Escuela de Sistemas y Tecnologías

Transparencias de **ANALISTA DE SISTEMAS**
Edición 2010 – Materia: **POO en C#**

TEMA: Programación Orientada a Objetos

Consideraciones

- Estas transparencias no tienen el objetivo de suplir las clases.
- Por tanto, serán complementadas con ejemplos, códigos, profundizaciones y comentarios por parte del docente.
- El orden de dictado de estos temas está sujeto a la consideración del docente.
- El material aquí expuesto es (en su mayoría) un resumen del repartido teórico denominado "**OOP en c#**" por lo que se exhorta su lectura como complemento (indispensable) de este material

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Herencia
- Casteo
- Polimorfismo
- Operaciones y Clases Abstractas
- Asociaciones entre Clases
- Colecciones
- Atributos y Métodos de Clase

Introducción a Orientación a Objetos (1)

- **Orientación a Objetos**:
  - *"Es una técnica de programación, un paradigma."* [Bjarne Stroustrup, 1991, creador de C++]
  - La principal idea de este paradigma es diseñar una estructura de objetos junto con su comportamiento que permita a los objetos interactuar (a través de mensajes) para lograr un objetivo.
- **Lenguaje OO**:
  - *"Es un lenguaje que provee mecanismos que soportan el paradigma de la programación orientada a objetos."* [Ídem]

Introducción a Orientación a Objetos (2)

- Pilares de Orientación a Objetos:
  - Abstracción
  - Encapsulamiento
  - Relaciones
  - Herencia
  - Polimorfismo

Definiciones y Conceptos Básicos (1)

- ¿Qué es un objeto?
  - *"Un Objeto es un concepto, abstracción o cosa con un significado y limites claros en el problema en cuestión"* [Rumbaugh]
  - *"Tiene tres características principales: estado, comportamiento e identidad"* [Booch]
- Identidad:
  - Característica de todos los objetos. Está dada automáticamente.
- Estado:
  - Es el conjunto de valores de sus atributos en un instante dado.
- Comportamiento:
  - Conjunto de operaciones que el objeto puede ejecutar.

Definiciones y Conceptos Básicos (2)

- ¿Qué es una clase?
  - Una clase describe un conjunto de objetos; los cuales son creados a imagen y semejanza de su clase.
- Atributo:
  - Dato que contendrá toda instancia de la clase.
- Operación:
  - Especificación de una función que se le pedirá a un objeto que ejecute. Dice el "*que*" pero no el "*como*".
- Método:
  - Implementación de una operación.

Definiciones y Conceptos Básicos (3)

- Referencia:
  - Una referencia es una variable cuyo tipo de datos es una clase.
  - Se utiliza para manipular objetos, pues una referencia apunta a un objeto (o a ninguno, en cuyo caso se dice "referencia nula").
  - Si la comparación de dos referencias es verdadera, significa que ambas apuntan al mismo objeto.

Definiciones y Conceptos Básicos (4)

- Acceso a elementos de un Objeto:
  - Para acceder a los elementos de un objeto, basta con utilizar el punto luego de una referencia al objeto seguido del nombre del elemento que se desea acceder.
- Modificadores de Acceso:
  - Los modificadores de acceso permiten determinar desde donde se puede acceder a un elemento.

Definiciones y Conceptos Básicos (5)

- Modificadores de Acceso:
  - Los más utilizados (no los únicos) son:
    - **public** permite que el elemento sea accedido desde cualquier porción de código.
    - **private** permite que el elemento sea accesible solamente desde dentro de la propia clase.

Definiciones y Conceptos Básicos (6)

```
tipoAcceso class NombreClase
{
        // Definición de atributos
        tipoAcceso  tipoDato  nombreAtributo;
        // Definición de propiedades
        tipoAcceso  tipoDato  NombrePropiedad
        {
                get {return nombreAtributo;}
                set {nombreAtributo=value;}
        }
        // Operaciones y métodos
        tipoAcceso tipoRetorno NombreMetodo(parámetros)
        {
                ...
        }
}
```


Creación de Objetos (1)

- Constructor:
  - Un constructor es el método que es invocado al momento de crear una nueva instancia de una clase.
  - Su objetivo es inicializar el objeto que se está creando.
  - Por tanto toda clase siempre tendrá al menos un constructor (aún cuando el programador no lo defina).
  - El nombre del constructor debe ser el mismo que el nombre de la clase.

