

Elementos del Paradigma Orientado a Objetos en C#

Tema 2 – Actividades de trabajo autónomo

Material de la Asignatura

- **Antes de asistir a las clases de laboratorio** deberá estudiar y practicar con todo el material presentado aquí
- En estas transparencias se indica explícitamente la materia que ha de estudiarse antes de cada laboratorio
 - Facilitando así su seguimiento
- Esas transparencias poseen por título
Material Laboratorio X

Material de la Asignatura

- Por tanto el material de esta presentación esta dividido en 4 partes:
 - Material a estudiar para el Laboratorio 1
 - Material a estudiar para el Laboratorio 2
 - Material a estudiar para el Laboratorio 3
 - Material a estudiar para el Laboratorio 4

Visual Studio

- Para compilar y ejecutar los ejemplos es necesario el Visual Studio 2019 o superior (usar vuestro usuario UO):
 - <https://azureforeducation.microsoft.com/devtools>
- También es posible la ejecución de los mismos desde el entorno C# Rider de JetBrains
 - <https://www.jetbrains.com/rider/>
 - Más ligero que Visual Studio
 - Gratuito si se registra con la cuenta de Uniovi, renovable anualmente
 - No obstante, **es necesario saber manejar Visual Studio para los exámenes prácticos**
- Actividad opcional:
 - Si lo desea puede leer antes [https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/ms165088\(v=vs.100\)](https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/ms165088(v=vs.100))

Primer Programa

```
namespace Basico {  
    /// <summary>  
    /// Primera clase de ejemplo  
    /// </summary>  
    class Hola {  
        /// <summary>  
        /// Método de entrada al programa  
        /// </summary>  
        public static void Main() {  
            // * Muestra "Hola Mundo" por consola  
            System.Console.WriteLine("Hola Mundo");  
        }  
    }  
}
```

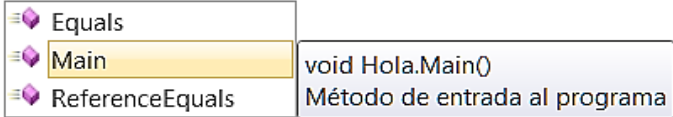
Consulta el código en:

basic/hello

Comentarios

- Los comentarios se pueden escribir en **XML**
 - Visual Studio genera el documento XML si se activa éste en las propiedades del proyecto (por omisión no está activado)
- Otras herramientas como **NDoc** procesan el XML y generan otro tipo de documentación (HTML o LaTeX)
- El propio Visual Studio muestra la ayuda escrita por nosotros como *hints*, conforme escribimos el código

```
class Hola {  
    /// <summary>  
    /// Método de entrada al programa  
    /// </summary>  
    public static void Main() {  
        // * Muestra "Hola Mundo" por consola  
        System.Console.WriteLine("Hola Mundo");  
        Hola.  
    }  
}
```



Ensamblados

- Un ensamblado (*assembly*) es una **colección lógica** de recursos de una aplicación (archivos “.exe”, “.dll”, “.ini”, “.jpg”...)
- Es una unidad reutilizable de implantación (despliegue); **componente**
- La generación de ensamblados puede llevarse a cabo mediante la utilización del **Assembly Linker** (AL.exe) del *.net Framework*
 - Cada proyecto en Visual Studio genera un ensamblado
- Los ensamblados poseen un conjunto de **módulos**: código gestionado (archivos “.exe” y “.dll”)

Espacios de Nombres

- Los espacios de nombres (**namespaces**) en C# son una agrupación **lógica** de tipos, al igual que sucede en C++
 - No poseen ninguna relación con los niveles de ocultación
 - La reutilización física de código se lleva a cabo por medio de ensamblados
 - Los espacios de nombres se pueden anidar
- Para incluir un espacio de nombres, se utiliza la declaración **using** al principio antes de una clase o **namespace**
 - `using System;`
 - No es posible incluir un único tipo de un **namespace** (como en Java o C++)

Nivel de Ocultación de una Clase

- Una clase puede ser **public** o **internal** (por omisión)

```
[public|internal| ] class Hola
```

- Si es interna, sólo se podrá acceder a ella desde dentro de su ensamblado (desde su proyecto)
 - Aquellas clases que se utilizan para implementar una funcionalidad, pero no forman parte de la fachada de un assembly, deberían ser **internal**
- **Pregunta: Cuáles son las diferencias con Java?**

Modularidad en lenguajes OO

- Comparemos los **namespaces** de C# con los **packages** Java
- Hay tres **elementos** a considerar con cada mecanismo:
 1. Agrupación lógica
 2. Agrupación física
 3. Ocultación de la información

C# vs. Java

- **namespaces** de **C#**

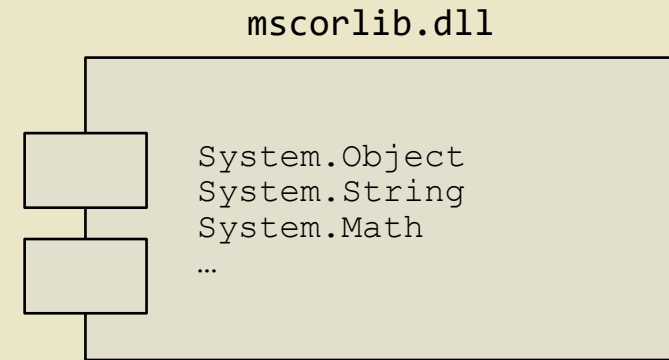
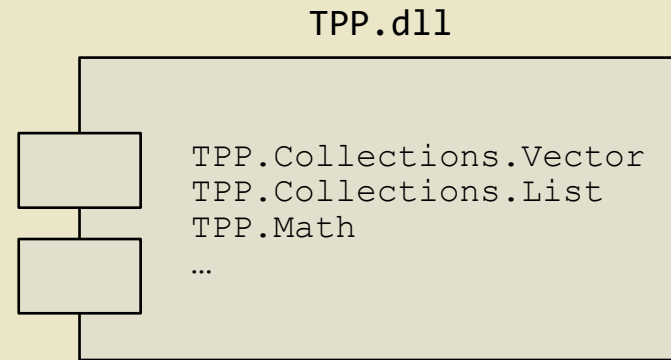
1. Agrupación **lógica** de tipos para evitar colisiones de nombres
2. Los assemblies de .NET (no los namespaces) se usan para la **reutilización física** de código (se debe añadir una referencia al assembly en Visual Studio)
3. No existe ningún nivel de ocultación que restrinja el acceso a los miembros de un namespace (si existe para assemblies)

- **packages** de **Java**

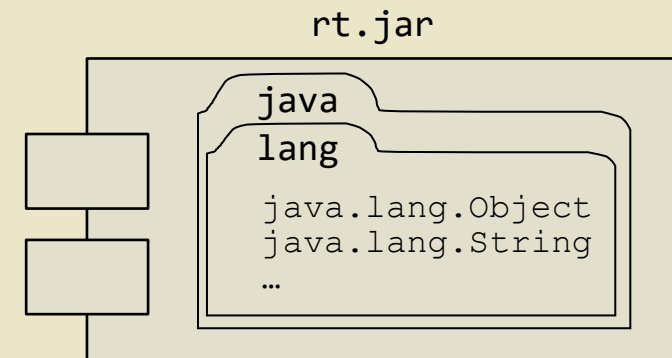
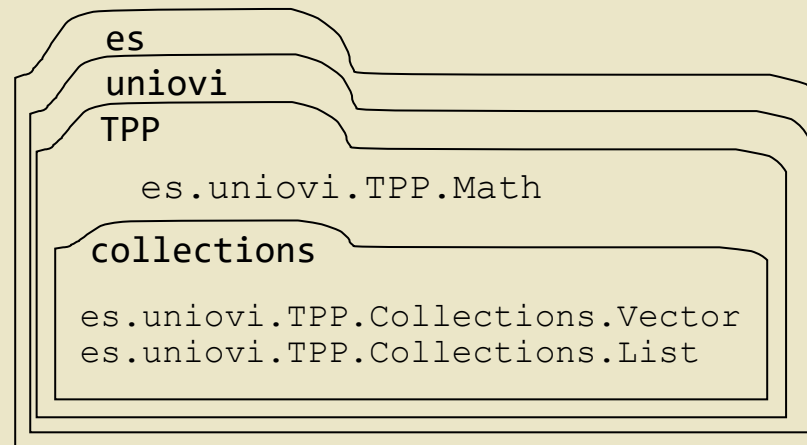
1. Agrupación **lógica** de tipos...
2. ... y también **física** (ficheros **.class**, **.jar** y **.zip** en un directorio incluido en el **classpath**)
3. Existen niveles de ocultación de la información para controlar el acceso a los miembros y tipos desde fuera de un package

