" Value="75" />
 </Trigger>
 ...
</Style.Triggers>
```

© JMA 2012

## Desencadenadores de eventos

- Un desencadenador EventTrigger inicia un conjunto de acciones sobre un guion gráfico (Storyboard) en función de la aparición de un evento.
- RoutedEvent: establece el objeto RoutedEvent que activará este desencadenador.
- SourceName: establece el nombre del objeto con el evento que activa este desencadenador. Sólo los desencadenadores de elemento o de plantilla usan esto.

```
<EventTrigger RoutedEvent="Mouse.MouseEnter">
 <EventTrigger.Actions>
 <BeginStoryboard>
 <Storyboard>
 <DoubleAnimation Duration="0:0:0.2" Storyboard.TargetProperty="MaxHeight" To="90" />
 </Storyboard>
 </BeginStoryboard>
 </EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Mouse.MouseLeave">
 <EventTrigger.Actions>
 <BeginStoryboard>
 <Storyboard>
 <DoubleAnimation Duration="0:0:1" Storyboard.TargetProperty="MaxHeight" />
 </Storyboard>
 </BeginStoryboard>
 </EventTrigger.Actions>
</EventTrigger>
```

© JMA 2012

## Combinación condiciones

- MultiTrigger permite establecer valores de propiedad según varias condiciones.
 

```
<MultiTrigger>
 <MultiTrigger.Conditions>
 <Condition Property="HasItems" Value="false" />
 <Condition Property="Width" Value="Auto" />
 </MultiTrigger.Conditions>
 <Setter Property="MinWidth" Value="120"/>
</MultiTrigger>
```
- MultiDataTrigger se utilizan cuando la propiedad de la condición está enlazada a datos.

© JMA 2012

## Plantillas de controles

- Una plantilla ControlTemplate especifica la estructura y el comportamiento visuales de un control.
- Puede personalizar la apariencia de un control proporcionándole una nueva plantilla ControlTemplate.
- Cuando se crea una plantilla ControlTemplate, se reemplaza la apariencia de un control existente sin cambiar su funcionalidad.
- Las plantillas ControlTemplate se crean en XAML, por lo que se puede cambiar la apariencia de un control sin escribir código.

© JMA 2012

## Diseño de la plantilla

- Una plantilla ControlTemplate es la combinación de objetos FrameworkElement para compilar un único control.
- Una plantilla ControlTemplate debe tener un solo objeto FrameworkElement como su elemento raíz, que normalmente contiene otros objetos FrameworkElement.
- La combinación de objetos constituye la estructura visual del control.
- La extensión de marcado TemplateBinding enlaza una propiedad de un elemento que se encuentra en la plantilla ControlTemplate a una propiedad pública definida por el control.
- El control ContentPresenter que muestra el contenido de cualquier tipo de objeto.
- Dispone de la Triggers para asociar desencadenadores.

© JMA 2012

# Estados visuales

- Un comportamiento visual describe la apariencia del control cuando se encuentra en un estado determinado.
- Para especificar la apariencia de un control cuando se encuentra en un estado determinado se emplean objetos VisualState.
- Un objeto VisualState contiene un objeto Storyboard que cambia la apariencia de los elementos que se encuentran en la plantilla ControlTemplate.
- No es necesario escribir código para que esto suceda, ya que la lógica del control cambia de estado mediante VisualStateManager.
- Cuando el control entra en el estado especificado por la propiedad VisualState.Name, se inicia Storyboard.
- Cuando el control sale del estado, Storyboard se detiene.

© JMA 2012

## VisualState

```
<VisualStateManager.VisualStateGroups>
 <VisualStateGroup Name="CommonStates">
 <VisualState Name="Normal" />
 <VisualState Name="MouseOver">
 <Storyboard>
 <ColorAnimation Storyboard.TargetName="BorderBrush"
 Storyboard.TargetProperty="Color" To="Red" />
 </Storyboard>
 </VisualState>
 <VisualState Name="Pressed">
 <Storyboard>
 <ColorAnimation Storyboard.TargetName="BorderBrush"
 Storyboard.TargetProperty="Color" To="Transparent"/>
 </Storyboard>
 </VisualState>
 </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

© JMA 2012

# Contrato de control

- Un control que usa una plantilla ControlTemplate para especificar su estructura visual (mediante objetos FrameworkElement) y su comportamiento visual (mediante objetos VisualState) emplea el modelo de control de elementos.
- Los elementos que un autor de plantillas ControlTemplate debe conocer se comunican mediante el contrato de control.
- Si se entienden los elementos de un contrato de control, es posible personalizar la apariencia de cualquier control que use el modelo de control de elementos.

```
[TemplatePartAttribute(Name = "PART_EditableTextBox", Type = typeof(TextBox))]
[TemplatePartAttribute(Name = "PART_Popup", Type = typeof(Popup))]
public class ComboBox : ItemsControl

<ControlTemplate TargetType="ComboBox">
...
<TextBox x:Name="PART_EditableTextBox" ... />
...
<Popup x:Name="PART_Popup" ...
</ControlTemplate>
```

© JMA 2012

# Temas

- Los temas de Windows Presentation Foundation (WPF) se definen mediante el mecanismo de estilos y plantillas que Windows Presentation Foundation (WPF) expone para personalizar los efectos visuales de cualquier elemento.
- Los recursos de tema de Windows Presentation Foundation (WPF) están almacenados en diccionarios de recursos incrustados.
- Estos diccionarios de recursos deben incrustarse en un ensamblado firmado y pueden incrustarse en el mismo ensamblado que el propio código o en un ensamblado paralelo.
- En el caso de PresentationFramework.dll, el ensamblado que contiene los controles de Windows Presentation Foundation (WPF), los recursos de tema están en una serie de ensamblados paralelos.

© JMA 2012

# Temas

- Combinar para una aplicación.  

```
<ResourceDictionary.MergedDictionaries>
 <!--
 Windows Vista theme themes/Aero.NormalColor.xaml Aero
 Windows Classic theme themes/Classic.xaml Classic
 Default Windows XP theme themes/Luna.NormalColor.xaml LunaBlue
 Olive green Windows XP theme themes/Luna.Homestead.xaml LunaOliveGreen
 Silver Windows XP theme themes/Luna.Metallic.xaml LunaSilver
 Windows XP Media Center Edition theme themes/Royale.NormalColor.xaml Royale
 Zune Windows XP theme themes/Zune.NormalColor.xaml Zune
 Expression Blend theme themes/BlendDictionary.xaml Blend
 -->
 <ResourceDictionary Source="/PresentationFramework.Aero, Version=4.0.0.0,
 Culture=neutral, PublicKeyToken=31bf3856ad364e35,
 ProcessorArchitecture=MSIL;component/themes/aero.normalcolor.xaml" />
 <ResourceDictionary Source="Theme/MiTema.xaml" />
</ResourceDictionary.MergedDictionaries>
```
- Para acceder a los recurso de un tema en el ResourceDictionary :  

```
xmlns:Themes="clr-
namespace:Microsoft.Windows.Themes;assembly=PresentationFramework.Aero"
```

© JMA 2012

# Creación de controles

- Alternativas a la escritura de un nuevo control
  - Contenido enriquecido.
  - Estilos.
  - Plantillas de datos.
  - Plantillas de control.
  - Desencadenadores.
- Modelos para la creación de controles
  - Derivar de UserControl
  - Derivar de Control
  - Derivar de FrameworkElement

© JMA 2012

# Principios de diseño

- Utilizar propiedades de dependencia
- Utilizar eventos enrutados
- Utilizar el Binding
- Diseñar para diseñadores (Visual Studio)
- Definir y usar recursos compartidos
- Definir la estructura y el comportamiento visuales en un ControlTemplate
- Proporcionar el contrato de control

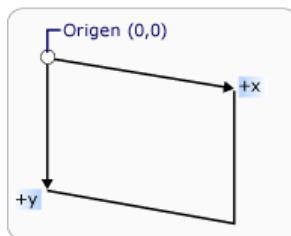
© JMA 2012

# Gráficos

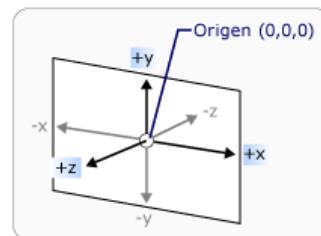
- Gráficos 2D
- Gráficos 3D → ViewPort3D

## Coordinadas

2 Dimensiones



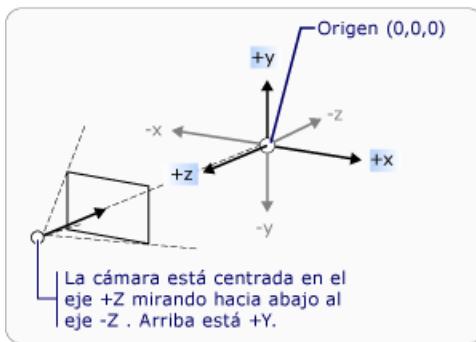
3 Dimensiones



© JMA 2012

## Gráficos 3D

Al crear una escena 3-D, es importante recordar que, en realidad, se está creando una representación 2-D de los objetos 3-D. Dado que una escena 3-D tiene un aspecto diferente dependiendo del punto de vista del espectador, debe especificar ese punto de vista. La clase [Camera](#) permite especificar este punto de vista para una escena 3-D.



© JMA 2012

## Animaciones

**Una animación es una ilusión que se crea mediante el cambio rápido entre una serie de imágenes, cada una de las cuales es ligeramente diferente de la anterior.**

WPF incluye un sistema de control de tiempo eficaz que se expone a través del código administrado y del Extensible Application Markup Language (XAML) que se integra perfectamente en el marco de WPF. La animación WPF facilita la animación de controles y otros objetos gráficos.

© JMA 2012

# Transformaciones

**Se desplazan todos los puntos de un modelo u objeto según un valor o modo especificado.**

## En 2-D:

- *RotateTransform.*
- *ScaleTransform.*
- *SkewTransform.*
- *TranslateTransform.*

## En 3-D:

- *RotateTransform3D.*
- *ScaleTransform3D.*
- *TranslateTransform3D.*
- *MatrixTransform3D.*

© JMA 2012

## Entonces para animar los gráficos...

- Es posible animar directamente las propiedades de los elementos primitivos, pero suele ser más fácil animar las transformaciones que cambian la posición o el aspecto de los modelos.
- Para animar un objeto en WPF, se crea una escala de tiempo, se define una animación (que, en realidad, es un cambio de algún valor de propiedad a lo largo del tiempo) y se especifica la propiedad a la que aplicar la animación.

© JMA 2012

# Animaciones From/To/By

- Se considera la animación básica.
- Es una animación que ocurre entre dos valores inicio y final y va incrementando con un valor de incremento el valor de inicio.
- Tiene una propiedad From con la que se especifica el valor inicio y una propiedad To para especificar el valor final.
- En lugar de la propiedad To se puede usar una propiedad By.
- Para comenzar lo primero que tenemos que hacer es inicializar un StoryBoard (conjunto de imágenes o animaciones en secuencia) para ello utilizamos BeginStoryboard, este lo podemos usar tanto con un Trigger o un EventTrigger.