Creación de Objetos (2)

- Constructor (cont.):
  - Una clase puede definir cualquier cantidad de constructores, todos con diferente firma.
  - Los constructores se pueden clasificar como:
    - **Por defecto**: puede haber solo uno (si existe) que no recibe parámetros.
    - **Común**: pueden existir muchos con diferente firma.
    - **Completo**: puede haber solo uno (si existe) que recibe un parámetro por atributo para una completa inicialización de la instancia que se está creando.

Creación de Objetos (3)

- Constructor (cont.):
  - Un constructor puede invocar a otro mediante el uso de la palabra reservada **this** seguida de una lista de parámetros.
  - Los constructores se diferencian del resto de los métodos por no definir un tipo de retorno (ni siquiera **void**).
  - Si el programador no define ningún constructor para una clase, entonces el lenguaje asumirá la existencia de un constructor por defecto.

Creación de Objetos (4)

- Declaración e Instanciación:
  - La **declaración** (definición) de una referencia no es lo mismo que su posterior **instanciación** (creación de un objeto que es apuntado por la referencia).
- Sintaxis:

```
// Declaración
NomClase varRef;
// Instanciación
varRef = new NomClase();
          ó
// Ambas al mismo tiempo
NomClase varRef = new NomClase();
```


Creación de Objetos (5)

- Ejemplo:

```
Persona unaPer;
unaPer = new Persona();

          ó

Persona unaPer = new Persona();
```

Herencia (1)

- Herencia:
  - Es el mecanismo por el cual se permite a una clase heredar los atributos, propiedades, operaciones y métodos de otra clase.
  - Una clase solo puede heredar de una clase, no de muchas (se denomina **Herencia Simple**).
  - La clase de la cual se hereda se denomina **clase base**; la clase que hereda se denomina **clase derivada**.
  - Se heredan todos los elementos de la clase base (aunque lo marcado con **private** en la clase base no sea accesible desde la clase derivada).

Herencia (2)

- Sintaxis:

```
tipoAcceso class NombreClase : NomClaseBase
{
    ...
}
```

- Ejemplo:

```
public class Alumno : Persona
{
    ...
}
```


Herencia (3)

- Subclase Directa e Indirecta:
  - Si la clase B es subclase de A entonces B es subclase directa de A.
  - Si además C es subclase de B entonces C es subclase directa de B e indirecta de A.
- Instancia Directa e Indirecta:
  - Si una instancia "a" es el resultado de **new** A() se dice que "a" es instancia directa de A.
  - Si la clase B es subclase de A y "b" es una instancia resultado de **new** B() se dice que "b" es instancia directa de B e indirecta de A.

Herencia (4)

- La clase Object:
  - La clase **Object** es una clase ya predefinida en el lenguaje y es la clase base de todas las demás clases.
  - Esto significa que cualquier clase es subclase (directa o indirecta) de **Object**.
- Reemplazabilidad:
  - Si B es subclase de A y "b" es resultado de **new** B() entonces "b" es de tipo B pero también de tipo A.

Herencia (5)

- Constructores bajo Herencia:
  - Cuando se crea un objeto de una clase, se invoca primero al constructor de esa clase, pero inmediatamente se invoca al constructor de la clase base y así sucesivamente hasta invocar al constructor de Object.
  - Esto se conoce como "encadenamiento de constructores".
  - Por lo tanto, para crear una instancia de una clase derivada en la jerarquía de clases, se debe invocar a todos los constructores desde esa clase hasta Object.
  - Cada constructor se encarga de inicializar los datos que le corresponden, es decir aquellos de la clase a la que pertenece.