C# vs. Java

- **.Net**



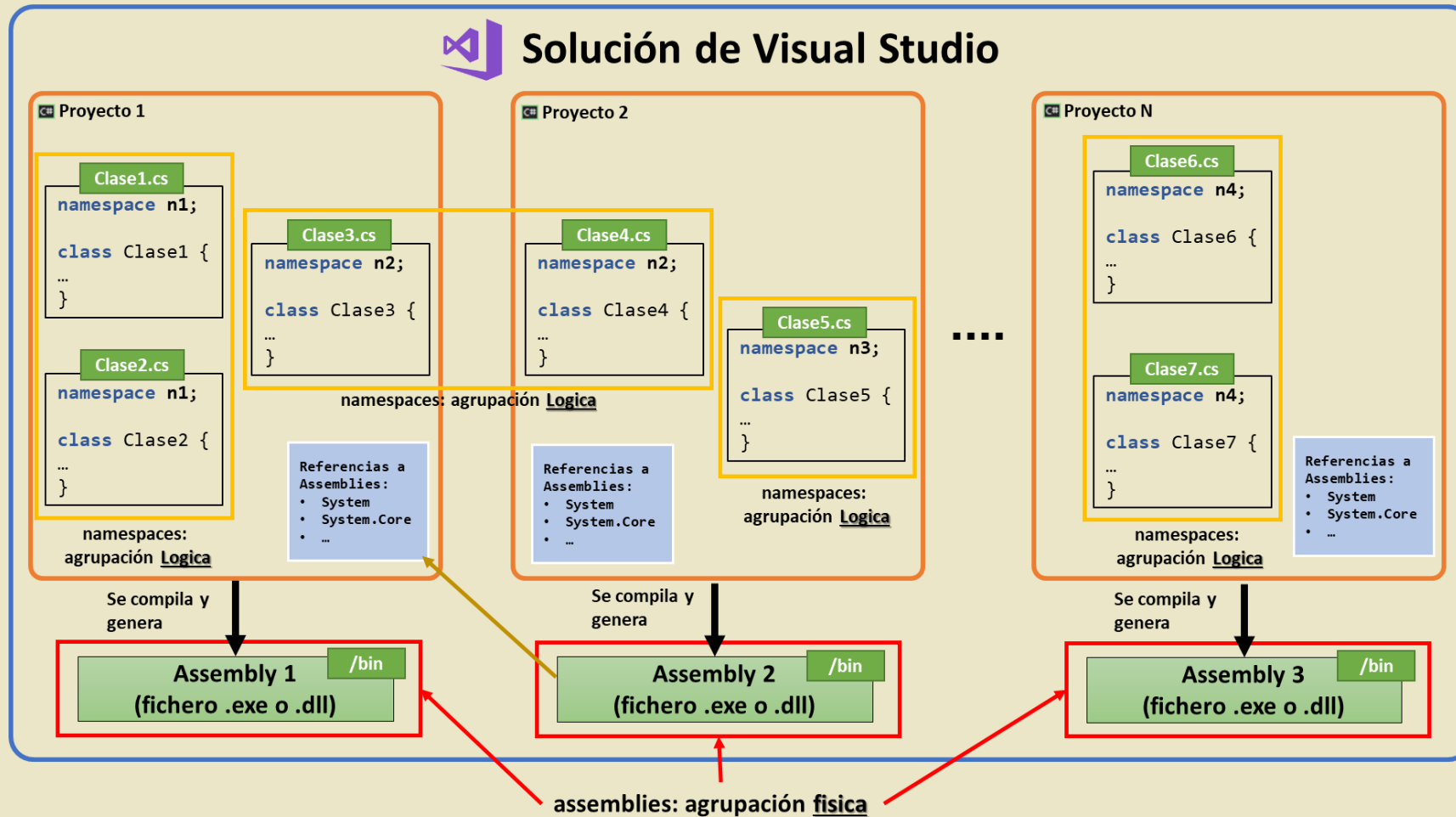
- **Java**



C#

- Los *namespace* se definen entre {}
`namespace TPP.OrientacionObjetos.Basico {`
 ...
`}`
- Para evitar poner siempre el nombre completo de un *namespace* cuando se haga uso de uno de sus tipos se emplea la palabra `using`
`using System;`
 - No se permite incluir un tipo suelto de un *namespace* (si se permite en Java o C++)
- Este mecanismo de control de acceso a tipos y miembros es muy importante para conseguir **acoplamiento débil**

Modularidad en C#



Ocultación de la Información

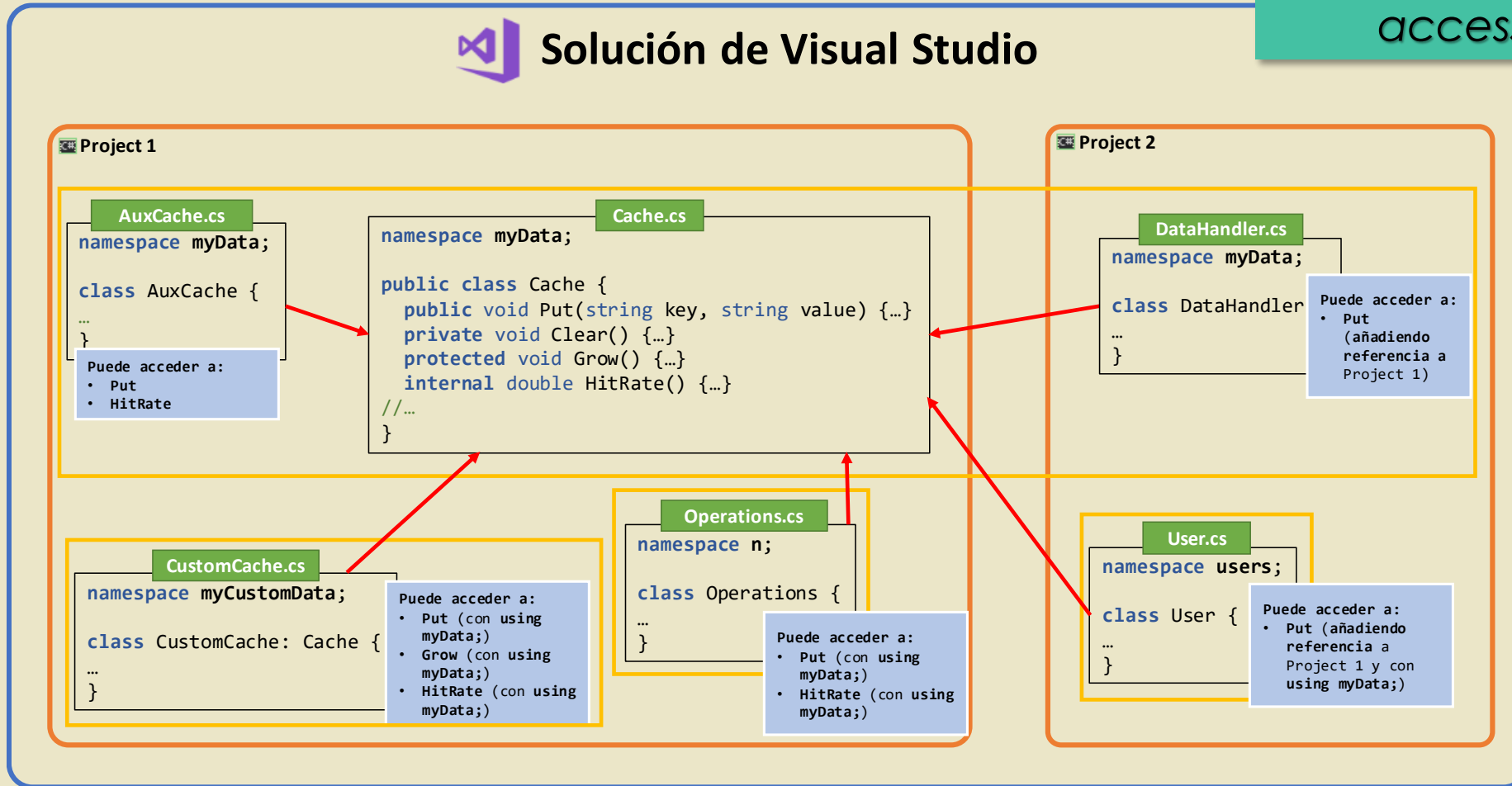
- Niveles de ocultación de los miembros de una clase
 - **public**: Accesible (desde cualquier punto del programa)
 - **private**: Inaccesibles desde fuera de la clase
 - **protected**: Accesible desde dentro de la clase y sus clases derivadas
 - **internal**: Accesible desde cualquier clase del ensamblado
 - **protected internal**: Accesible desde cualquier clase del ensamblado o sus derivadas (aunque no pertenezcan al ensamblado)
- El nivel de ocultación por omisión es **private**

Ocultación de la Información

Consulta el código en:
[access.levels](#)



Solución de Visual Studio



Punto de Entrada

- En un programa sólo puede haber un punto de entrada
- El punto de entrada deberá ser un método de clase (**static**) denominado **Main**
 - Puede retornar nada (**void**) o un entero (**int**)
 - Puede tener cualquier nivel de ocultación
 - Puede declararse
 - Sin parámetros
 - O con un parámetro de tipo **String[]** (los parámetros pasados por línea de comando)

Parámetros Línea de Comando

- Visual Studio se puede configurar para pasar parámetros por línea de comando
- Seleccionando *Proyecto* | *Propiedades* | *Depuración* podremos especificar los argumentos por línea de comando

Tipos Simples

- Se ofrecen conversiones implícitas cuando no hay pérdida de información (conversiones como en Java, no como en C++)
- La conversión explícita se lleva a cabo mediante casts (como Java y C++)

Tipo	Bytes	Descripción
<code>byte</code>	1	Bytes sin signo (valores 0-255)
<code>char</code>	2	Caracteres Unicode
<code>bool</code>	1	<code>true</code> o <code>false</code>
<code>sbyte</code>	1	Bytes con signo (de -128 a 127)
<code>short</code>	2	Valores con signo de dos bytes (de -32,768 a 32,767)
<code>ushort</code>	2	Valores sin signo de dos bytes (de 0 a 65,535)
<code>int</code>	4	Enteros con signo (de -2,147,483,648 a 2,147,483,647)
<code>uint</code>	4	Enteros sin signo (de 0 a 4,294,967,295)
<code>float</code>	4	Reales de precisión simple. De $\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$ con 7 cifras en la mantisa
<code>double</code>	8	Reales de doble precisión. De $\pm 5.0 \times 10^{-324}$ a $\pm 1.8 \times 10^{308}$ con 15-16 cifras en la mantisa
<code>decimal</code>	16	Precisión fija de 28 dígitos, utilizada en cálculos financieros
<code>long</code>	8	Enteros con signo de doble precisión
<code>ulong</code>	8	Enteros sin signo de doble precisión

Constantes y Consola

- C# permite anteponer **const** a la declaración de variables y atributos, identificando así las variables cuyo valor no puede modificarse

```
const double PI = 3.141592;
```

Consulta el código en:

[basic/console](#)

- Las constantes
 - Cadena de caracteres se delimitan por ""
 - Carácter se delimitan por ''
- La consola se controla con la clase **System.Console**
- Los métodos **Write** y **WriteLine** permiten mostrar texto con formato { *índiceParámetro* [, *alineación*] [: *formato*] }
- La salida está internacionalizada

Enumeraciones

- Las enumeraciones son un conjunto finito de posibles valores
- Definen **un nuevo tipo** con su propio grado de ocultación
- Es posible asignarles explícitamente valores enteros

```
enum Colores {  
    azul, verde=3, rojo, amarillo  
}  
  
class Enumerados {  
    static void Main(string[] args) {  
        Colores color;  
        color = Colores.azul;  
        Console.WriteLine(color); // * azul  
        color = (Colores)3;  
        Console.WriteLine(color); // * verde  
    }  
}
```

Operadores

- Un resumen de los operadores de C#

- Aritméticos (enteros y reales)

+ - * / %

- Lógicos

&& || !