© JMA 2012

## Animaciones From/To/By

- Los EventTrigger definen tres propiedades fundamentales:
- La propiedad **SourceName** de tipo string que se refiere al nombre del elemento (que se le asocia al elemento con la propiedad Name o x:Name) sobre el que se trabaja.
- La propiedad **RoutedEvent** contiene el nombre del evento que provocará el “desencadenado” de las acciones definidas en el EventTrigger.
- La propiedad **Actions** es la propiedad que define el conjunto de acciones a desencadenar cuando ocurra el evento especificado como condición del EventTrigger.
- A diferencia de los Trigger y DataTrigger, los EventTrigger no tienen concepto de terminación.
- La propiedad TargetProperty indica que propiedad del objeto se va a controlar.

© JMA 2012

# DOCUMENTOS

© JMA 2012

## Introducción

- Windows Presentation Foundation (WPF) proporciona un conjunto versátil de componentes que permiten a los programadores generar aplicaciones con características de documento avanzadas y una experiencia de lectura mejorada.
- Además de las mejoras en las funciones y en la calidad, Windows Presentation Foundation (WPF) proporciona servicios de administración simplificados para el empaquetado, la seguridad y el almacenamiento de los documentos.
- Tipos de documentos
  - Los documentos fijos están diseñados para las aplicaciones que requieren una presentación "what you see is what you get" (WYSIWYG) precisa, independiente del hardware de pantalla o de impresión utilizado.
  - Los documentos dinámicos están diseñados para optimizar su presentación y legibilidad y son óptimos para su uso cuando la facilidad de lectura constituye el principal escenario de consumo del documento.

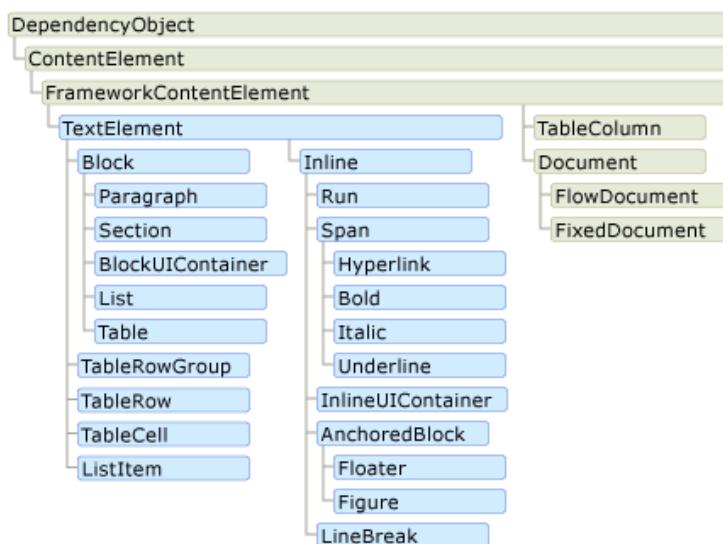
© JMA 2012

# Controles de documentos

- Control de documentos fijos: DocumentViewer
  - DocumentViewer proporciona una interfaz de usuario intuitiva que ofrece compatibilidad integrada para las operaciones comunes, tales como las operaciones de salida impresa, copia en el portapapeles, aplicación de zoom o búsqueda de texto. DocumentViewer se ha diseñado para mostrar el contenido en modo de sólo lectura; la edición o modificación de contenido no está disponible y no se admite.
- Controles de documentos dinámicos
  - FlowDocumentReader dispone de características que permiten al usuario elegir dinámicamente distintos modos de visualización, incluido el modo de visualización de una sola página (una página a la vez), dos páginas a la vez (formato de lectura de libro) y desplazamiento continuo (sin límite).
  - FlowDocumentPageViewer muestra el contenido en el modo de visualización de una sola página.
  - FlowDocumentScrollView muestra el contenido en modo de desplazamiento continuo.

© JMA 2012

# Elementos de documentos



© JMA 2012

## FlowDocument: XAML

```
<FlowDocumentReader xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
 <FlowDocument>
 <Paragraph>
 <Bold>Some bold text in the paragraph.</Bold>
 Some text that is not bold.
 </Paragraph>
 <List>
 <ListItem>
 <Paragraph>ListItem 1</Paragraph>
 </ListItem>
 <ListItem>
 <Paragraph>ListItem 2</Paragraph>
 </ListItem>
 <ListItem>
 <Paragraph>ListItem 3</Paragraph>
 </ListItem>
 </List>
 </FlowDocument>
</FlowDocumentReader>
```

© JMA 2012

## FlowDocument: C#

```
Paragraph myParagraph1 = new Paragraph(new Run("Paragraph 1"));
Paragraph myParagraph2 = new Paragraph(new Run("Paragraph 2"));
Paragraph myParagraph3 = new Paragraph(new Run("Paragraph 3"));
```

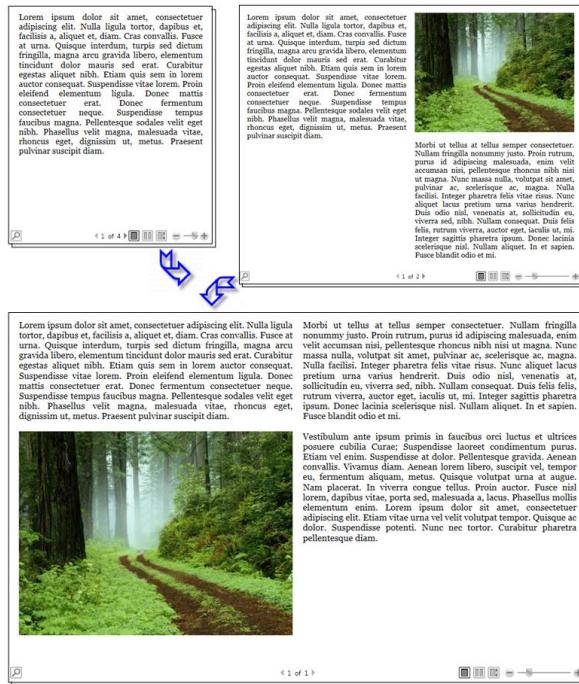
```
Section mySection = new Section();
mySection.Background = Brushes.Red;
```

```
mySection.Blocks.Add(myParagraph1);
mySection.Blocks.Add(myParagraph2);
mySection.Blocks.Add(myParagraph3);
```

```
FlowDocument myFlowDocument = new FlowDocument();
myFlowDocument.Blocks.Add(mySection);
```

```
this.Content = myFlowDocument;
```

© JMA 2012



© JMA 2012

## XAML Dinámico

```

FlowDocument doc = new FlowDocument();
doc.Padding = new Thickness(60, 100, 30, 30);
// Set PageHeight and PageWidth to "Auto".
doc.PageHeight = Double.NaN;
doc.PageWidth = Double.NaN;
// Specify minimum page sizes.
doc.MinPageWidth = 680.0;
doc.MinPageHeight = 480.0;
//Specify maximum page sizes.
doc.MaxPageWidth = 1024.0;
doc.MaxPageHeight = 768.0;

string sDoc = null;
if (t.Documentacion != null) {
 sDoc = @"<Section xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
Style='{StaticResource Detalle_Section}'>";
 sDoc+= (t.Documentacion.toXPS() + "</Section>";
doc.Blocks.Add((System.Windows.Markup.XamlReader.Parse(sDoc) as Section));
}
Visor.Document = doc;

```

© JMA 2012

# LENGUAJE C#

© JMA 2012

## Introducción

- C# ha sido diseñado específicamente para la plataforma .NET
- Lenguaje orientado a objetos y componentes
- Heredero del C++ y Java, sintaxis común mejorada y ampliada
- Algunas de las principales características de este lenguaje incluyen clases, interfaces, delegados, boxing y unboxing, espacios de nombres, propiedades, indexadores, eventos, sobrecarga de operadores, versionado, atributos, código inseguro, y la creación de documentación en formato XML. No son necesarios archivos de cabecera ni archivos IDL (Interface Definition Language).

© JMA 2012

# Introducción

- Sencillez
  - Código autocontenido
  - Tamaño de tipos básicos fijo
  - Operador acceso no mutable (., ::)
  - Sin macros ni herencia múltiple
- Modernidad
  - Amplio conjunto de tipos básicos
  - Estructuras inmediatas (foreach)
  - Atributos o metadatos.
  - Excepciones
- Orientación a objetos
  - Encapsulación: public, private, protected, internal
  - Herencia: Sin herencia múltiple. Suplida por interfaces
  - Polimorfismo: Usando herencia o interfaces.
  - Por defecto métodos sellados. Modificador virtual
  - POO pura. Sin métodos o variables globales
- Orientación a componentes
  - Propiedades y Eventos
- Paradigma funcional y lenguajes dinámicos
  - Inferencia de tipos
  - Expresiones Lambda
  - Tipos anónimos, dinámicos,

© JMA 2012

# Sintaxis

- Sensible a mayúsculas y minúsculas.
- Marca de fin de instrucción: ; (punto y coma): Instrucción en varias líneas sin partir palabras. Pueden ir varias instrucciones en una línea.
- Comentarios: // (hasta fin de línea) o /\* ... \*/ (bloque de comentario)
- Literales:
  - Booleanos: true y false
  - Nulo: null
  - Numéricos: Decimal, Hexadecimal: 0x, Binario: 0b, Exponencial: E
    - Separadores de dígitos ([ver. 7](#)): 0b0001\_0000, 100\_000\_000\_000
  - Caracteres: '...' (Apostrofe)
  - Cadenas: "..." (Comillas) o @..." (cadena textual)
    - Interpolación ([ver. 6](#)): \$"...{expresion}..."
- Delimitadores de bloques: {...}
- Etiquetas → Nombre\_etiqueta:

© JMA 2012

## Secuencia de escape

Secuencia de escape	Nombre del carácter
\u...	Unicote
\x...	Hexadecimal
'	Comilla simple
"	Comilla doble
\	Barra invertida
\0	Null
\a	Alerta
\b	Retroceso
\f	Avance de página
\n	Nueva línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical

© JMA 2012

## Un nombre de elemento o identificadores

- No debe comenzar por un dígito.
- Puede contener caracteres alfabéticos (incluidos acentuados, la ñ y caracteres Unicode no imprimibles), dígitos y signos de subrayado.
- No se pueden utilizar los caracteres de puntuación y símbolos utilizados en el lenguaje.
- No puede superar los 16.383 caracteres de longitud.
- No puede coincidir con ninguna de las palabras clave reservadas si no están precedido por @ (Nombres de escape).
- En la elección debe tenerse en cuenta que influye en los ensamblados reutilizados en otros lenguajes.

© JMA 2012

# Documentador

- Comentarios XML, comienzan por la marca ///
- Preceden inmediatamente al elemento a comentar.