Herencia (6)

- Constructores bajo Herencia (cont.):
  - Mediante **:base**(…) se puede invocar a un constructor de la clase base (pasando o no parámetros).
  - Por tanto, cuando el programador no define ningún constructor y el lenguaje provee un constructor por defecto, ese constructor por defecto lo único que hace es invocar al constructor por defecto de su clase base.
  - Es importante notar que cada clase derivada agrega sus atributos específicos. Por tanto sólo los constructores de esa clase debieran inicializar esos atributos específicos.

Herencia (7)

- Redefinición:
  - Dado que la clase derivada hereda las operaciones de la clase base: ¿Qué sucede si la clase derivada necesita proveer la misma operación que la clase base pero implementada de otra forma?
  - Una operación redefine a otra operación de su clase base cuando le asigna otro método en la clase derivada.
  - La redefinición es útil para brindar el mismo servicio (la misma operación) pero con las especificidades de la nueva clase derivada (otro método).

Herencia (8)

- Redefinición (cont.):
  - En muchos casos la clase derivada desea realizar más acciones de las que están definidas en el método de la clase base. En estos casos no sería deseable tener que repetir todo el método de la clase base para luego agregar el código específico de la clase derivada.
  - Por tanto se permite, desde la clase derivada, invocar al método de la clase base mediante **base.NombreMetodo()**

Herencia (9)

- Sobrecarga:
  - La sobrecarga es la capacidad de permitir que muchas operaciones reciban el mismo nombre pero con diferente firma (diferente cantidad y/o tipo de parámetros).
  - Los constructores son un buen ejemplo de sobrecarga.
  - La diferencia con la redefinición es que ésta última trata de la misma operación con diferentes métodos, mientras que la sobrecarga son diferentes operaciones.

Casteo (1)

- Casteo:
  - El casteo permite redefinir el tipo de una referencia, devolviendo una nueva referencia del tipo casteado.
  - El casteo se hace necesario debido a la reemplazabilidad.
- Tipos de Casteo:
  - **Casteo hacia abajo**: es necesario explicitar "a qué clase se desea bajar", es decir cuál es la clase derivada de destino (pues pueden haber muchas).
  - **Casteo hacia arriba**: es implícito, no hay que escribir nada, pues por reemplazabilidad, un objeto de tipo clase derivada también es de tipo clase base.

Casteo (2)

- Ejemplo:

```
// Alumno es clase derivada de Persona
Persona unaPer = new Alumno();
```

- Sabemos que el tipo real del objeto apuntado por "unaPer" es Alumno (pues apunta a un objeto de tipo Alumno); pero "unaPer" fue declarada como de tipo Persona.
- Entonces a la referencia "unaPer" sólo se le podrán solicitar elementos definidos en Persona (aún cuando sabemos que apunta a un objeto de tipo Alumno).

Casteo (3)

- Por tanto, para poder acceder a los elementos públicos de Alumno desde la referencia "unaPer" se la debe castear de la siguiente forma:

```
(Alumno)unaPer
```

- Muchas veces es conveniente consultar el tipo real de un objeto mediante el operador **is**
- Sintaxis:

```
varRef is NomClase
```

- Ejemplo:

```
if (unaPer is Alumno) ...
```

Polimorfismo (1)

- Es la capacidad de asociar diferentes métodos a la misma operación.
- Existen varias formas de lograr un comportamiento polimórfico.

Polimorfismo (2)

- Formas de obtener Polimorfismo:
  - **Mediante Redefinición**: se redefine un método heredado de la clase base.
  - **Mediante Operaciones Abstractas**: se define (por 1ª vez) una operación abstracta heredada de la clase base.
  - **Mediante Interfaces**: se verá más adelante.
- En los 3 casos se tiene la misma operación con diferentes métodos (cada uno en un lugar diferente)

Polimorfismo (3)

- **Ventajas:**
  - El código "cliente" no tiene porqué saber el tipo real del objeto apuntado por la referencia (sólo basta con que la referencia se declare como de tipo clase base).
  - En caso de agregar una nueva clase derivada con su propio método para una operación, todo el código cliente ya escrito permanece sin cambios, pero con la posibilidad de llamar al nuevo método de la nueva clase derivada.