- De comparación

== != >= <= < >

- Manipulación de bits

& | ^ ~ >> <<

- Asignación

= += -= *= /= %= &= |= ^= <<= >>=

- Incremento y Decremento

++ -- (prefijo y postfijo)

- Operador ternario condicional:

?:

- El operador + para cadenas implica concatenación

- El resto de operadores está en <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/>

If, While y DoWhile

- **Condicional *If***

`if (expresion) sentencia1`
`[else sentencia2]`

- La expresión ha de ser lógica (`bool`)
- Las sentencias pueden sustituirse por bloques (sentencias anidadas)

- **Iteración *While***

`while (expresion) sentencia`

- La expresión ha de ser lógica (`bool`)
- La sentencia puede sustituirse por un bloque

- **Iteración *DoWhile***

`do sentencia while expresion`

- La expresión ha de ser lógica (`bool`)
- La sentencia puede sustituirse por un bloque

For

```
for ([inicializaciones]; [expresión];  
      [iteraciones]) sentencia
```

- Las ***inicializaciones*** pueden ser declaraciones de variables o sentencias, separadas por coma
 - La ***expresión*** ha de ser lógica (**bool**)
 - Si no se pone, se entiende que es **true**
 - ***Iteraciones*** son un conjunto de sentencias separadas por coma
 - La ***sentencia*** puede sustituirse por un bloque
-
- En los tres bucles vistos se puede utilizar **break** y **continue**

Switch

```
switch (expresion) {  
    case expresion-constante:  
        [sentencia  
         sentencia-salto]  
    [default: sentencia ]  
}
```

Consulta el código en:

basic/switch

- El tipo de expresión ha de ser cualquier tipo de entero (carácter) o **String**
- El tipo de expresión constante ha de ser compatible con el de expresión (puede ser **String**)
- La sentencia salto puede ser de dos formas:
 - **break**
 - **goto case expresion-constante**
- La sentencia salto, si se ha puesto una sentencia en el case es obligatoria
- La sentencia puede sustituirse por un bloque
- [basic/switch](#)

Clases

- Una clase puede ser **internal** (por omisión) o **public**
- Los miembros de una clase pueden ser
 - **public**: Accesible (desde cualquier punto del programa)
 - **private** (por omisión): Inaccesibles desde fuera de la clase
 - **protected**: Accesible desde dentro de la clase y sus clases derivadas
 - **internal**: Accesible desde cualquier clase del ensamblado
 - **protected internal**: Accesible desde cualquier clase del ensamblado de base o sus derivadas (aunque no pertenezcan al ensamblado)

Constructores y Destrucciones

- Un **constructor** define cómo se inicializa un objeto
 - Se llama de forma implícita justo **después** de su construcción
- Un **destructor** define la forma de liberar los recursos usados por un objeto
 - Se llama implícitamente **antes** de su destrucción

Constructores

- Un **constructor** define una inicialización de un objeto, justo después de su construcción
- Es un método
 - De igual nombre al nombre de la clase
 - Que no tiene ningún valor de retorno
- **Por omisión** existe un constructor sin parámetros
- La inicialización por omisión de los campos es:
 - **0** para valores numéricos y enumerados
 - **false** para valores lógicos
 - **'\0'** para caracteres
 - **null** para referencias

Destructor

- En C#, un **destructor** es un método que
 - Se ha de llamar ~ seguido del nombre de la clase
 - No tiene valor de retorno

```
class Clase {  
    public Clase(parámetros) {  
        asignación de recursos adicionales  
    }  
    ~Clase() {  
        liberación de recursos adicionales  
    }  
}
```

Destructor

- **Siempre** que un objeto implemente un destructor, se ejecutará éste previamente a la liberación de su memoria
 - C#: Se asegura su ejecución pero **no** de un modo determinista (no sabemos exactamente cuándo se ejecutará)
 - Java (**finalize**): No se asegura su ejecución (no determinista)
 - C++: Se asegura su ejecución pero de un modo determinista
- El recolector de basura ejecutará el destructor de un objeto cuando éste se libere
- Pregunta: Entonces, cual es la diferencia entre los destructores de C# y los métodos **finalize** de Java?

Objetos

- A los objetos se accede a través de referencias
- Dentro de un método de instancia (no **static**) la referencia **this** nos permite acceder al objeto implícito
 - Objeto utilizado para invocar al método actual
- Los objetos se **crean** con el operador **new**
- A un objeto se le pueden pasar **mensajes** (miembros públicos) con el operador **.**

Consulta el código en:

[encapsulation/constructor.destructor](#)

Clases Parciales

- En ocasiones, una clase ofrece un elevado número de miembros
 - Aunque todos tienen una cohesión, pueden estar a su vez clasificados
 - Puede ser interesante **separarlos físicamente** en distintos ficheros / directorios
- Por ello, en C# se crearon las clases parciales (**partial**)
 - Una clase parcial **se puede implementar en varios ficheros**
 - Es necesario anteponer la palabra reservada **partial**
- Analícese la siguiente simplificación del uso típico de clases parciales

Consulta el código en:
[encapsulation/partial.classes](#)

Static

- La identificación de miembros de clase se realiza anteponiendo la palabra reservada **static**
- El acceso a éstos se hace mediante la utilización de la clase, seguida del operador **.** (**Math.PI**)
 - No es posible utilizar un objeto (como en Java y C++)
- C# ofrece el concepto de **constructor static**
 - Su código se ejecutará cuando se cargue la clase en memoria
 - Se utiliza como mecanismo de inicialización
 - Pregunta: ¿Existe algo similar en Java?
- Para las clases de utilidad (clase con métodos de clase y constructor privado) C# ofrece el concepto de **static class**

Clases de Utilidad

- Para las clases de utilidad (clase con métodos de clase y constructor privado) C# ofrece el concepto de **static class**
 - Java no da soporte directo a este elemento
- Una clase **static class** de C#
 - No permite definir atributos de instancia
 - No permite definir propiedades de instancia
 - No permite definir métodos de instancia
 - No permite definir constructores de instancia

Consulta el código en:

[encapsulation/utility.classes](#)

```
<<utility>>  
Math  
    E, PI: double  
    Sin(double):double  
    Cos(double):double  
    ...
```

Propiedades

- C# ofrece el concepto de **propiedad** para acceder al estado de los objetos como si de atributos se tratase, obteniendo los beneficios del encapsulamiento:

Se oculta el estado interno del objeto, ofreciendo un acceso indirecto mediante las propiedades (**encapsulamiento**)

- Se puede cambiar la implementación de la propiedad sin modificar el acceso por parte del cliente
- Las propiedades pueden ser de **lectura y/o escritura**
- Las propiedades en C# pueden
 - Catalogarse con todos los niveles de ocultación
 - Ser de clase (**static**)
 - Ser abstractas
 - Sobrescribirse (enlace dinámico)

Propiedades

- El uso de las propiedades se ha extendido **para ayudar en la construcción de objetos**
 - Si una clase **Persona** posee un constructor sin parámetros
 - Y posee las propiedades públicas de lectura y escritura **nombre, apellido, edad, DNI**
- Se puede construir un objeto **Persona** indicando los valores de sus propiedades (no necesariamente todas) en cualquier orden
 - Evita la sobrecarga excesiva del constructor

Consulta el código en:

encapsulation/properties

Propiedades

```
class Persona {  
    string Nombre { get; set; }  
    string Apellido { get; set; }  
    int Edad { get; set; }  
    string DNI { get; set; }  
    static void Main() {  
        Persona maria = new Persona {  
            FirstName = "Maria",  
            Surname = "Herrero",  
            Age = 43,  
            IDNumber = "23746887-F"  
        };  
        Persona juan = new Persona {  
            Surname = "Nadie", Age = 10, FirstName = "Juan" };  
    }  
}
```

Consulta el código en:

encapsulation/properties

Strings

- El tipo **string** forma parte del lenguaje C#
- Es un alias de la clase **System.String**
- Por tanto,
 - **string** (y **String**) son clases
 - sus instancias objetos
 - las variables de este tipo, referencias
- El operador + concatena strings
- Los objetos de tipo **string** son inmutables: no se puede cambiar (modificar) su estado
- Si queremos modificar su estado, debe utilizarse la clase **StringBuilder**

Consulta el código en:

[encapsulation/strings](#)



Material Laboratorio 1

El material ofrecido hasta esta transparencia debe ser **obligatoriamente estudiado** con anterioridad al **Laboratorio 1**