Etiquetas recomendadas	Finalidad
<summary>	Describe un miembro de un tipo
<c>	Establecer un tipo de fuente de código para un texto
<code>	Establecer una o más líneas de código fuente o indicar el final de un programa
<example>	Indicar un ejemplo
<exception>	Identifica las excepciones que pueden iniciar un método
<include>	Incluye XML procedente de un archivo externo
<list>	Crear una lista o una tabla
<para>	Permite agregar un párrafo al texto
<param>	Describe un parámetro para un método o constructor

© JMA 2012

# Documentador

Etiquetas recomendadas	Finalidad
<paramref>	Identifica una palabra como nombre de parámetro
<permission>	Documenta la accesibilidad de seguridad de un miembro
<returns>	Describe el valor devuelto de un método
<see>	Especifica un vínculo
<seealso>	Genera una entrada de tipo Vea también
<remarks>	Describe un tipo
<value>	Describe una propiedad
<typeparam>	Describe un parámetro de tipo genérico
<typeparamref>	Identifica una palabra como nombre de parámetro de tipo

- Sandcastle - Documentation Compiler for Managed Class Libraries
- <http://sandcastle.codeplex.com/>

© JMA 2012

# Directivas al compilador

- Compilación condicional  

```
#if <condición1>
 <código1>
elif <condición2>
 <código2>
else
 <códigoElse>
endif
```
- Definición de constantes  

```
#define <nombrelidentificador>
#define <nombrelidentificador>
Constantes predefinidas: DEBUG y TRACE
En VS: Proyectos → Propiedades → Propiedades de configuración
 → Generar → Constantes de compilación condicional
```

© JMA 2012

# Directivas al compilador

- Generación de diagnósticos  

```
#warning <Mensaje de Aviso>
#error <Mensaje de Error>
```
- Supresión temporal de avisos  

```
pragma warning disable <Número de Error>
 // código ...
pragma warning restore <Número de Error>
```
- Cambios en la numeración de líneas  

```
#line <número> "<nombreFichero>"
```
- Regiones de código  

```
#region "Descripción de la Región"
 // código ...
#endregion
```

© JMA 2012

# Código fuente

- Estructura del código fuente:
  1. Instrucciones de importación, si corresponde
  2. Instrucciones de Espacio de nombres, si corresponde
  3. Instrucciones class, struct, interface, delegate y enum, si corresponde
- Procedimiento Principal
  - static void Main() {...}
  - static void Main(string[] args) {...}
  - static int Main() {...}
  - static int Main(string[] args) {...}

© JMA 2012

## TIPOS, VARIABLES, CONSTANTES, OPERADORES, INSTRUCCIONES

© JMA 2012

## Tipos valor

Tipo en C#	Estructura de tipo CLR	Descripción	Almacenamiento nominal	Intervalo de valores
sbyte	System.SByte	Bytes con signo	8 bytes	-128 a 127
byte	System.Byte	Bytes sin signo	8 bytes	0 a 255
short	System.Int16	Enteros cortos con signo	16 bytes	-32.768 a 32.767
ushort	System.UInt16	Enteros cortos sin signo	16 bytes	0 a 65.535
int	System.Int32	Enteros normales	32 bytes	-2.147.483.648 a 2.147.483.647
uint	System.UInt32	Enteros normales sin signo	32 bytes	0 a 4.294.967.295
long	System.Int64	Enteros largos	64 bytes	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
ulong	System.UInt64	Enteros largos sin signo	64 bytes	0 a 18.446.744.073.709.551.615

© JMA 2012

## Tipos valor

Tipo en C#	Estructura de tipo CLR	Descripción	Almacenamiento nominal	Intervalo de valores
float	System.Single	Reales punto flotante con precisión simple	32 bytes	-3.4028235E+38 a -1,401298E-45 para valores negativos; 1,401298E-45 a 3,4028235E+38 para valores positivos.
double	System.Double	Reales punto flotante con precisión doble	64 bytes	-4,94065645841246544E-324 a 4,94065645841246544E-324
decimal	System.Decimal	Reales de 28 dígitos decimales de precisión	128 bytes	35 con 28 posiciones a la derecha del signo decimal; (+/-1E-28)
bool	System.Boolean	Valores lógicos	32 bytes	true o false
char	System.Char	Caracteres Unicode	16 bytes	0 a 65535 (sin signo) ['\u0000', '\uFFFF']
string	System.String	Cadena de longitud variable	En función a la plataforma	De 0 a 2.000 millones de caracteres Unicode aprox.
object	System.Object	Cualquier objeto	4 bytes	Cualquier objeto

© JMA 2012

# Sufijos de constantes numéricas

Sufijo	Tipo del literal entero
ninguno	Primero de: <b>int, uint, long, ulong</b>
<b>L ó l</b>	Primero de: <b>long, ulong</b>
<b>U ó u</b>	Primero de: <b>int, uint</b>
<b>UL, UI, uL, ul, LU, Lu, IU ó lu</b>	<b>ulong</b>

Sufijo	Tipo del literal real
<b>F ó f</b>	<b>float</b>
<b>ninguno, D ó d</b>	<b>double</b>
<b>M ó m</b>	<b>decimal</b>

© JMA 2012

## Tipos valor como referencias

- Boxing y Unboxing (Empaquetado y desempaquetado)
  - Los tipos valor se empaquetan en sus correspondientes clases.
  - Permite a los tipos valor comportarse como objetos.
  - Aportan funcionalidad adicional al tipo valor
  - Mantienen la eficiencia de los tipos valor.
- Tipos valor que aceptan NULL
  - Los tipos valor se transforman en sus correspondientes clases Nullable.
  - Se crean utilizando el sufijo modificador de tipo ?
  - Tiene dos propiedades públicas de sólo lectura: HasValue, del tipo bool, y Value, del tipo subyacente del tipo que acepta valores NULL.

© JMA 2012

## Tipos de referencia

Tipo de clase	Descripción
System.Object	Clase base definitiva de todos los demás tipos.
System.String	Tipo de cadena del lenguaje C#.
System.ValueType	Clase base de todos los tipos de valor.
System.Enum	Clase base de todos los tipos enum.
System.Array	Clase base de todos los tipos de matriz.
System.Delegate	Clase base de todos los tipos delegados.
System.Exception	Clase base de todos los tipos de excepción.

© JMA 2012

## Conversiones

- **Implícitas**
  - Valor: de un tipo otro de mayor rango de valores.
  - Referencias: de un tipo derivado a cualquiera de sus tipos superiores en la jerarquía de herencia y tipo que implementa un interfaz a un tipo de la interfaz implementada.
- **Explícitas**
  - Empleando funciones o métodos de conversión (System.Convert).
  - Mediante la sobrecarga de operadores.
  - (*<TipoResultante>*)*<Referencia>* ó  
*<Referencia> as <TipoResultante>*
    - Siempre y cuando el tipo declarado de la referencia sea un tipo superior en la jerarquía de herencia del tipo resultante o el tipo declarado de la referencia sea de un tipo interfaz que implemente el tipo resultante. Requiere comprobación de tipos (operador is).

© JMA 2012

# Expresiones y Operadores

Operador	Descripción
<code>&lt;objeto&gt;.&lt;miembro&gt;</code>	Accede a miembros de los objetos o estructuras.
<code>&lt;nombre&gt;[&lt;índice&gt;]</code>	Acceso a los elementos de las matrices e indizadores.
<code>[&lt;Metadatos&gt;]</code>	Marca un bloque de atributos.
<code>&lt; ... &gt;</code>	Marca un bloque de parámetros tipo en los genéricos.
<code>(&lt;expresión&gt;)</code>	Prioriza la evaluación de la expresión.
<code>&lt;nombre&gt;(...)</code>	Invocación de métodos y delegados.
<code>new &lt;tipo&gt;</code>	Se utiliza para crear objetos e invocar constructores
<code>new { ... }</code>	Inicializador de objeto anónimo.
<code>typeof(&lt;tipo&gt;)</code>	Obtiene el objeto System.Type para un tipo.
<code>default(&lt;tipo&gt;)</code>	Obtiene el valor predeterminado del tipo.
<code>checked(&lt;expresión&gt;)</code>	Evaluúa la expresión en contexto comprobado
<code>unchecked(&lt;expresión&gt;)</code>	Evaluúa la expresión en contexto no comprobado

© JMA 2012

# Expresiones y Operadores

Operador	Descripción
<code>&lt;expr1&gt; + &lt;expr2&gt;</code>	Suma operandos aritméticos. Concatenación de cadenas. Combinación de delegados.
<code>&lt;expr1&gt; - &lt;expr2&gt;</code>	Resta operandos aritméticos. Elimina la asociación de delegados.
<code>&lt;expr1&gt; * &lt;expr2&gt;</code>	Multiplica operandos aritméticos.
<code>&lt;expr1&gt; / &lt;expr2&gt;</code>	Divide operandos aritméticos. El resultado será real o entero en función de los operandos.
<code>&lt;expr1&gt; % &lt;expr2&gt;</code>	Modulo (resto) de la división entera.
<code>++&lt;variable&gt;</code>	Preincremento: incrementa en 1 el valor de la variable antes de consultar su valor.
<code>&lt;variable&gt;++</code>	Postincremento: incrementa en 1 el valor de la variable después de consultar su valor.
<code>- &lt;variable&gt;</code>	Predecremento: decrementa en 1 el valor de la variable antes de consultar su valor.
<code>&lt;variable&gt;- -</code>	Postdecremento: decrementa en 1 el valor de la variable después de consultar su valor.

© JMA 2012

# Expresiones y Operadores

Operador	Descripción
<code>&lt;expr1&gt; &amp; &lt;expr2&gt;</code>	AND binario para operadores de tipos enteros y lógico para operadores booleanos.
<code>&lt;expr1&gt;   &lt;expr2&gt;</code>	OR binario para operadores de tipos enteros y lógico para operadores booleanos.
<code>&lt;expr1&gt; ^ &lt;expr2&gt;</code>	XOR binario para operadores de tipos enteros y lógico para operadores booleanos.
<code>&lt;expr&gt; &lt;&lt; &lt;contador&gt;</code>	Desplaza a la izquierda sin desbordamiento los bit de la expresión rellenado con ceros las posiciones libres.
<code>&lt;exprn&gt; &gt;&gt; &lt;contador&gt;</code>	Desplaza a la derecha sin desbordamiento los bit de la expresión rellenado con ceros las posiciones libres.
<code>~&lt;expresión&gt;</code>	Complemento binario. Negación bit a bit.
<code>!&lt;expresión&gt;</code>	Negación lógica.
<code>&lt;expr1&gt; &amp;&amp; &lt;expr2&gt;</code>	AND lógico cortocircuitado, solo evalúa la segunda expresión si la primera es true.
<code>&lt;expr1&gt;    &lt;expr2&gt;</code>	OR lógico cortocircuitado, solo evalúa la segunda expresión si la primera es false.