Operaciones y Clases Abstractas (1)

- Operación Abstracta:
  - Una operación abstracta es una operación sin método.
  - Gracias a la herencia se puede tener una operación definida en la clase base y redefinida en la clase derivada. Es importante notar que una operación será abstracta en aquella clase que no le provee un método.
  - Es decir, una operación puede ser abstracta en una clase, pero concreta en otra.

Operaciones y Clases Abstractas (2)

- Clase Abstracta:
  - ¿Una clase abstracta es una clase que no puede instanciarse?
  - Un objeto es **instancia directa** de la clase sobre la que se creó el objeto, e **instancia indirecta** de todas las clases bases de ésa.
  - Las instancias de las clases derivadas de una clase abstracta, serán instancias indirectas de la clase abstracta.
  - Por lo tanto una clase abstracta es una clase de la cual no se pueden obtener instancias directas.

Operaciones y Clases Abstractas (3)

- **Regla:** si una clase contiene (al menos) una operación abstracta, entonces la clase debe ser abstracta.
- Cuidado ya que el recíproco no es cierto: una clase puede ser abstracta sin necesidad de tener alguna operación abstracta.
- Como consecuencia de esa regla, si una clase B hereda una operación abstracta de su clase base A, entonces B tiene dos opciones:
  - Definirle un método en B
  - No definirle un método en B
- En el segundo caso tendrá una operación abstracta (pues la hereda y no la redefine) por lo que debe ser ella también una clase abstracta (tanto A como B serán clases abstractas).

Asociaciones entre Clases (1)

- Una **asociación** entre dos clases es una relación entre ambas que permitirá a sus respectivos objetos estar linkeados durante un cierto tiempo.
- Mientras las clases se **asocian**, sus respectivos objetos se **linkean**.
- Un **link** es entonces una instancia de una asociación.
- La asociación es solo una forma de relacionar dos clases (por ejemplo la herencia es otra).

Asociaciones entre Clases (2)

- La **multiplicidad** indica con cuántos otros se relacionan (linkean) los objetos de cada clase asociada.
- Existen dos multiplicidades: si cada objeto A se relaciona con muchos objetos B, pero cada objeto B se relaciona con 1 solo objeto A, entonces existirá:
  - una multiplicidad de N (de A a B)
  - otra multiplicidad de 1 (de B a A)
- La multiplicidad N se denomin asterisco (*****) en UML.

Asociaciones entre Clases (3)

- La **navegabilidad** indica el sentido en el cual se "conocen" los objetos linkeados:
  - Si cada objeto A conoce los objetos B con los que se relaciona pero un objeto B no conoce los objetos A: **navegabilidad de A hacia B**.
  - Si cada objeto A no conoce los objetos B con los que se relaciona pero un objeto B sí conoce los objetos A: **navegabilidad de B hacia A**.
  - Si ambos se conocen: **doble navegabilidad**.

Asociaciones entre Clases (4)

- Como implementar una Asociación:
  - Los lenguajes orientados a objetos no proveen una construcción específica (es decir una palabra reservada del lenguaje).
  - Por tanto se utilizan referencias.
  - La implementación de una asociación depende de las navegabilidades así como de las multiplicidades.
  - Éstas determinan en qué clase colocar la referencia y el tipo de la referencia (si es una referencia a un objeto o si es una referencia a una colección).

Colecciones (1)

- Conjunto de referencias de tamaño dinámico.
- La primer posición es 0 (cero).
- Propiedades básicas:

  `.Count`
  - determina la cantidad de objetos en la colección.
- Métodos básicos:

  `.Add(objeto)`
  - Permite agregar un nuevo elemento a la colección. Este es colocado al final de la colección (aumentando su tamaño).

Colecciones (2)

- Métodos básicos (cont.):

  `.RemoveAt(posición)`
  - Permite eliminar el elemento de la colección que esta ubicado en la posición indicada.