Arrays

- La única característica del lenguaje que nos permite coleccionar **referencias** a objetos son los *arrays*
- Los arrays (vectores) son estructuras de datos con múltiples valores **de un mismo tipo**
- Un array, de cualquier dimensión, **es un objeto**
- Para acceder a éste, necesitamos pues una referencia.
- Un referencia se declara concatenando un par de corchetes `[]` al tipo de cada elemento.

```
int[] arrayEnteros;  
bool[] vectorValoresLógicos;  
Angulo[] vectorObjetosAngulo;
```


Arrays

- Un objeto array se crea con el operador **new**, indicando además el tamaño del array

```
arrayEnteros=new int[10];  
vectorValoresLógicos=new bool[2];  
vectorObjetosAngulo=new Angulo[91];
```

- Se reserva memoria para albergar el número de variables indicadas: 10 enteros, 2 valores lógicos y 91 **referencias**
- Los arrays se indexan desde 0 hasta longitud-1
- En el caso de arrays de objetos, tenemos reservado espacio para referencias, **no para los objetos**:

```
for (int i=0;i<91;i++)  
    vectorObjetosAngulo[i]=new Angulo(i);
```

Arrays

- Los valores de los elementos del array, tras su creación son
 - **0** para los valores numéricos
 - **'\0'** para los caracteres
 - **false** para los valores lógicos
 - **null** para las referencias

Arrays

- C# posee una sintaxis para crear *arrays* con una inicialización previa
- Los distintos valores de los elementos del *array*, se enumeran dentro de llaves ({ }), separados por comas

```
char[] digitos = {'0', '1', '2', '3', '5', '6', '7', '8', '9'};
```

```
int[] enteros = { 2, 3, 234, -234, 43 };
```

```
bool[] lógica = { true, false };
```

```
Angulo[] angulos = { new Angulo(0), new Angulo(90), new Angulo(180) };
```

- En una sentencia se puede crear el array y su contenido:

```
Angulo[] angulos;
```

```
angulos = new Angulo[] { new Angulo(0), new Angulo(90), new Angulo(180) };
```

Arrays

- Los *arrays* poseen una propiedad **Length** que devuelve el número de elementos de un *array*
- Se deben iterar, por tanto, con un bucle **for** del siguiente modo

```
for (int i = 0; i < vectorObjetosAngulo.Length; i++)  
    Console.WriteLine(vectorObjetosAngulo[i]);
```

- C# incluyó en su versión 2 el bucle **foreach** con la siguiente sintaxis

```
foreach (Angulo angulo in vectorObjetosAngulo)  
    Console.WriteLine(angulo);
```

- Esta sintaxis no permite modificar los elementos del array

Arrays Multidimensionales

- Existen dos alternativas para crear *arrays* de varias dimensiones
 - **Arrays “lineales”**: Memoria contigua lineal, de varias dimensiones
 - Más eficientes, menos versátiles
 - **Length** es el producto de los tamaños
 - El tamaño de cada dimensión se puede obtener con **GetLength**

```
string[,] vector=new string[3,4];  
for (int i = 0; i < vector.GetLength(0); i++)  
    for (int j = 0; j < vector.GetLength(1); j++)  
        vector[i,j]=" ("+(i+1)+" , "+(j+1)+" )";
```

- **Arrays de Arrays**: permite la construcción de arrays irregulares

```
int[][] triangular=new int[10][];  
for (int i=0;i<triangular.Length;i++)  
    triangular[i] = new int[triangular.Length-i];
```

Consulta el código en:

[encapsulation/arrays](#)

Structs

- En C# un conjunto de campos públicos se puede representar como un **Struct**
 - Pueden tener constructores, propiedades, métodos, campos, operadores y miembros de clase (**static**)
 - Los structs no pueden heredar de otras clases o structs
 - Implícitamente derivan de **ValueType**
 - Sí pueden implementar interfaces
 - No pueden tener destructores
 - La ocultación de sus miembros es, por omisión, **public**
 - Aunque se creen con **new**, ¡siempre se almacenan en la pila!
 - Se crearon para ser utilizados en la transferencia de datos, y no sobrecargar el recolector de basura

Consulta el código en:

[encapsulation/structs](#)

Paso de Parámetros

Consulta el código en:

[overload/parameter.passing](#)

- C# posee tres tipos de paso de parámetros
 - **Paso por valor** (por omisión)
 - El parámetro formal es una copia del parámetro real (argumento)
 - En el paso de objetos, lo que se copia es la referencia \Rightarrow ¡El objeto es el original!
 - **Paso por referencia de entrada y salida**
 - El parámetro formal es un alias del parámetro real (argumento)
 - Se pasa con un valor (entrada) y se le puede asignar otro (salida), modificando el original
 - Se utiliza la palabra reservada **ref** tanto en el parámetro como en el argumento
 - **Paso por referencia de salida**
 - El parámetro formal es un alias del parámetro real (argumento)
 - Se pasa sin valor (entrada) y sirve para devolver más de un valor
 - Se utiliza la palabra reservada **out** tanto en el parámetro como en el argumento

Parámetros Opcionales

- En C# es posible asignar **valores por omisión** a los parámetros
 - Estos siempre tienen que ser los últimos (más a la derecha)

```
/// <summary>
/// Devuelve el listado de personas
/// </summary>
/// <param name="pagina">La página del listado que queremos ver</param>
/// <param name="elementosPorPagina">El número de personas por página</param>
/// <param name="soloMayoresEdad">Que sólo liste personas mayores de edad</param>
/// <returns>La página del listado solicitada</returns>
public Persona[] GetPagina(int pagina = 1,
    int elementosPorPagina = 10, bool soloMayoresEdad = true) {
    ...
}
```


Parámetros Opcionales

Consulta el código en:

[overload/named.optional.parameters](#)

- Permite
 - Invocaciones a una misma función con distinto número de argumentos
listado.GetPagina(3);
 - Evitar el abuso de la sobrecarga
 - No es necesario pasar todos los parámetros con valor por omisión anteriores (se permite “saltar” parámetros por omisión)
listado.GetPagina(soloMayoresEdad: true);
 - Puede utilizar para variar el orden de los argumentos
listado.GetPagina(soloMayoresEdad: false, elementosPorPagina: 10, pagina: 1);
 - Puede utilizarse para mejorar la documentación del código, nombrando los parámetros en la invocación
listado.GetPagina(1, 10, soloMayoresEdad: false);

Sobrecarga de Métodos

- La sobrecarga de métodos permite dar distintas implementaciones a un mismo identificador de método
- En C#, para sobrecargar un método es necesario modificar
 - O el número de parámetros
 - O el tipo de alguno de sus parámetros
 - O el paso de alguno de sus parámetros (valor, **ref** o **out**)

Sobrecarga de Operadores

Consulta el código en:

[overload/operators](#)

- C# ofrece sobrecarga de operadores
 - Emplea, al igual que C++, la palabra reservada **operator**
 - Los operadores son siempre métodos de clase (**static**) \Rightarrow No son polimórficos
 - Sobrecargando un operador (+), obtenemos automáticamente, si ha lugar, su asignación (+=)
 - Sobrecargando ++ o -- prefijo obtenemos automáticamente su versión postfija
 - Permite sobrecargar la **conversión explícita** (cast) y la **implícita** de tipos del lenguaje
 - Puesto que la invocación a un método no es un lvalue, implementa el operador [] con un tipo de propiedad *ad hoc* \Rightarrow los **indexers**

Declaración Implícita de Variables

- Desde C# 4.0, para las variables **locales** no se requiere especificar su tipo

siempre que se asigne un valor en su declaración

```
var vector = new[] { 0, 1, 2 }; // vector es int[]
foreach (var item in vector) {    // item es int
    ...
}
```

- Es útil cuando
 - Los tipos poseen nombres largos (debido a la genericidad)
 - No es sencillo identificar el tipo de la expresión (LINQ)
 - No existe un tipo explícito (los tipos anónimos que veremos más adelante)

Métodos Extensores

- En C# 4.0 se ha añadido la posibilidad de añadir métodos a clases de las que no poseemos el código

- **String, Int32, IEnumerable...**

Consulta el código en:

overload/extension.methods

- Para ello,
 - Hay que implementar un método de clase (**static**)
 - En una clase de utilidad (**static**)
 - Su primer parámetro tiene que ser del tipo que deseamos ampliar
 - El primer parámetro tiene que declararse anteponiendo la palabra reservada **this**

```
static class ExtensoraString {  
    static public uint ContarPalabras(this string cadena) {  
        ...  
    }  
}
```

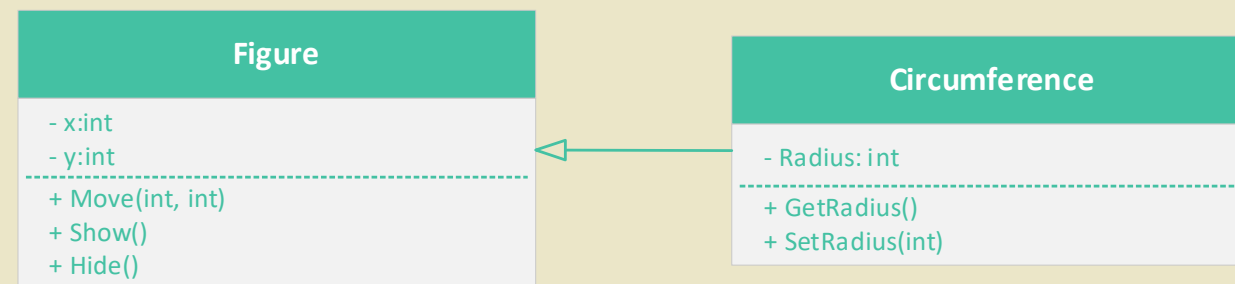
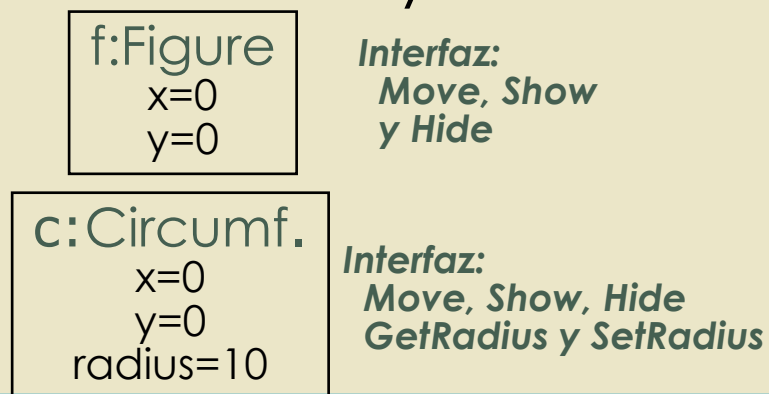
- Se creó para implementar LINQ, añadiendo métodos a **IEnumerable** e **IQueryable**