© JMA 2012

# Expresiones y Operadores

Operador	Descripción
<code>&lt;expr1&gt;&lt;comp&gt; &lt;expr2&gt;</code>	Compara dos expresiones, donde <code>&lt;comp&gt;</code> puede ser: <code>==</code> (igual), <code>!=</code> (distinto), <code>&lt;</code> (menor), <code>&lt;=</code> (menor o igual), <code>&gt;</code> (mayor), <code>&gt;=</code> (mayor o igual).
<code>&lt;expresión&gt; is &lt;tipo&gt;</code>	Comprueba si el tipo en tiempo de ejecución de un objeto es compatible con un tipo dado
<code>&lt;cond&gt;?&lt;exp1&gt;:&lt;exp2&gt;</code>	Condicional: devuelve el primer valor sin la condición es true o el segundo si es false.
<code>&lt;objeto&gt;?.&lt;miembro&gt;</code>	Condicional de NULL ( <b>ver. 6.0</b> ): si el objeto es nulo devuelve nulo, sino el resultado de invocar la miembro, equivale a <code>&lt;objeto&gt; == null ? null : &lt;objeto&gt;.&lt;miembro&gt;</code>
<code>&lt;expr1&gt;??&lt;expr2&gt;</code>	Condicional: devuelve el primer valor si no es nulo o el segundo si el primero es nulo. (Valor de sustitución del null)
<code>&lt;expresión&gt; as &lt;tipo&gt;</code>	Devuelve conversión de referencias entre tipos compatibles o null si no es compatible.
<code>(&lt;tipo&gt;) &lt;expresión&gt;</code>	Realiza conversiones explícitas entre tipos compatibles.

© JMA 2012

# Expresiones y Operadores

Operador	Descripción
<dest> = <expr>	Asigna la expresión al destinatario.
<dest> <opr>= <expr>	Equivale a <destinatario> = <destinatario> <opr> <expr> Donde el <opr> puede ser: * / % + - <<>> & ^
( ... ) => <expresión>	Función anónima (expresión lambda) ( <b>ver. 4.0</b> )
delegate { ... }	Función anónima (método anónimo)
nameof(<miembro>)	( <b>ver. 6.0</b> ) Extrae una cadena con el identificador de la variable, propiedad o campo especificado como argumento (durante la compilación con comprobación de tipos): OnPropertyChanged(nameof(Propiedad));
&<expresión>	Dirección de memoria de su operando. [No administrado].
<expr1>-><expr2>	Acceso a un miembro de la estructura apuntada. El primer operador debe ser un puntero. Similar a (*<expr1>).<expr2>. [No administrado].
sizeof(<tipo>)	Obtiene el tamaño en bytes de un tipo de valor.
stackalloc <tipo>[<tamaño>]	Asigna un bloque de memoria en la pila y devuelve un puntero. [No administrado].

© JMA 2012

# Expresiones y Operadores

Categoría	Prioridad de Operadores
Principal	(Expresión) x.y método(x) tabla[x] x++ x-- new typeof checked unchecked
Unario	+ - ! ~ ++x --x (T)x
Multiplicativo	* / %
Sumatorio	+ -
Desplazamiento	<<>>
Tipos y relacionales	< > <= >= is as
Igualdad	== !=
AND lógico	&
XOR lógico	^
OR lógico	
AND condicional	&&
OR condicional	
Uso combinado de Null	??
Condicional	: ?:
Asignación	= *= /= %= += -= <<= >>= &= ^=  =

© JMA 2012

# Variables y Constantes

- Variables

```
<tipoVariable> <nombreVariable> = <valorInicial>, <nombreVariable> =
<valorInicial>, ...;
– Inicialización por defecto por el CLR:
 • Numéricas = 0, Booleanas = false, Caracteres = '\x0000', Cadenas = "",
 Referencias = null
– Expresión de valor predeterminada:
 <nombreVariable> = default(<tipo>);
– El ámbito de la variable es el bloque donde está definida.
– Tipos especiales de declaraciones de tipo (3.0):
 • var: tipo implícito (ver. 3.0)
 • dynamic: omite la comprobación de tipos en tiempo de compilación (ver. 4.0)
```

- Constantes

```
const <tipo> <nombre> = <valor>, <tipo> <nombre> = <valor>, ...;
readonly <tipo> <nombre> = <valor>, <tipo> <nombre> = <valor>, ...;
```

© JMA 2012

# Matrices y Colecciones

- Matrices (clase Array<>)

```
<tipoDatos>[] <nombreTabla>;
<tipoDatos>[] <nombreTabla> = new <tipoDatos>[<númeroDeElementos>];
<tipoDatos>[] <nombreTabla> = new <tipoDatos>[] {<valores>};
<tipoDatos>[] <nombreTabla> = {<valores>};
<tipo>[, ...] <nombre> = new
<tipoDatos>[<númeroDeElementos>,<númeroDeElementos>, ...];
<tipoDatos>[][]... <nombreTabla>;
– Acceso: Índice 0 a númeroDeElementos-1
 <nombreTabla>[<índice>,<índice>] = <Valor>
 <Variable> = <nombreTabla>[<índice>,<índice>]
```

- Colecciones

```
<tipoColeccion> <nombreColección>;
<tipoColeccion> <nombreColección>= new <tipoColeccion>();
<tipoColección> <nombreColección>= new<tipoColección>(<tamañoInicial>)
<tipoColección> <nombreColección>= new<tipoColección>(<otraColección>);
<tipoColección> <nombreColección>= new <tipoColección> {<valores>} (ver. 3.0)
```

© JMA 2012

# Instrucciones condicionales

```

if (expression)
 statement1
[else
 statement2]

switch (expression) {
 case constant-expression:
 statement
 jump-statement
 [default:
 statement
 jump-statement]
}
jump-statement:
 break;
 goto identifier;
 goto case constant-expression;
 goto default;

```

© JMA 2012

# Tratamientos de excepciones

```

try {
 try-block
} catch (exception-declaration-1) { ...
} catch (exception-declaration-2) { ...
...
} catch (exception-declaration-n) { ...
} finally { finally-block }

```

Filtrado de excepciones (**ver 6.0**):

catch (exception-declaration-1) when (expression)

throw <objetoExcepciónALanzar>;

Expresiones throw (**ver 7.0**):

name = value ?? throw ...;

© JMA 2012

## Instrucciones iterativas

```
while (expression) { ... }

do { ... } while (expression);

for ([initializers]; [expression]; [iterators]) { ... }

foreach (type identifier in expression) { ... }
 – (System.Collections.IEnumerable)

• Alteración del flujo:
 – continue;
 – break;
```

© JMA 2012

## Instrucciones de control

- El desbordamiento aritmético produce una excepción.  
`checked { ... }`  
`checked (expression)`
- Se hace caso omiso del desbordamiento aritmético y el resultado se trunca.  
`unchecked { ... }`  
`unchecked (expression)`
- Impide que el recolector de elementos no utilizados cambie la ubicación de una variable. [No administrado].  
`fixed ( type* ptr = expr ) { ... }`
- Bloqueo de exclusión mutua de un objeto, hace esperar al otro subprocesso hasta que termine el subprocesso.  
`lock(expression) { ... }`

© JMA 2012

# Instrucciones de control

- La instrucción `using` define un ámbito al final del cual el objeto se destruye. Realiza una invocación implícita del método `Dispose` al terminar el ámbito del bloque, por lo tanto debe implementar la interfaz `System.IDisposable`.  
`using (expression | type identifier = initializer) { ... }`
- Salto incondicional (dentro del ámbito de método actual)  
`label-identifier:`  
`goto label-identifier;`
- Devolución del control y de valores:  
`return <valor>;` devuelve el valor de un método  
`yield return <valor>;` genera y devuelve el siguiente valor de una iteración  
`yield break;` indica que se completó la iteración

© JMA 2012

# DEFINICIÓN DE TIPOS

© JMA 2012

# Modificadores de Accesibilidad

Modificador	Tipo	Descripción
public	Publico	Accesible desde cualquier punto sin restricciones.
protected	Protegido	Accesible solamente desde los miembros del tipo donde se encuentra declarado y desde sus herederos.
internal	Interno o amigo	Accesible solamente desde el ensamblado donde se encuentra declarado.
internal protected	Interno y Protegido	Accesible solamente desde los miembros del tipo donde se encuentra declarado y desde sus herederos declarados en su mismo ensamblado.
private	Privado	Accesible solamente desde los miembros del tipo donde se encuentra declarado.

© JMA 2012

# Enumeraciones

```
[attributes] [modifiers] enum
 <nombreEnumeración> [: <integral-type>]
{
 <literal>[= <valor>],
 <literal>[= <valor>],
 ...
}
```

```
<nombreEnumeración> <nombreVariable> =
 <nombreEnumeración>.<literal>
```

© JMA 2012

# Interfaces

```
[attributes] [modifiers] interface INombre
[:<ListaDeInterficies>] {
 Métodos
 Propiedades
 Indizadores
 Eventos
}[:]
```

Los modificadores permitidos son new y los cinco modificadores de acceso.

Por convenio deberían ir prefijados por la letra I (de Interface) para diferenciarlos de la clase base al realizar las implementaciones.

© JMA 2012

# Clases

```
[attributes] [modifiers] class nombre [:base-list] {
 Constructores
 Destructores
 Constantes
 Campos
 Métodos
 Propiedades
 Indizadores
 Operadores
 Eventos
 Delegados
 Clases
 Interfaces
 Estructuras
}[:]
```

Modificadores de Accesibilidad

Los modificadores **new**, **protected** y **private** sólo se permiten en clases anidadas.  
Notación Pascal

© JMA 2012

# Clases

Modificador	Tipo	Descripción
<b>new</b>	Ocultación o sombreado	Ocultar o sombra un elemento heredado de una clase base.
<b>abstract</b>	Abstracta	Marca la clase como abstracta, no es instanciable. Incompatible con sealed
<b>sealed</b>	Sellada	No se puede heredar de esta clase.

- Modificador parte de la clase
  - El modificador partial indica que la clase puede estar repartida en varios archivos.
- Herencia e interfaces:
  - :<ClaseBase>
  - :<ListaDeInterfaz>
  - :<ClaseBase>, <ListaDeInterfaz>

© JMA 2012

# Estructuras

```
[atributes] [modifiers] struct <Nombre> [: <Lista de interfaces>] {
 Constantes
 Atributos
 Métodos
 Propiedades
 Eventos
 Indizadores
 Operadores
 Constructores
 Constructores estáticos
 Tipos anidados
}
```

Los modificadores permitidos son new y los cinco modificadores de acceso.

© JMA 2012

# Estructuras

- Similares a las clases, pueden tener: atributos, métodos, propiedades, ...
- Pero son más rápidas por ser de tipo valor.
- Las estructuras se diferencian de las clases de diferentes maneras:
  - Las estructuras son tipos de valor.
  - Todos los tipos de estructura se heredan implícitamente de la clase System.ValueType.
  - No se puede heredar de una estructura (son implícitamente sealed)
  - La asignación a una variable de un tipo de estructura crea una copia del valor que se asigne.
  - El valor predeterminado de una estructura es el valor producido al establecer todos los campos de tipos de valor en su valor predeterminado, y todos los campos de tipos de referencia en null.
  - Las operaciones boxing y unboxing se utilizan para realizar la conversión entre un tipo struct y un tipo object.
  - El significado de this es diferente para las estructuras (solo ámbito).
  - Las declaraciones de atributos de instancia para una estructura no pueden incluir inicializadores de variable.
  - Los atributos de tipo tabla pueden incluir el modificador fixed para indicar búferes fijos (solo en contextos no seguros).
  - Una estructura no puede declarar un constructor de instancia sin parámetros (solo a partir de la *ver. 6.0*).
  - Una estructura no puede declarar un destructor.