  `.Clear()`
  - Elimina todos los elementos de la colección (es decir las referencias pero no los objetos referenciados).
- Acceso a un Elemento:

  `[posición]`
  - Permite obtener una referencia al elemento que se encuentra en la posición indicada .

Colecciones (3)

- Colecciones sin tipar: ArrayList
  - Este tipo de colección admite cualquier tipo de elementos.
  - Para poder usarla se debe utilizar el espacio de nombres **System.Collections**
- Colecciones fuertemente tipadas: List
  - List <tipoDato>
  - Este tipo de colección solo admite objetos del tipo especificado; o de subtipos de éste.
  - Para poder usarla se debe utilizar el espacio de nombres **System.Collections.Generic**

Colecciones (4)

- Estructura repetitiva para recorrer una colección:

```
foreach (NomClase   nomVar  in nomCol)
{
    ……
}

foreach (Persona   unaPer  in ColPersonas)
{
    Console.WriteLine(unaPer.ToString());
}
```


Atributos y Métodos de Clase (1)

- Atributo de Instancia:
  - La definición dada anteriormente de atributo se denomina formalmente "Atributo de Instancia", pues cada instancia guardará su propio valor del atributo (el "dueño" del atributo es la instancia).
- Atributo de Clase:
  - Un atributo de clase es aquel cuyo valor es único y compartido entre todas las instancias de la clase (el "dueño" del atributo es la clase).
  - También son llamados "Atributos Estáticos" pues se utiliza la palabra reservada **static** para definirlos.
  - El hecho que sean llamados estáticos no significa que su valor no pueda cambiar (no son constantes).

Atributos y Métodos de Clase (2)

- Una consecuencia de que los atributos de clase pertenezcan a la clase y no a las instancias, es que su valor es independiente del hecho de que existan instancias o no de esa clase.
- Por tanto un atributo de clase puede alojar un valor aún cuando no exista ningún objeto de esa clase (cosa que no puede hacerse con los atributos de instancia).
- Para acceder a un atributo de clase público se utiliza también el punto pero sobre el nombre de la clase, no sobre la referencia.

Atributos y Métodos de Clase (3)

- Sintaxis:

```
public class NomClase
{
        public static tipoDato nomAtributo = valor;
}

nomClase.nomAtributo;
```

- Ejemplo:

```
public class Impuesto
{
        public static int Iva = 22;
}

Impuesto.Iva;
```


Atributos y Métodos de Clase (4)

- Método de Clase:
  - Un método de clase es un método que permite manipular los atributos estáticos de la clase a la cual pertenece.
  - Son la forma de acceder a los atributos de clase independientemente de la existencia de objetos.
  - Para su definición también se utiliza la palabra reservada **static**

Atributos y Métodos de Clase (5)

- Sintaxis:

```
public class NomClase
{
        public static tipoRetorno NomMetodo()
        {       ...     }
}

nomClase.nomMetodo();
```

- Ejemplo:

```
public class Impuestos
{
    public static int Iva = 22;
    public static double CalculoIva(double precio)
    {   return (precio * Impuestos.Iva);  }
}

double total = Impuestos.CalculoIva(500);
```


Manejo de Excepciones (1)

- ***Try – Catch***
  - Esta cláusula permite capturar excepciones en tiempo de ejecución. Todas las excepciones del sistema derivan de la clase base **System.Exception**.
  - Pueden haber varios bloques **Catch**, cada uno capturando una excepción diferente. Estos bloques se deben colocar de excepciones especificas (una clase excepción para un error determinado, como por ejemplo **InvalidCastException**) a generales.
  - En el código del bloque **Catch** se incluyen las sentencias para el tratamiento de la excepción capturada.

Manejo de Excepciones (2)

- Sintaxis:

```
Try
{
        //código a ejecutar con posibilidad de error
}
Catch ( TipoException  ex)
{
        //código a ejecutar en caso de error
}
Finally
{
        //código que siempre se ejecuta con o sin error
}
```


FIN

Programación Orientada a Objetos