Unit Testing, Refactoring & TDD

- Actividades de Lectura Obligatoria
 - Léase la siguiente introducción a las pruebas unitarias (**Unit Testing**)
 - http://en.wikipedia.org/wiki/Unit_testing
 - Léase la descripción de **refactoring**
 - http://en.wikipedia.org/wiki/Code_refactoring
 - Lea qué es **TDD**
 - <http://www.agiledata.org/essays/tdd.html>
 - Realizar el siguiente tutorial (**Exercise 1: Red, Green...**)
 - [https://docs.microsoft.com/en-us/previous-versions/gg454256\(v=msdn.10\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/gg454256(v=msdn.10)?redirectedfrom=MSDN)

Herencia

- La herencia es un mecanismo de **reutilización de código** (la herencia de por sí, sin tener en cuenta el polimorfismo)
- El **estado** de una instancia derivada está definido por la unión (herencia) de las estructuras de las clases base y derivada
- El conjunto de mensajes (**interfaz**) que puede aceptar un objeto derivado es la unión (herencia) de los mensajes de su clase base y derivada



Sintaxis

- A todos los efectos, la herencia es una **relación transitiva**:

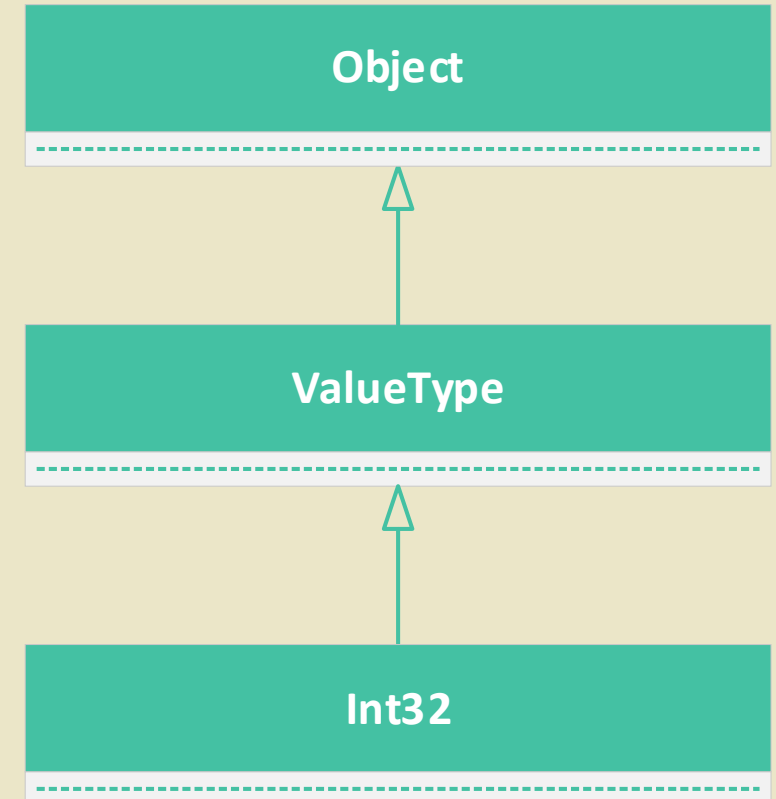
- **ValueType** hereda de **Object**
- **Int32** hereda de **ValueType**
- **Int32** hereda de **Object**

- Sintaxis:

```
[public|internal| ] class Derivada: Base {  
    ...  
}
```

- Si una clase no especifica su clase base, heredará implícitamente de la clase **System.Object**

```
class MiClase {...} // Equivalente a:  
class MiClase: Object {...}
```



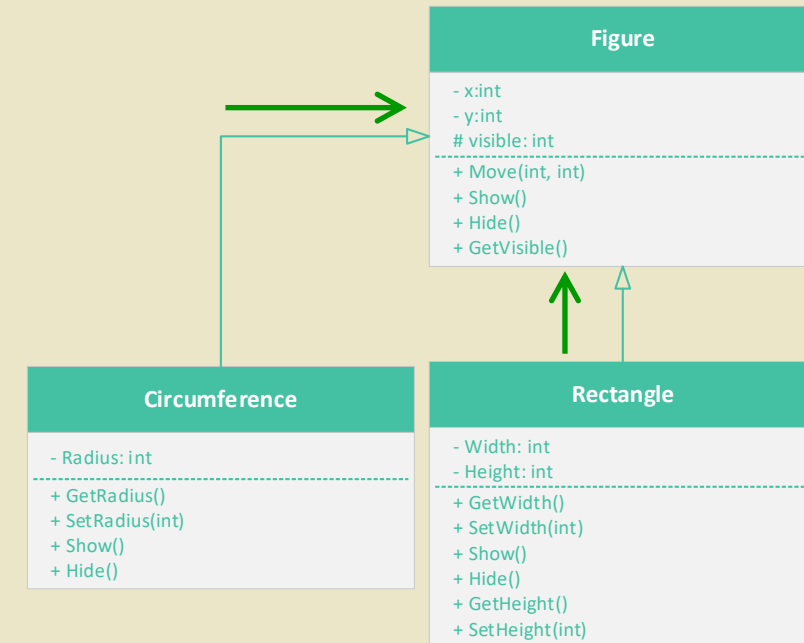
Invocación a Constructores Base

- En todo lenguaje orientado a objetos los constructores derivados han de invocar a los constructores base
 - En Java se hace con **super**
 - En C++ con la lista de inicialización
 - En C# con **base**

```
public Circunferencia(int x, int y, int radio):base(x,y) {  
    this.radio=radio;  
}
```

Polimorfismo

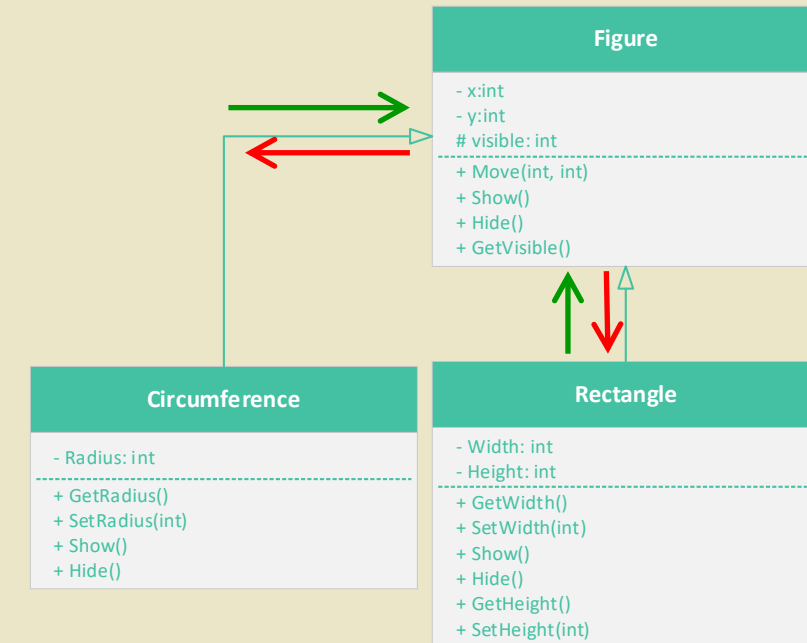
- Es un mecanismo de **generalización**, que hace que la abstracción más general pueda representar abstracciones más específicas
 - El tipo general representa, por tanto, varias formas (*poli morfismo*)
- Por ello, la conversión ascendente en la jerarquía es automática
 - **Las referencias derivadas promocionan a referencias base** (subtipado)



```
void Method(Figure f) {
...
}
```

Polimorfismo

- Cuando se trabaje con referencias “polimórficas” sólo se pondrán pasar los mensajes del tipo de la referencia
 - En nuestro ejemplo, para **f** sólo los mensajes de **Figure**
- Puesto que **f** puede ser una circunferencia o un rectángulo, no tiene sentido pedirle el radio o el ancho
 - Por ello, la conversión descendente ha de forzarse con un ahormado (*cast*)
 - Podrá lanzar la excepción **InvalidCastException** si el objeto no es realmente del tipo solicitado
 - Para conocer el tipo dinámico, se ofrecen los operadores **is** y **as**