© JMA 2012

# Espacios de nombres

- El espacio de nombres es un ámbito que permite organizar el código y proporciona una forma de crear tipos únicos globalmente exclusivos.

```
namespace CompanyName.TechologyName[.Feature][...][.Design] {
 Otro espacio de nombres
 class, struct, interface, delegate y enum, si corresponde
}
```

- No existe una correspondencia 1 a 1 entre espacios de nombres y ensamblados.
- Las clases de un espacio de nombres se pueden repartir en varios ensamblados.
- Un ensamblado puede contener clases de varios espacios de nombres.

© JMA 2012

# Espacios de nombres

- La directiva using se utiliza para:
  - Crear un alias para un espacio de nombres.
  - Permitir el uso de los tipos de un espacio de nombres sin que sea necesario preceder el nombre del tipo por el nombre de su espacio de nombres.
  - El **calificador de alias de espacios de nombres** :: garantiza que las búsquedas de nombres de tipos no se vean afectadas por la introducción de nuevos tipos y miembros.
  - Los alias extern permiten crear y hacer referencia a diferentes jerarquías de espacios de nombres en diferentes ensamblados (se definen como directivas de compilación)

```
using [alias =]class_or_namespace;
```

- Para utilizar una clase de un espacio de nombres, no basta con la directiva using, es necesario tener referenciado el ensamblado que la contiene.
- Directiva Using Static (**ver. 6.0**)
  - Permite especificar una clase en la que se pueden acceder los métodos estáticos disponibles en el ámbito global, sin prefijos de tipo.

```
using static System.Math;
```

© JMA 2012

# Modificadores de Miembros

Modificador	Tipo	Descripción
static	Estático o de clase	Pertenece al propio tipo (la clase) no a las instancias. Se puede utilizar con campos, métodos, propiedades, operadores, eventos y constructores, pero no con indizadores, destructores o tipos.
virtual	Virtual	Indica que puede reemplazarse en una clase derivada. Incompatible con static, override y abstract.
override	Sobscribe	Reemplaza un método, propiedad, indizador o evento heredado. El elemento en la clase base debe estar marcado como virtual. Incompatible con new, static, virtual y abstract.
abstract	Abstracta	Marca al método, propiedad, indizador o evento como abstracto, no implementado (sin cuerpo). Son implícitamente virtual. Las clases derivadas están obligadas a redefinirlos salvo que sean abstractas. Incompatible con static.
unsafe	Inseguro	Denota un contexto no seguro, es necesario para cualquier operación que involucre a punteros.
extern	Externo	Indica que el método se implementa externamente, no implementado (sin cuerpo). Se usa normalmente con el atributo [DllImport("????.dll")].

© JMA 2012

# Constantes y Atributos

- Constantes

[attributes] [modifiers] const type declarators = value;

- Los modificadores permitidos son new y los cinco modificadores de acceso.

- Atributos o campos

[attributes] [modifiers] <tipoVariable> <nombreVariable> =  
 <valorInicial>, ...;

- Los modificadores permitidos son new, static, readonly, volatile y los cinco modificadores de acceso.
- El modificador volatile indica que un campo puede ser modificado en el programa por el sistema operativo, el hardware o un subprocesso en ejecución de forma simultánea.
- Las asignaciones a los campos readonly sólo pueden tener lugar en la propia declaración o en un constructor de la misma clase.

© JMA 2012

# Procedimientos y funciones

<tipoRetorno> <nombreFunción>(<ListaDeParámetros>) { ... }

**void** <nombreProcedimiento>(<ListaDeParámetros>) { ... }

- Lista de parámetros

- Por valor: <tipo> <Nombre>
- Por referencia: **ref** <tipo> <Nombre>, debe estar instanciado.
- De salida: **out** <tipo> <Nombre>
- Número variable: **params** <tipo>[] <Nombre>, de 0 a n. (Último de la firma)
- Opcionales (ver. 4.0): <tipo> <Nombre> = <ValorPorDefecto> (Últimos de la firma)

- Devolución de valores: **return** [expression];

- Invocación

<nombreFunción>(<const>, <var>, ref <var>, out <var>)

- Argumentos con nombre (ver. 4.0):
  - <nombreFunción>(<Nombre>: <const> o <var>, ...)
- Saltar argumentos opcionales:
  - <nombreFunción>(<const>, <NombreArg>, <var>)

- Firma del método: número, orden y tipos de los parámetros

© JMA 2012

# Métodos

- Declaración en la interfaz:  
`[atributes] [modifiers] <tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>) { ... }`  
`[atributes] [modifiers] void [<interfaz>.]<nombreProcedimiento>(<ListaDeParámetros>) { ... }`  
`[atributes] [modifiers] <tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>);`  
`[atributes] [modifiers] void [<interfaz>.]<nombreProcedimiento>(<ListaDeParámetros>);`
- Los modificadores permitidos son new, static, virtual, sealed, override, abstract, extern y los cinco modificadores de acceso.
- Declaración en la interfaz:  
`[atributes] [new] <tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>);`  
`[atributes] [new] void [<interfaz>.]<nombreProcedimiento>(<ListaDeParámetros>);`
- Métodos parciales (ver. 3.0)
  - Se pueden definir en una parte de una declaración de tipo e implementarse en otra.
  - La implementación es opcional; si ninguna parte implementa el método parcial, la declaración de método parcial y todas sus llamadas se quitan de la declaración de tipo resultante de la combinación de las partes.

```
partial void <nombreProcedimiento>(<ListaDeParámetros>); // Declaración
partial void <nombreProcedimiento>(<ListaDeParámetros>) { ... } // Implementación
```
- Métodos automáticas con cuerpo de expresión (ver. 6.0)  
`<tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>) => expresión;`

© JMA 2012

# Sobrecarga de Operadores

```
public static result-type operator unary-operator (this-type operand)
public static result-type operator binary-operator (this-type operand1,
op-type operand2)
public static result-type operator binary-operator (op-type operand1,
this-type operand2)
public static result-type operator binary-operator (op-type op, int
despla) // << >>
public static implicit operator conv-type-out (conv-type-in operand)
public static explicit operator conv-type-out (conv-type-in operand)
```

- Operadores unarios sobrecargables:  
`+ - ! ~ ++ -- true false`
- Operadores binarios sobrecargables:  
`- + - * / % & | ^ << >> == != > < >= <=`

© JMA 2012

# Propiedades

[attributes] [modifiers] type [<interfaz>.]identifier {< Descriptores de acceso>}

- Los modificadores permitidos son new, static, virtual, sealed, override, abstract, extern y los cinco modificadores de acceso.
- Descriptores de acceso
 

```
[modifiers] get {...}
[modifiers] set { ... value ... }
```

  - Los modificadores permitidos son los cinco modificadores de acceso.
- Declaración en la interfaz:
 

```
[attributes] [new] type identifier { get; set; }
```
- Propiedades auto implementadas (**ver. 3.0**)
 

```
[attributes] [modifiers] type identifier {get; set;}
```
- Propiedades auto implementadas solo lectura (**ver. 6.0**)
 

```
[attributes] [modifiers] type identifier {get; private set;}
```

```
[attributes] [modifiers] type identifier {get; }
```
- Inicialización de las propiedades auto implementadas (**ver. 6.0**)
 

```
[attributes] [modifiers] type identifier {get; set;} = value;
```

© JMA 2012

# Indizadores

[attributes] [modifiers] type <interfaz>.this [formal-index-parameter-list]  
 {<DescriptoresDeAcesso>}

[attributes] [new] type this [formal-index-parameter-list] {get;set;}

- Los modificadores permitidos son new, virtual, sealed, override, abstract, extern y una combinación válida de los cinco modificadores de acceso.
- Descriptores de acceso
 

```
get {...}
set { ... value ... }
```
- Para proporcionar al indizador un nombre que puedan utilizar otros lenguajes para la propiedad indizada predeterminada, se debe utilizar el atributo:
 

```
[System.Runtime.CompilerServices.CSharp.IndexerName("<Nombre>")]
```

  - El indizador tendrá el nombre <Nombre>. Si no se especifica el atributo de nombre, el nombre predeterminado será Item.
- Declaración en la interfaz:
 

```
[attributes] [new] type this [formal-index-parameter-list] {get;set;}
```
- Inicializador del indizador (**ver. 6.0**)
 

```
new Colección() { [índice] = valor, ... }
```

© JMA 2012

# Iteradores

- Métodos que calculan y devuelven una secuencia de valores
- Debe devolver `IEnumerator` o `IEnumerable`
- La sentencia `foreach` se apoya en un patrón de enumeración
- Las interfaces del enumerador son `System.Collections.IEnumerator` y los tipos construidos a partir de `System.Collections.Generic.IEnumerator<T>`.
- Las interfaces enumerables son `System.Collections.IEnumerable` y los tipos construidos a partir de `System.Collections.Generic.IEnumerable<T>`.
- Instrucciones:
  - `yield return <valor>`; genera el siguiente valor de la iteración
  - `yield break`; indica que se completó la iteración

```
public IEnumerator<T> GetEnumerator() {
 for (int i = count - 1; i >= 0; --i) {
 yield return items[i];
 }
}
```

© JMA 2012

# Delegados

- Define un tipo de referencia que se puede utilizar para encapsular un método con una firma específica (similar a un puntero a función con tipo).
   
`[attributes] [modifiers] delegate result-type identifier ([formal-parameters]);`
- Los modificadores permitidos son `new` y los cinco modificadores de acceso.
   
`<TipoDelegado> <Variable> = new
 <TipoDelegado>(<Objeto>.<Método>); // Antes del 2.0
 <TipoDelegado> <Variable> = <Objeto>.<Método>;`
- Donde la firma de `<Objeto>.<Método>` debe coincidir con la declarada en `<TipoDelegado>`.
- Métodos anónimos:
   
`delegate([formal-parameters]) { ... Cuerpo... }`

© JMA 2012

# Eventos

[attributes] [modifiers] event type declarator;  
 [attributes] [modifiers] event type member-name {accessor-declarations};

- Descriptores de acceso
 

```
add { ... value ... }
remove { ... value ... }
```
- Declaración en la interfaz:
 

```
[attributes] [new] event type declarator;
```
- Patrón estándar:
 

```
void <Evento>(object sender, <HerederoDeEventArgs> e)
```
- Conceptos:
  - Cualquier objeto capaz de producir un evento es un remitente de eventos.
  - Cualquier objeto capaz de contener un remitente de eventos, puede ser un consumidor de eventos.
  - Los controladores de eventos son procedimientos llamados cuando se produce un evento correspondiente.