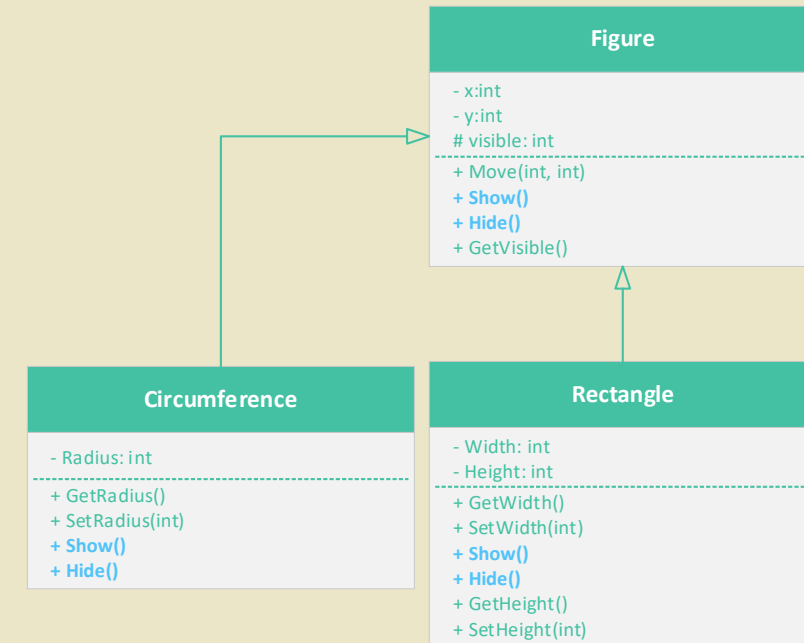
```
void Method(Figure f) {
...
}
```

Enlace Dinámico

- Los métodos heredados se pueden **especializar** en las clases derivadas (por ejemplo, **Show** y **Hide**)
- Pero, ¿qué sucedería en el siguiente código polimórfico? ¿A qué método se llamaría?

```
void Show(Figure f) {  
    f.Show();  
}
```

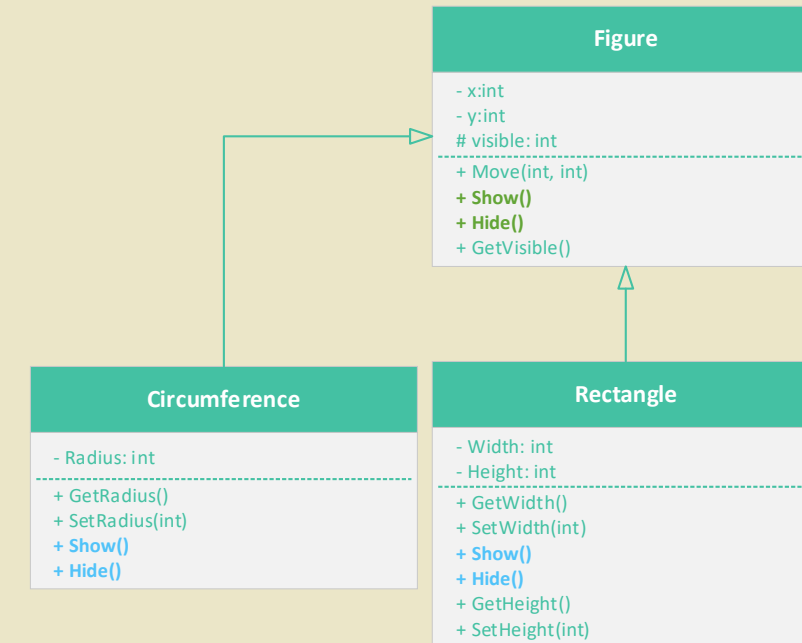
- Si queremos que se llame al método real implementado por el objeto, debemos hacer uso del **enlace dinámico** (*dynamic binding*)
 - Mecanismo por el cual, en tiempo de ejecución, se invoca al método del tipo dinámico implementado por el objeto (no al estático declarado en su clase)



Enlace Dinámico

- C# no tiene enlace dinámico por defecto
- **Para que exista enlace dinámico** en C# tenemos que:
 - Poner la palabra reservada **virtual** al método que reciba el mensaje (referencia)
 - Redefinir (derogar, sobrescribir) su funcionalidad utilizando la palabra reservada **override** en los métodos derivados
- Si simplemente se trata de una coincidencia de nombres de métodos, y no queremos que haya polimorfismo, se pone la palabra reservada **new**
 - Éste es el valor por omisión (pero se muestran *warnings*)
- Estas palabras reservadas se aplican a métodos y propiedades

Consulta el código en:
[inheritance.polymorphism/
dynamic.binding](inheritance.polymorphism/dynamic.binding)



virtual
override

La Clase Object

```
+ Equals(Object):bool  
+ GetHashCode():int  
+ GetType():Type  
+ ReferenceEquals(Object, Object):bool  
+ ToString():String
```

- El método ToString representa un objeto como cadena de caracteres
- Se implementa una vez, y se utiliza en múltiples contextos:
 - Mostrar objetos en consola
 - Para mostrar los mensajes de las excepciones no capturadas
 - Para mostrar los elementos de ComboBoxes
 - Al concatenar (+) a una cadena de caracteres cualquier objeto
 - ...
- Por omisión, **ToString** devuelve una cadena con el nombre del tipo
- Deberemos, pues, redefinir **ToString** en nuestras clases

Comparación de Objetos

- ¿Cómo se comparan dos objetos?
- Podemos estar interesados en saber si
 - Dos objetos son exactamente el mismo \Rightarrow Comparación por **identidad**
 - Dos objetos representan la misma entidad \Rightarrow Comparación por **estado**
- En C#
 - La comparación por identidad se realiza con el operador `==`
 - La comparación por estado se realiza mediante la redefinición (*override*) del método
`bool Object::Equals(Object o)`

GetHashCode

- Por eficiencia, la implementación de **Equals** suele requerir la implementación del método
int Object::GetHashCode()
- Deberá implementarse siguiendo los siguientes criterios
 - En la implementación es necesario que dos objetos iguales (**Equals**) devuelvan el mismo código *hash*
 - Dos objetos que devuelvan un mismo código *hash* no necesariamente serán iguales (**Equals**)
 - Debe retornar un entero de forma rápida
- Actividad Obligatoria: Implemente los métodos **Equals** y **GetHashCode** para la clase **Persona**

Operadores **is** y **as**

- La utilización de polimorfismo hace que tengamos una representación más general que el tipo del objeto (**Object** es más general que **String**)
- En ocasiones, queremos llamar a un **mensaje específico** (de la clase hija) y no es posible
- Esto es común cuando utilizamos **colecciones polimórficas** que hacen uso de **Object**

```
ArrayList lista = new ArrayList();  
lista.Add(new Persona("Pepe", "Pérez", "Martínez", 57));  
// * Error de compilación  
Persona pepe = lista[0];
```

- El problema de hacer un *cast* es que podría lanzar un **InvalidCastException**
 - No estamos seguros de que el objeto introducido sea una Persona (podría ser String...)

Operadores **is** y **as**

- Para ello se introduce el operador **is**

```
if (lista[0] is Persona)  
    ((Persona) lista[0]).CumplirAños();
```

- Si después de utilizar el operador **is** vamos a realizar un *cast*, es mejor utilizar el operador **as**
 - Se obtiene el tipo del objeto con un mejor rendimiento dinámico

```
Persona persona = lista[0] as Persona;  
  
if (persona != null)  
    persona.CumplirAños();
```

Consulta el código en:

inheritance.polymorphism/as.is

AutoBoxing

- En .Net la forma genérica de tratar cualquier elemento es mediante referencias de tipo **Object**
 - Desde la versión 2.0, también es posible obtener este comportamiento mediante genericidad
- Pero los tipos simples (**int**, **char**, **float**, **double...**) ¡no heredan de **Object**!
- Se ha añadido una **conversión implícita** de los tipos simples a **ValueTypes** (derivados de **Object**) de la plataforma .Net
- Por ejemplo, un **int** promociona a un **Int32** y un **Int32** se convierte automáticamente a un **int**

```
private static int Autoboxing(Int32 objeto) {  
    return objeto;  
}  
  
static void Main(string[] args) {  
    // * Boxing  
    int i=3; Int32 oi=i; Object o=i;  
    Console.WriteLine(o);  
    // * Unboxing  
    i=oi; i=(int)o; Console.WriteLine(i);  
    // * Autoboxing mediante paso de parámetros  
    Console.WriteLine( Autoboxing(i) );  
}
```

Consulta el código en:

inheritance.polymorphism/autoboxing

Structs

- En C# un conjunto de campos públicos se puede representar como un **Struct**
 - Pueden tener constructores, propiedades, métodos, campos, operadores y miembros de clase (**static**)
 - Los structs no pueden heredar de otras clases o structs
 - Implícitamente derivan de **ValueType**
 - Sí pueden implementar interfaces
- El autoboxing genera automáticamente structs para los tipos simples
 - **int** ⇒ **System.Int32**
 - **long** ⇒ **System.Int64**
 - **double** ⇒ **System.Double**
 - **char** ⇒ **System.Char**



Material Laboratorio 2

El material ofrecido hasta esta transparencia debe ser **obligatoriamente estudiado** con anterioridad al **Laboratorio 2**

Clases y Métodos Abstractos

- Cuando en una abstracción necesitamos que un **mensaje** forme parte de su interfaz, pero no podemos implementarlo, este mensaje se declara como método abstracto
 - En C# se emplea la palabra reservada **abstract**
 - El método no se implementa (es un mensaje)
- Todo método abstracto ofrece enlace dinámico
 - Al contrario de C++ **no** hay que especificar que es **virtual** (error de compilación)
- En su redefinición, recuérdese que hay que utilizar **override**
- Toda clase que posea, al menos, un método abstracto, será una **clase abstracta**
 - Habrá que declarar ésta como **abstract**

Interfaces

- En ocasiones necesitamos los beneficios del polimorfismo (mantenibilidad) sin poder establecer relaciones de herencia entre abstracciones
 - No existe una relación “real” general / específico
 - El lenguaje no ofrece herencia múltiple
- Lo que necesitamos es un (o más) método (propiedad) común a las distintas abstracciones
- Un interfaz (*interface*) es un conjunto de métodos (y/o propiedades) públicos, que ofrecen un conjunto de clases
 - Realmente constituyen un **conjunto de mensajes**

Sintaxis

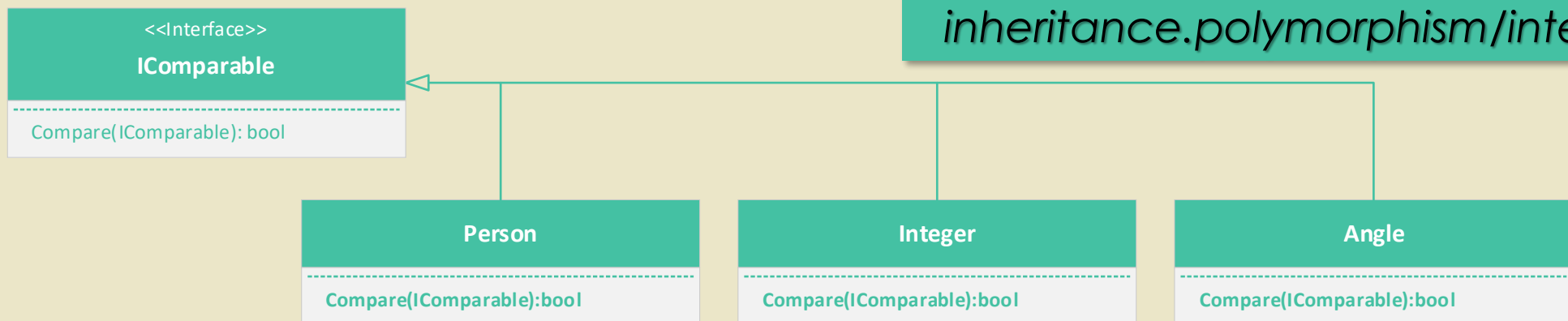
- La sintaxis de declaración de un *Interface* es:

```
[public|internal|] interface  
  nombre[:interfaces-base] {  
    declaración-mensajes  
  }
```

- Es común, aunque no obligatorio, iniciar los identificadores de los interfaces con la letra **I**
- Todos los mensajes de un *interface* son públicos (**no** permite poner **public**, al serlo por definición)
- Todos los mensajes de un *interface* son virtuales (**no** permite poner **virtual**, al serlo por definición)
- Todos los mensajes de un *interface* son abstractos (**no** permite poner **abstract**, al serlo por definición)
- Un *interface* puede heredar de cualquier número de *interfaces*
- No pueden tener miembros **static**

Ejemplo Interfaces

- Si queremos implementar el método de clase **maximo**, su algoritmo sería
 Si **obj1** mayor que **obj2**
 Retornar **obj1**
 Si no, retornar **obj2**
- Independientemente de tipo de **obj1** y **obj2**
- Para ello, identificamos la capacidad (característica) de poderse comparar (**Comparable**)



IDisposable

- Es un interface del namespace **System** con un único método **void Dispose()**



- Su única responsabilidad es **liberar los recursos adicionales** gestionados por el objeto
 - Por tanto, en .Net toda clase que gestiona recursos adicionales debe implementar esta interfaz
 - Ejemplos: **File, Image, HttpApplication, OdbcDataReader, Socket...**

IDisposable y Destructores

- Comúnmente, una clase que defina un destructor, implementará **IDisposable**
 - Desde el destructor se invocará a **Dispose**

```
class Fichero: IDisposable {  
    private string nombreFichero;  
    private bool estaAbierto;  
    public Fichero(string nombreFichero) {  
        this.nombreFichero = nombreFichero;  
        this.estaAbierto = true;  
        Console.WriteLine("Abriendo el fichero {0}.", nombreFichero);  
    }  
    public void Dispose() {  
        if (this.estaAbierto) {  
            this.estaAbierto = false;  
            Console.WriteLine("Cerrando el fichero {0}.", nombreFichero);  
        }  
    }  
    ~Fichero() { this.Dispose(); }  
    ...  
}
```

IDisposable y using

- El invocar explícitamente a un método para liberar recursos es susceptible de ser olvidado
- Puede ser muy tedioso debido al uso de **excepciones** (por eso Java introdujo **finally**)
- Por ello, C# utiliza la palabra reservada **using** para asegurar la liberación de los recursos adicionales de un objeto **IDisposable**
 - Incluso si se lanza una excepción y no se maneja

```
using (Fichero fichero = new Fichero("entrada.txt")) {  
    string línea = fichero.LeerLínea();  
    // Lanza una excepción DivideByZeroException  
    fichero.EscribirLínea(línea + línea.Length/"".Length);  
} // Se cierra el fichero
```

Consulta el código en:

*inheritance.polymorphism
/idisposable*

Implementación Explícita Interfaces

- Puede darse el siguiente caso

```
interface I1 { void m(); }  
interface I2 { void m(); }  
class C: I1, I2 {...}
```

- Y que se quiera dar una implementación distinta a **I1.m** que a **I2.m**
- Para ello se pueden implementar los dos métodos m de forma **explícita**, utilizando la siguiente sintaxis:

```
class C: I1, I2 {  
    public void I1.m() { /* implementación de I1.m */}  
    public void I2.m() { /* implementación de I2.m */}  
}
```

Actividad Opcional

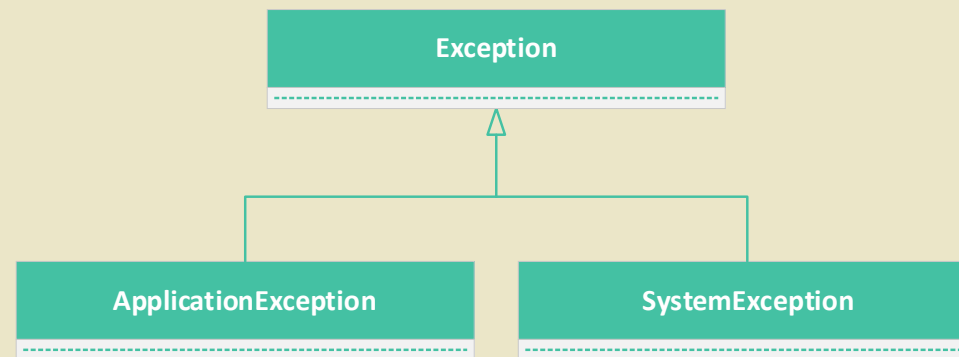
- En los lenguajes con herencia simple, es muy importante conocer cuándo debemos utilizar ésta
 - Sólo podemos derivar directamente de una clase
- La utilización de la herencia como mecanismo de reutilización de código es, a menudo, incorrecta
 - Para ello, suele ser más correcto utilizar **composición**
- Lea el siguiente artículo para entender bien las diferencias y cuando elegir uno u otro
 - <https://www.artima.com/articles/composition-versus-inheritance>

Excepciones

- Una **excepción** es un objeto que encapsula información acerca de un evento irregular ocurrido en tiempo de ejecución
 - Los errores son un tipo (común) de excepciones

Excepciones en C#

- En C# sólo se pueden lanzar excepciones del tipo **System.Exception** (o derivado)
 - No se pueden lanzar excepciones *Value Types* (objetos en pila)
- La clase **System.ApplicationException** supone un medio para crear excepciones definidas por aplicaciones
 - No deben representar un error grave
- La clase **System.SystemException** proporciona un medio para separar las excepciones del sistema de las excepciones definidas por aplicaciones



Lanzando Excepciones (C#)

- Para el lanzamiento de excepciones en C#
 - Se utiliza la palabra reservada **throw** para lanzar excepciones
 - Tiene que ir seguida de un objeto del tipo **Exception** (o derivado)
- C# no ofrece un mecanismo para especificar las excepciones lanzadas por un método
 - En Java es obligatorio
 - En C++ es opcional
- En C# **no es obligatorio** manejar ninguna excepción (al igual que C++)
 - En Java sí

Capturando Excepciones

- Si se utiliza código que pueda lanzar excepciones, puede que nos interese manejar éstas
- Para ello, utilizaremos **try** y **catch**
- Después de **try** se pondrá, entre llaves, el código que puede lanzar alguna excepción
- Después de este bloque, se puede manejar cada tipo de excepción lanzada con un bloque **catch** distinto
- Si queremos que, en cualquier caso, se ejecute un código, éste podrá ubicarse en un bloque **finally**
- En C# las excepciones lanzadas por un método no se tienen que declarar (al contrario que Java)

Consulta el código en:

[exceptions/exceptions](#)

Manejando Cualquier Excepción

- C# ofrece la posibilidad de poner un catch sin parámetros (como C++)
 - Capturando así cualquier excepción
- C# (como C++) ofrece throw sin parámetros para relanzar excepciones

```
public static void m(string s) {  
    switch (s) {  
        case "1": throw new Exception(s);  
        case "2": throw new ApplicationException(s);  
        case "3": throw new SystemException(s);  
    }  
}  
  
public static void Main(string[] args) {  
    try {  
        m(args[0]);  
    }  
    catch {  
        Console.Error.WriteLine("La manejamos  
                                y la volvemos a lanzar.");  
    }  
    throw; }  
}
```

Excepciones No Manejadas

- En C# una excepción no manejada escribe en la salida estándar de error el resultado de invocar a **ToString** que retorna:
 - El tipo de la excepción lanzada
 - Su propiedad **Message** (mensaje pasado al constructor)
 - Su propiedad **StackTrace**