© JMA 2012

# Eventos

Pasos a seguir:

1. En caso de ser necesario, definir una clase que proporcione los datos del evento. Esta clase debe derivar de System.EventArgs, que es la clase base de los datos del evento.
2. En caso de ser necesario, declarar un tipo delegado para el evento (Action<>).
3. Obligatoriamente, definir un miembro de evento público en la clase remitente de eventos.
4. Opcionalmente, incluir en la clase remitente un método protegido que provoque el evento. Este método se debe denominar OnNombreEvento. El método OnNombreEvento provoca el evento invocando a los delegados. Antes de provocar el evento es necesario comprobar que el evento está asociado a controladores de eventos, en caso contrario, si no se encuentra asociado generará una excepción.
 

```
if(<NombreEvento> != null) <NombreEvento>(this, e);
```
5. En los métodos apropiados provocar el evento o invocar, en caso de que este incluido, al método que lo provoca.
6. Crear los controladores de eventos, normalmente en la clase consumidora de eventos aunque no obligatoriamente. Debe tener la misma firma que el delegado del evento.
7. Asociar o registrar, en el consumidor de eventos, el controlador de eventos al evento de una instancia del remitente de eventos. Un controlador de eventos puede estar asociado a varios eventos de varias instancias siempre y cuando tengan la misma firma. Un evento puede tener asociados varios controladores de eventos (se ejecutan en el mismo orden en el que se han asociado).
 

```
<Instancia> <Evento> += new <TipoDelegadoEvento>(<Objeto>.<MétodoControlador>)
```

Para desasociar un evento:

```
<Instancia> <Evento> -= new <TipoDelegadoEvento>(<Objeto>.<MétodoControlador>)
```

© JMA 2012

# Auto referencia y Ámbito

- **this**

- Clases

- Referencia a la instancia de la clase que se está implementando.
- Operador de ámbito que resuelve los conflictos de nombre entre parámetros y atributos.

- Estructuras

- Operador de ámbito que resuelve los conflictos de nombre entre parámetros y atributos.

- **base**

- Clases

- Operador de ámbito que permite el acceso a la parte heredada.
- Solo permite a la clase base inmediata.

© JMA 2012

# Constructores y destructores.

- Constructores:

```
[attributes] [modifiers] <NombreClase>{[formal-parameter-list]} [<Inicializador>] { ... }
[attributes] static <NombreClase>() { ... } // Constructor de clase
– Los constructores de instancia no se heredan.
– Los constructores se pueden sobrecargar.
– Los constructores no se pueden invocar directamente, salvo con el operador new o los inicializadores.
– Si la clase no tiene constructor, se genera automáticamente un constructor predeterminado público sin parámetros y los campos del objeto se inicializan con los valores predeterminados.
– Se puede realizar la inicialización de objetos sin llamadas explícitas a un constructor: (ver. 3.0)
 Clase C = new Clase { Propiedad1=v1, P2=v2 };
– Si se marcan como privados impiden la instanciación desde fuera de la clase.
```

- Destructores:

```
[attributes] ~<NombreClase> () { ... }
– Los destructores no se pueden heredar ni sobrecargar.
– No se puede llamar a los destructores. Se invocan automáticamente.
```

- Inicializador:

```
– : base (argument-list)
– : this (argument-list)
```

- Los modificadores permitidos son extern y los cinco modificadores de acceso.

© JMA 2012

# ELEMENTOS AVANZADOS

© JMA 2012

## Atributos (anotaciones o metadatos)

- Información adicional para el ensamblado, módulo y elementos (y sus miembros)  
[<indicadorElemento>:<nombreAtributo> (<parámetros>)]  
Elemento
  - assembly: Indica que el atributo se aplica al ensamblado en que se compile el código fuente que lo contenga.
  - module: Indica que el atributo se aplica al módulo en que se compile el código fuente que lo contenga.
  - <elementos>: Cualifica al elemento al que precede y no es necesario dar el indicador de elemento.
- Parámetros
  - Parámetros sin nombre: Dependerá de su posición a la hora de determinar a qué parámetro se le está dando cada valor.
  - Parámetros con nombre: Son opcionales y pueden colocarse en cualquier posición en la lista de <parámetros> del atributo (<nombreParámetro>=<valor>).
- El atributo Obsolete se utiliza para marcar tipos y miembros de tipos que ya no se deberían utilizar.
- Clases de atributo condicional

```
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}
[Test]
class C {}
```

© JMA 2012

# Componentes genéricos

- Pueden ser clases, estructuras, interfaces, delegados y métodos.
- La definición se realiza mediante alias de posibles tipos.
- En el momento de su utilización se resuelven los alias con tipos existentes.
- Se pueden restringir los tipos posibles de sustitución de alias.
- Declaraciones de clases genéricas  
`[attributes] [modifiers] class nombre<listaAliasDeTipo> [:base-list] [lista de restricciones de tipo]{  
... Miembros ...  
};`
- Declaraciones de estructuras genéricas  
`[attributes] [modifiers] struct nombre<listaAliasDeTipo> [:ListaDeInterfces] [lista de restricciones de tipo]{  
... Miembros ...  
};`
- Declaraciones de Interfaces genéricas  
`[attributes] [modifiers] interface nombre<listaAliasDeTipo> [:ListaDeInterfces] [lista de restricciones de tipo]{  
... Miembros ...  
};`

© JMA 2012

# Componentes genéricos

- Declaraciones de delegados genéricas  
`[attributes] [modifiers] delegate result-type  
identifier<listaAliasDeTipo> ([formal-parameters]) [lista de restricciones de tipo];`
- Declaraciones de métodos genéricos  
`[attributes] [modifiers] <tipoRetorno>  
<nombreMétodo><listaAliasDeTipo> (<ListaDeParámetros>)  
[lista de restricciones de tipo] { ... }`
- Restricción de tipo:
  - **where** AliasDeTipo: ... restricciones ...
    - Posibles restricciones (definición opcional, separadas por comas):
      - tipoClase, class ó struct (principio de la lista)
      - lista de interfaces
      - Otro alias de la listaAliasDeTipo
  - **new()** (al final de la lista)

© JMA 2012

# Clases estáticas

- Las clases estáticas formalizan el patrón de diseño de la “clase sellada no instanciable”.
- Todos sus miembros deben ser estáticos.
- Son clases de utilidad.
 

```
[attributes] [modifiers] static class nombre [:base-list] {
 ... Miembros estáticos o de clase ...
}
```
- Métodos de extensión (*ver. 3.0*)
 

```
public static <tipoRetorno> <nombreMétodo>(<this <tipoClaseAExtender>
 <ParmConRefAExtender>, <ListaDeParámetros>) { ... }
```

  - Deben estar declarados en clases estáticas (clases con el sufijo Extensions).
  - Extienden las clases existentes con métodos estáticos que puedan invocarse mediante la sintaxis de método de instancia de las clases existentes.
  - Para poder ser usados se debe importar explícitamente (using) su espacio de nombres.

© JMA 2012

# Tipos dinámicos, Anónimos y Expresiones lambda

- Uso de tipo dinámico (*ver. 4.0*)
 

```
dynamic variable;
```

  - Se trata de un tipo estático, pero un objeto de tipo dynamic omite la comprobación de tipos en tiempo de compilación.
  - En la mayoría de los casos, funciona como si tuviera el tipo object.
  - En tiempo de compilación, se supone que un elemento de tipo dynamic admite cualquier operación.
- Tipos anónimos (*ver. 3.0*)
 

```
var x = new { prop1 = val1, prop2 = val2, ... }
```

  - Habilita la creación inmediata de tipos estructurados sin nombre que se pueden agregar a colecciones y a los que se puede tener acceso utilizando var.
- Expresiones lambda (*ver. 3.0*)
 

```
(input parameters) => expression
```

  - Habilita expresiones insertadas con parámetros de entrada que se pueden enlazar a delegados o árboles de expresión.
  - Son funciones anónimas y simplifican su creación.
  - Equivale a: 

```
delegate(input parameters) { return expresión; }
```
  - Los parámetros toman el tipo de la definición en la invocación. Los paréntesis son opcionales cuando el parámetro es único.
  - Si el cuerpo de la función requiere algo más que una expresión, se crea un bloque que requiere un return explícito.

© JMA 2012

# Expresiones de consulta (ver. 3.0)

- Aparecen para dar soporte a Linq
- Palabras clave que especifican cláusulas en una expresión de consulta:
  - Cláusulas from
  - Cláusula where (opcional)
  - Cláusulas de ordenación (opcional)
  - Cláusula join (opcional)
  - Cláusula select o group
  - Cláusula into (opcional)

```
from typeopt identifier in expression
let identifier = expression
where boolean-expression
join typeopt identifier in expression on expression equals expression
join typeopt identifier in expression on expression equals expression into identifier
orderby orderings
expression ordering-ascendingopt descendingopt
select expression
group expression by expression
into identifier query-body
```

- Ej: from ... into x ... from x in ( from ... ) ...

© JMA 2012

# Métodos asincrónicos (ver. 5.0)

- Las palabras clave **async** y **await** permiten crear un método asincrónico casi tan fácilmente como se crea un método sincrónico.

```
async Task<string> MetodoAsync() {
 Task<string> tarea = ...;
 // ...
 string rslt = await tarea;
 // ...
 return rslt;
}
```

© JMA 2012

# Extensiones para el código no seguro

- Marcar:

```
externopt unsafeopt static
unsafeopt externopt static
externopt static unsafeopt
unsafeopt static externopt
static externopt unsafeopt
static unsafeopt externopt
unsafe block
```

- Punteros:

```
Type* p = &var;
```

- La instrucción **fixed**, que se utiliza para “fijar” una variable móvil de manera que su dirección permanece constante en toda la duración de la instrucción:

```
fixed (pointer-type fixed-pointer-declarators) embedded-statement
```

- El operador sizeof devuelve el número de bytes ocupados por una variable de un tipo dado:

```
sizeof (unmanaged-type)
```

- Una declaración de variable local puede incluir un inicializador de asignación de pila que asigne memoria de la pila de llamadas:

```
stackalloc unmanaged-type [expression]
```

© JMA 2012

# Tuplas (ver: 7.0)

- Las tuplas de C# son tipos que permiten agrupar varios valores como una unidad mediante una sintaxis ligera. Entre otras ventajas, incluyen una sintaxis más sencilla, reglas para conversiones en función de un número (denominadas cardinalidad) y tipos de elementos y reglas coherentes para copias, pruebas de igualdad y asignaciones.

```
var unnamed = ("one", "two"); // (string, string)
var named = (id: 1, name: "one"); // (int, string)
```

- El struct ValueTuple incluye campos denominados Item1, Item2, Item3, etc., similares a las propiedades definidas en los tipos Tuple existentes. Estos nombres son los únicos que se pueden usar en tuplas sin nombre.:

```
var rslt = unnamed.Item2; // two
var rslt = named.name; // one
```

- Las tuplas pueden ser tipos de retorno o de parámetros:

```
(int, string) método((string, string) arg) { ... }
```

© JMA 2012

## Deconstrucción y descartes (ver: 7.0)

- La deconstrucción proporciona una manera ligera de recuperar varios valores para asignarlos a varias variables o argumentos.  

```
var (id, name) = named;
(int id, string name) = named;
```
- Aunque la deconstrucción apareció junto con las tuplas también pueden usarse con clases, estructuras o interfaces.
- Los descartes son variables de solo escritura cuyos valores se decide omitir, suelen designarse mediante un carácter de guion bajo ("\_") en una asignación.  

```
var (_ name) = named;
```
- Los tipos que no son de tupla no ofrecen compatibilidad integrada con los descartes. Para ello debe implementar de uno o varios métodos Deconstruct de instancia o implementar uno o varios métodos de extensión Deconstruct que devuelvan los valores que le interesen..

© JMA 2012

## Detección de patrones (ver: 7.0)

- La coincidencia de patrones es una característica que permite implementar la distribución de métodos en propiedades distintas al tipo de un objeto.
- Permite crear variables al vuelo una vez detectado el tipo evitando casting o asignaciones explícitas.
- La coincidencia de patrones admite expresiones is y switch. Cada una de ellas habilita la inspección de un objeto y sus propiedades para determinar si el objeto cumple el patrón buscado.

```
if (shape is Square s)
 return s.Side * s.Side;
else if (shape is Circle c)
 return c.Radius * c.Radius * Math.PI;
switch (shape) {
 case Square s: return s.Side * s.Side;
 case Circle c: return c.Radius * c.Radius * Math.PI;
```

- Con la palabra clave when se puede especificar reglas adicionales para el patrón.

```
switch (shape) {
 case Square s when s.Side == 0:
 case Circle c when c.Radius == 0:
 return 0;
 case Square s: return s.Side * s.Side;
```

© JMA 2012

# COLECCIONES

© JMA 2012

## Introducción

- A menudo, los datos similares pueden controlarse de forma más eficaz si se almacenan y manipulan como si fuesen una colección. Puede usar la clase System.Array pero presentan serias limitaciones al tener que estar dimensionados desde el principio. Las colecciones permiten agregar, quitar y modificar elementos individuales o intervalos de elementos de forma dinámica.
- Hay dos tipos principales de colecciones: las colecciones genéricas y las colecciones no genéricas. Las colecciones genéricas se agregaron en la versión 2.0 de .NET Framework y son colecciones con seguridad de tipos en tiempo de compilación. Debido a esto, las colecciones genéricas normalmente ofrecen un mejor rendimiento. Las colecciones genéricas aceptan un parámetro de tipo cuando se construyen y no requieren conversiones con el tipo Object al agregar o quitar elementos de la colección.
- A partir de .NET Framework 4, las colecciones del espacio de nombres System.Collections.Concurrent proporcionan operaciones eficaces y seguras para subprocesos con el fin de obtener acceso a los elementos de la colección desde varios subprocesos. Las clases de colección inmutables en el espacio de nombres System.Collections.Immutable (NuGet package) son intrínsecamente seguras para los subprocesos, ya que las operaciones se realizan en una copia de la colección original, mientras que la colección original no se puede modificar.

© JMA 2012

# Selección de una colección

- Para almacenar elementos como pares clave/valor para una consulta rápida por clave:
  - Hashtable, Dictionary<TKey, TValue>, ConcurrentDictionary<TKey, TValue>,  
ReadOnlyDictionary<TKey, TValue>, ImmutableDictionary<TKey, TValue>
- Con acceso a elementos por índice:
  - Array, ArrayList, List<T>, ImmutableList<T>, ImmutableList
- Con estructura de cola FIFO (el primero en entrar es el primero en salir):
  - Queue, Queue<T>, ConcurrentQueue<T>, ImmutableList<T>
- Con estructura de pila LIFO (el último en entrar es el primero en salir):
  - Stack, Stack<T>, ConcurrentStack<T>, ImmutableList<T>
- Para acceso a elementos de forma secuencial:
  - LinkedList<T>
- Con notificaciones cuando se quitan o se agregan elementos a la colección.  
(implementa INotifyPropertyChanged y INotifyCollectionChanged):
  - ObservableCollection<T>
- Una colección ordenada:
  - SortedList<TKey, TValue>, ImmutableList<TKey, TValue>, ImmutableList<T>
- Un conjunto (sin repetición):
  - HashSet<T>, SortedSet<T>, ImmutableList<T>, ImmutableList<T>

© JMA 2012

## List<T>

- Propiedades
  - Count
  - Item [int]
- Métodos
  - Add(T elemento)
  - AddRange(Collection c)
  - Clear()
  - Contains(T elemento)
  - Find( <>predicado>>)
  - Insert(int, T)
  - Remove(T)
  - RemoveAt(int)
  - Sort()
  - TrueForAll(<>predicado>>)

© JMA 2012

## List<T>

```
List<int> lista = new List<int>();
lista.Add(10);
lista.Add(20);
lista.Add(30);
foreach (var valor in lista)
 Console.WriteLine(valor);
var i = lista[1];
lista.RemoveAt(1);
bool pares = lista.TrueForAll(e => e%2 == 0);
```

© JMA 2012

Language-Integrated Query

# INTRODUCCIÓN A LINQ

© JMA 2012

## ¿Qué es LINQ?

- Mecanismo uniforme y extensible para consultar fuentes de datos de diferentes tipos: las expresiones de consulta.
- Sintaxis basada en nuevas palabras reservadas contextuales.
- Semántica “enchufable”: los lenguajes no definen la semántica de las nuevas palabras reservadas, sino únicamente un conjunto de reglas para reescribir esas expresiones como cascadas de llamadas a métodos.

© JMA 2012

## Características LINQ

- Sintaxis familiar para escribir consultas (“Parecida” a SQL).
- Sintaxis unificada independientemente de la fuente de datos.
- Esfuerzo enfocado en el negocio y no en el acceso a datos.
- Comprobación en tiempo de compilación de errores de sintaxis y seguridad de tipos.
- Compatibilidad con IntelliSense.
- Eficaces funcionalidades de filtrado, ordenación y agrupación.
- Simplicidad de código y orientado a objetos.
- Transaccional.
- Basado en Lambda Cálculo, Tipado fuerte, Ejecución retrasada (deferred), Inferencia de tipos, Tipos anónimos, Métodos extensores y Inicialización de objetos

© JMA 2012

## Lista de providers

- LINQ to SQL
- LINQ to XML
- LINQ to Objects
- LINQ to DataSets
- LINQ To Active Directory
- LINQ To Amazon
- LINQ To mysql, Oracle, PostgreSql
- LINQ To JSON
- LINQ To Sharepoint
- LINQ To JavaScript
- LINQ To Excel

© JMA 2012

## Expresiones de consulta

```
var delMadrid =
 from f in DatosFutbol.Futbolistas
 where f.CodigoClub == "RMA"
 select new { f.Nombre, f.Edad };
```



```
var delMadrid =
 DatosFutbol.Futbolistas
 .Where(f => f.CodigoClub == "RMA")
 .Select(f => new { f.Nombre, f.Edad }) ;
```

© JMA 2012

# Expresión de Consulta

```

from id in source
{ from id in source |
 join id in source on expr equals expr [into id] |
 let id = expr |
 where condition |
 orderby ordering, ordering, ... }
 select expr | group expr by key
[into id query]

```

Empieza con *from*

Cero o más *from, join, let, where, o orderby*

Termina con *select o group by*

Continuación *into* opcional

© JMA 2012

## Expresiones de consulta

- Fuentes de consultas
  - Los datos provienen de cierta fuente, que implementa `IEnumerable<T>`.
- Operadores de consulta estándar
  - No todos los operadores tienen un reflejo en la sintaxis de los lenguajes.
  - El patrón LINQ.

© JMA 2012

# Operadores

Restricción	Where
Proyección	Select, SelectMany
Ordenación	OrderBy, ThenBy
Agrupación	GroupBy
Encuentros	Join, GroupJoin
Cuantificadores	Any, All
Partición	Take, Skip, TakeWhile, SkipWhile
Conjuntuales	Distinct, Union, Intersect, Except
Un elemento	First, Last, Single, ElementAt
Agregados	Count, Sum, Min, Max, Average
Conversión	ToArray,ToList, ToDictionary
Conversión de elementos	OfType<T>, Cast<T>

© JMA 2012

# Operadores de Consulta

Expresión de consulta de Linq	Where(), Select(), SelectMany(), OrderBy(), ThenBy(), OrderByDescending(), ThenByDescending(), GroupBy(), Join(), GroupJoin()
Partición	Take(), Skip(), TakeWhile(), SkipWhile()
Conjunto	Distinct(), Union(), Intersect(), Except()
Conversión	ToArray(),ToList(), ToDictionary(), ToLookup(), AsEnumerable(), Cast<T>(), OfType<T>()
Generación	Range(), Repeat<T>(), Empty<T>(), Concat(), Reverse()
Cuantificación	Any(), All(), Contains(), SequenceEqual()
Elementos	First(), Last(), Single(), ElementAt(), DefaultIfEmpty(). {método}OrDefault()
Agregados	Count(), LongCount(), Max(), Min(), Sum(), Average(), Aggregate()

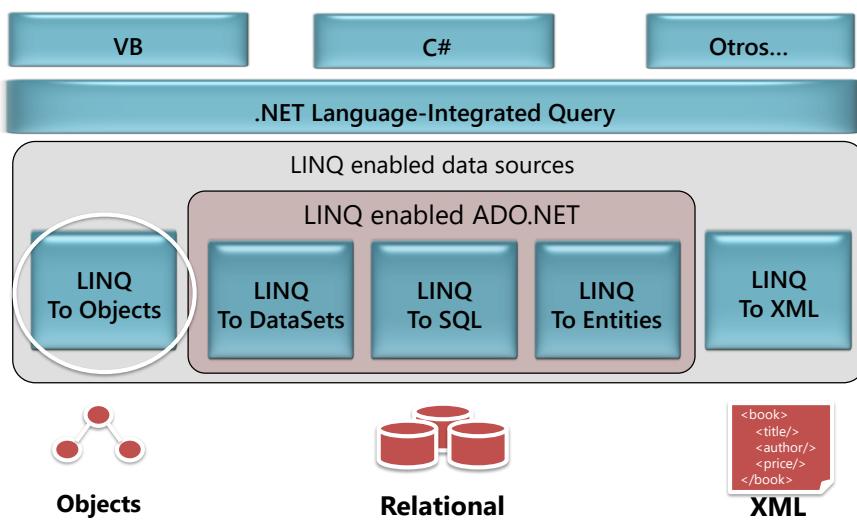
© JMA 2012

# Expresiones de consulta

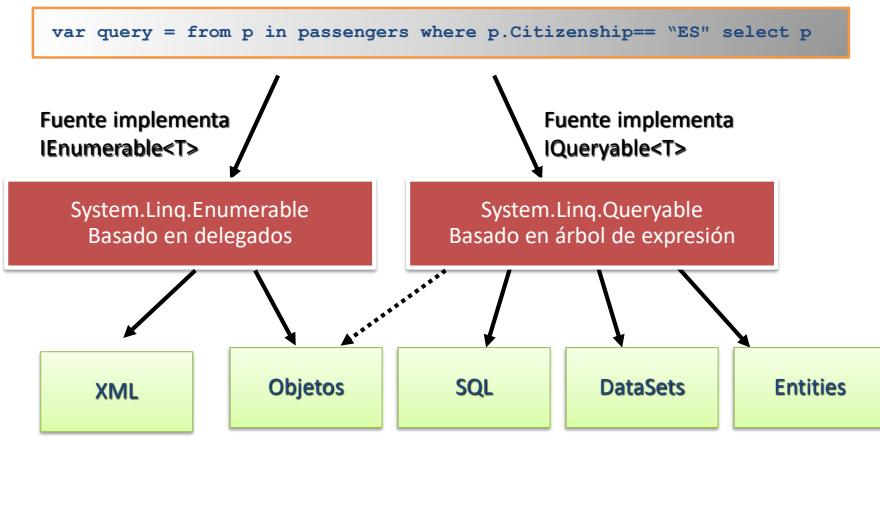
- Composicionales, jerárquicas
  - Anidamiento arbitrario.
  - Posibilidad de aplicar operadores adicionales.
- Declarativas y no imperativas
  - Diga qué usted desea obtener, no cómo.
  - El cómo va por el proveedor.
- Ejecución diferida
  - Las consultas se ejecutan solo a medida que sus resultados se solicitan.

© JMA 2012

## Language INtegrated Query (LINQ)



# Arquitectura de LINQ



© JMA 2012

## Dos clases de proveedores

	Basados en <code>IEnumerable&lt;T&gt;</code>	Basados en <code>IQueryable&lt;T&gt;</code>
Interfaz	<code>IEnumerable&lt;T&gt;</code>	<code>IQueryable&lt;T&gt;</code>
Ejecución	Local, en memoria	Usualmente remota
Implementación	Iteradores	Ánálisis de árboles de expresiones
Proveedores	LINQ to Objects LINQ to XML LINQ to DataSet	LINQ to SQL LINQ to Entities
Mis ejemplos	LINQ to Pipes LoggingLINQ	LINQ to TFS

© JMA 2012

## Extendiendo LINQ

- Habilite sus API existentes para LINQ
  - Específicamente para consultas en memoria.
  - Cree métodos extensores que devuelvan un objeto `IEnumerable<T>`.
- Desarrolle su propio proveedor de consultas
  - Implemente `IQueryable<T>`.
  - Analice árboles de expresiones y traduzca nodos a código o a un lenguaje de consultas diferente.

© JMA 2012

## Recomendaciones

- Analice cuándo y cómo sus consultas se ejecutan
  - Momento de ejecución.
  - Ejecución local vs. remota.
  - Lugar/capa de ejecución real.
- Mantenga las consultas dentro de ensamblados
  - No pase expresiones de consulta entre capas.

© JMA 2012

## Recomendaciones (2)

- Cuidado con los tipos anónimos!
  - Planifique de antemano qué tipos son importantes.
  - No abuse de las proyecciones.
- Aprenda:
  - A escribir consultas con y sin la sintaxis.
  - Las nuevas características de C# 3.0
  - Los detalles de la traducción de la sintaxis en llamadas a operadores y cómo funcionan éstos.

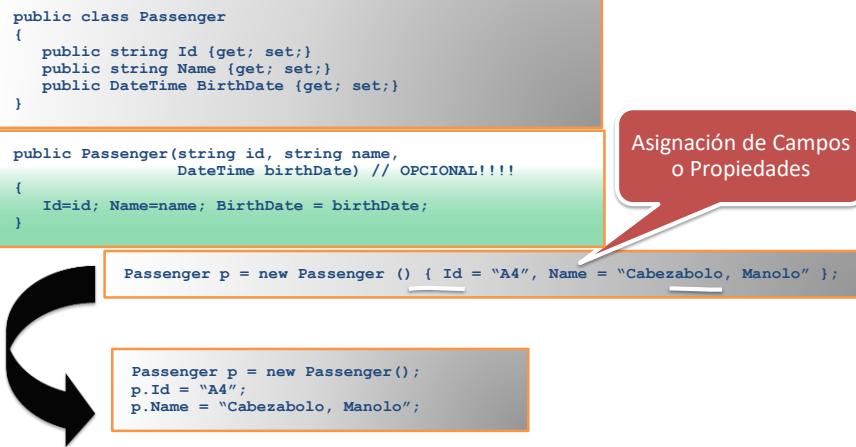
© JMA 2012

## Nuevas características

- Inicializadores de objetos
- Inferencia de tipos
- Tipos anónimos
- Métodos extensores
- Expresiones lambda
- Árboles de expresión
- LINQ!!!

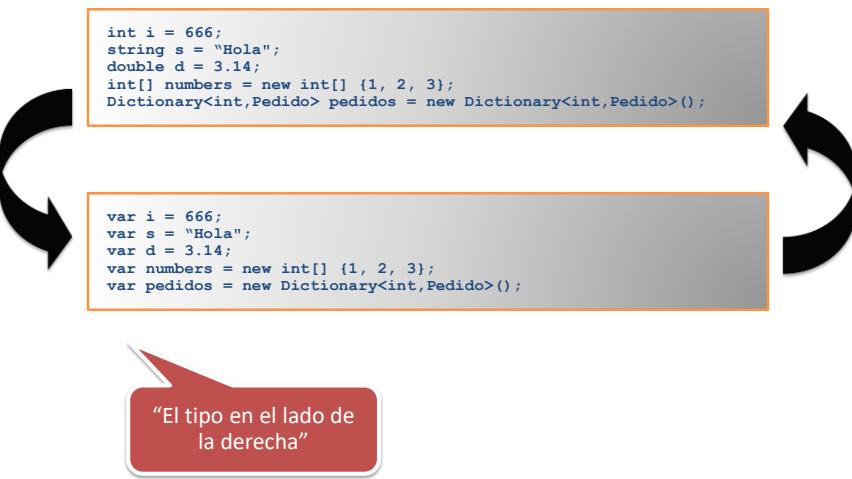
© JMA 2012

# Inicializadores de Objetos



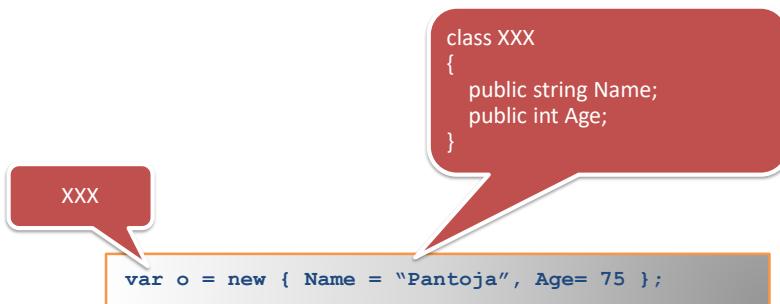
© JMA 2012

# Inferencia de Tipos



© JMA 2012

# Tipos Anónimos



© JMA 2012

## Métodos Extensos

```

namespace MisCosas
{
 public static class Extensiones
 {
 public static string Concatenar(this IEnumerable<string> strings,
 string separador) {...}
 }
}

```

Método extensor

```
using MisCosas;
```

Incluir extensiones en el ámbito

```
string[] nombres = new string[] { "Edu", "Juan", "Manolo" };
string s = nombres.Concatenar(", ");
```

IntelliSense!

obj.Foo(x, y)  
 ↓  
 XXX.Foo(obj, x, y)

© JMA 2012

# Expresiones Lambda

```
public delegate bool Predicate<T>(T obj);

public class List<T>
{
 public List<T> FindAll(Predicate<T> test) {
 List<T> result = new List<T>();
 foreach (T item in this)
 if (test(item)) result.Add(item);
 return result;
 }
 ...
}
```

Delegado genérico

Tipo genérico

© JMA 2012

# Expresiones Lambda

```
public class MiClase
{
 public static void Main() {
 List<Cliente> clientes = ObtenerListaClientes();
 List<Cliente> locales =
 clientes.FindAll(
 new Predicate<Cliente>(CiudadIgualCoruna)
);
 }

 static bool CiudadIgualCoruna(Cliente c) {
 return c.Ciudad == "A Coruña";
 }
}
```

© JMA 2012

# Expresiones Lambda

```
public class MiClase
{
 public static void Main() {
 List<Cliente> clientes = ObtenerListaClientes ();
 List<Cliente> locales =
 clientes.FindAll(
 delegate(Cliente c) { return c.Ciudad == "A Coruña"; }
);
 }
}
```

Delegado  
Anónimo

© JMA 2012

# Expresiones Lambda

```
public class MiClase
{
 public static void Main() {
 List<Cliente> clientes = ObtenerListaClientes ();
 List<Cliente> locales =
 clientes.FindAll(
 (Clientes c) => {return c.Ciudad == "A Coruña"; }
);
 }
}
```

Expresión  
Lambda

© JMA 2012

# Expresiones Lambda

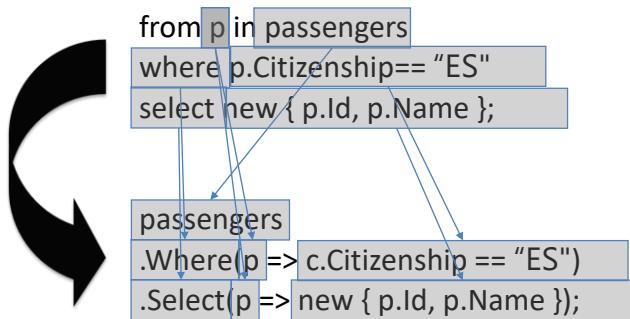
```
public class MiClase
{
 public static void Main() {
 List<Cliente> clientes = ObtenerListaClientes ();
 List<Cliente> locales =
 clientes.FindAll(c => c.Ciudad == "A Coruña");
 }
}
```

Expresión  
Lambda

© JMA 2012

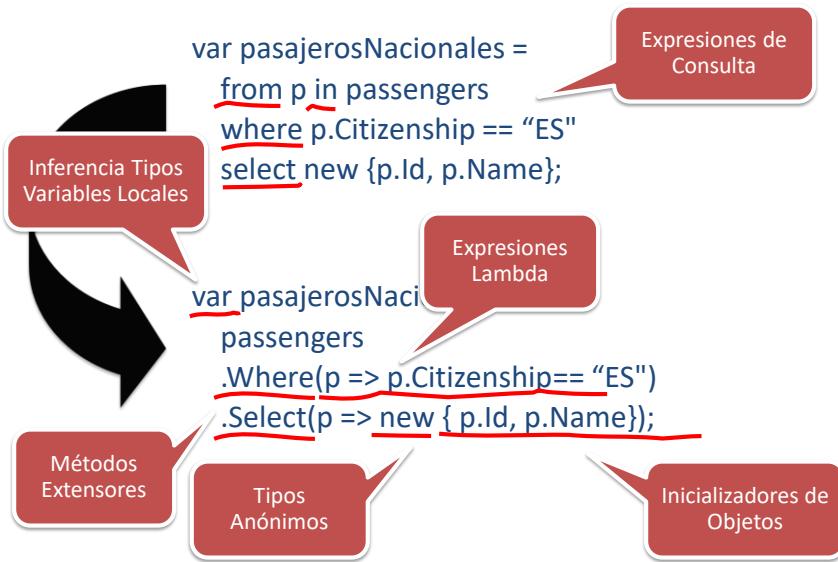
# Introduciendo LINQ

- Todos estos nuevos aspectos se trasladan a métodos extensores sobre colecciones:
  - Pueden transformarse en árboles de expresiones



© JMA 2012

# Introduciendo LINQ



© JMA 2012