Gestión de Recursos en C#

- El siguiente programa en C#

```
private void procesar(TextWriter fichero) {  
    // * Método de procesamiento de un fichero...  
    fichero.Write("hola"); // sin flush  
    // ... y puede lanzar una excepción  
    throw new Exception();  
}  
private void recursos(String nombreFichero) {  
    TextWriter fichero = new StreamWriter(nombreFichero);  
    // * Procesamos el archivo de cualquier forma  
    procesar(fichero);  
    fichero.Close();  
}  
public void run() {  
    try {  
        recursos("entradaMalGestionada.txt");  
    } catch (Exception) {  
        Console.Error.WriteLine("Excepción capturada.");  
    }  
}
```

No realiza una liberación correcta de recursos (el fichero no se guarda)

Gestión de Recursos en C#

- Se controla la excepción del siguiente modo

```
private void recursosPocoMantenible(String nombreFichero) {  
    StreamWriter fichero = new StreamWriter(nombreFichero);  
    // * Procesamos el archivo de cualquier forma  
    try {  
        procesar(fichero);  
    }  
    catch (Exception e) {  
        fichero.Close();  
        throw e; // * No sabemos cómo manejarla  
    }  
    fichero.Close();  
}
```

- Pero hay que cerrar el fichero en todos los **catch**, siendo una solución poco mantenible

Gestión de Recursos en C#

- Por ello se introdujo `finally`

```
private void recursosMantenible(String nombreFichero) {  
    TextWriter fichero = new StreamWriter(nombreFichero);  
    // * Procesamos el archivo de cualquier forma  
    try {  
        procesar(fichero);  
    }  
    finally {  
        // * Sólo un catch  
        fichero.Close();  
    }  
}
```

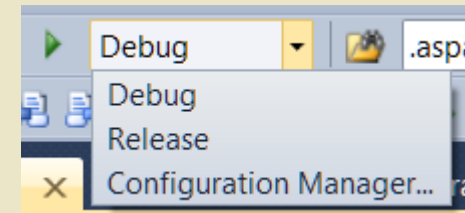
Gestión de Recursos en C#

- Aunque C# ofrece el concepto de **destructor**
 - Recordemos que éste no posee la misma semántica que los destructores de C++
 - Los destructores en C# son invocados cuando se liberan los objetos por el recolector de basura
- Es posible que los destructores de C# sean válidos para liberar determinados recursos
 - Sin embargo, en ocasiones es necesario una **liberación determinista**: especificar el momento exacto en el que queremos liberar el recurso
 - Por ejemplo si ese recurso se va a utilizar en otra rutina posterior
 - En este caso los destructores de C# no son válidos

Aertos en C#

- **Aerto** (aserción): Construcción del lenguaje de programación para asegurar que una condición deba ser siempre cierta
 - Si ésta fuese falsa, se trataría de un error de programación (parando la ejecución)
 - Están orientadas al proceso de desarrollo
 - Se pueden desactivar para la entrega
- La técnica más utilizada para implementar asertos está basada en **compilación condicional**
- C# ofrece compilación condicional

```
#if DEBUG
    Console.WriteLine("Modo Debug");
#else
    Console.WriteLine("Modo Release");
#endif
```



Utilizando Asertos en C#

- Se accede a los asertos mediante la clase de utilidad **Debug** del espacio de nombres **System.Diagnostics**
Assert(bool, [string message,
 [string detailedMessage]])
- Utiliza compilación condicional
 - Si no está definida la macro **DEBUG**, el código no es compilado

Consulta el código en:

exceptions/assert

Precondiciones en C#

- Las precondiciones pueden ser de dos tipos
 1. El método invocado no se puede ejecutar para determinados **valores de los parámetros** (el factorial de un número negativo)
 2. El método invocado no se puede ejecutar para un determinado **estado del objeto implícito** (sacar un elemento de una pila vacía)
- Para ambos casos C# ofrece las dos siguientes excepciones, dentro del *namespace* **System**
 - **ArgumentException** para los argumentos
 - **InvalidOperationException** para los estados de los objetos

Programación por contrato: Pre/Postcondiciones e invariantes

```
public void AddUser(string userName, string plainPassword, UserData data) //no throws clause!!{
    //INVARIANT : Always at the beginning of a method (except constructors). Is object
consistent?
    Invariant();
    int previousUserCount = GetUserCount();

    //PRECONDITIONS are not always wrong parameters: object can be in an invalid state.
InvalidOperationException is used
    if (UserFileIsLocked())
        throw new InvalidOperationException("The file is temporally inaccessible");
    //If arguments have an incorrect value, ArgumentException is used.
    if (!ValidUserName(userName))
        throw new ArgumentException("User name is invalid: please use a non-existing, non-null
user name");
    if (plainPassword.Length < 10)
        throw new ArgumentException("The password size must be at least 10");
    if (!PasswordWithEnoughComplexity(plainPassword))
        throw new ArgumentException("Password must have at least one upper and lowercase char,
number and symbol");
    if (data == null)
        throw new ArgumentException(("Extra user data cannot be null");
    ...
}
```

Programación por contrato: Pre/Postcondiciones e invariantes

```
...  
//TIP: We can create our own exceptions (inheriting from the Exception class) for this, but  
normally  
//ArgumentException and InvalidOperationException are enough for most cases  
  
//Do the work: add user name and data, encrypting the password  
_AddUser(userName, plainPassword, data);  
  
//POSTCONDITION of this method (invariants are object-scoped, postconditions are method-  
scoped)  
Debug.Assert(GetUserCount() == previousUserCount + 1);  
  
//INVARIANT check: Also, always end of a method (leave object consistent)  
Invariant();  
}
```

```
private void Invariant(){  
    //User file cannot get corrupt during the whole execution  
    Debug.Assert(CheckUserFileIntegrity());  
}
```



Material Laboratorio 3

El material ofrecido hasta esta transparencia debe ser **obligatoriamente estudiado** con anterioridad al **Laboratorio 3**

Genericidad

- La **genericidad** es la propiedad que permite construir abstracciones modelo para otras abstracciones
- Ofrece **dos beneficios** principales
 - Una mayor robustez (detección de errores en tiempo de compilación)
 - Una mayor rendimiento (bien implementada)
- Desde **C# 2.0**, es posible definir los siguientes elementos genéricos
 - Clases
 - Structs
 - Métodos
 - Interfaces
 - Delegados
- La **plataforma .Net 2.0** ha sido completamente modificada para soportar genericidad

default(T)

- En C#, **un tipo genérico** puede ser **cualquier tipo del lenguaje**, incluyendo los tipos simples
 - Como en C++
 - De forma distinta a Java, que no incluye los tipos simples
- En ocasiones, queremos asignar o retornar el valor por omisión de un tipo **T**
 - La asignación **variable = null** no sería válida, porque **T** también podría ser un *Value Type* (**int**, **char**...)
- Para ello se utiliza la palabra reservada **default**
- La expresión **default(T)** devuelve
 - **null**, si **T** es de tipo objeto
 - **0**, **'\0'** o **false**, si **T** es de tipo simple

Métodos Genéricos

```
class GenericidadMetodos {  
    public static T ConvertirReferencia<T>(Object referencia) {  
        if (!(referencia is T))  
            return default(T); // valor por omisión del tipo de T  
        return (T)referencia;  
    }  
  
    public static void Main() {  
        Object cadena = "hola", entero = 3;  
        // Conversiones correctas  
        Console.WriteLine(ConvertirReferencia<String>(cadena));  
        Console.WriteLine(ConvertirReferencia<int>(entero));  
        // Conversiones in correctas  
        Console.WriteLine(ConvertirReferencia<int>(cadena));  
        Console.WriteLine(ConvertirReferencia<String>(entero));  
    }  
}
```

Consulta el código en:

[generics/methods](#)

Clases Genéricas

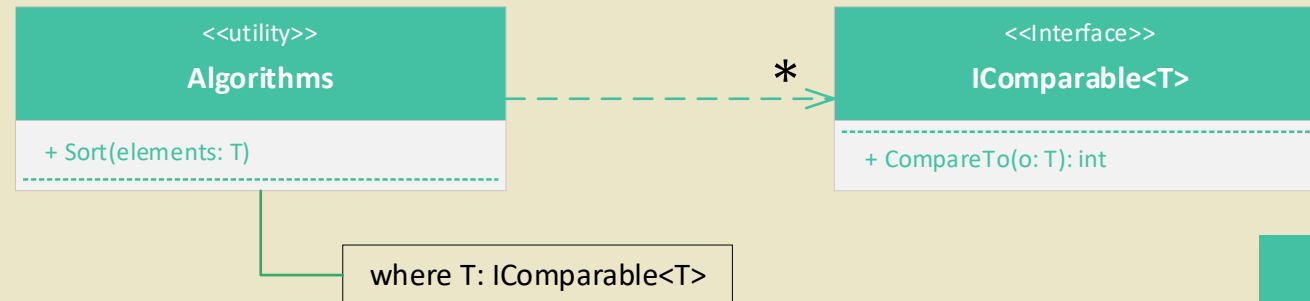
```
class GenericidadClase<T> {  
    private T atributo;  
    public GenericidadClase(T atributo) {  
        this.atributo=atributo;  
    }  
    public T get() {  
        return atributo;  
    }  
    public void set(T atributo) {  
        this.atributo = atributo;  
    }  
}  
  
class Run {  
    public static void Main() {  
        GenericidadClase<int> entero = new GenericidadClase<int>(3);  
        Console.WriteLine(entero.get());  
        GenericidadClase<string> cadena =  
            new GenericidadClase<string>("hola");  
        Console.WriteLine(cadena.get());  
    }  
}
```

Consulta el código en:

generics/classes

Genericidad Acotada

- ¿Qué puedo hacer con los elementos genéricos de una clase?
 - Realmente, los elementos genéricos son Objects
- La **genericidad acotada** (*bounded*) permite hacer más específico estos tipos
- Por ejemplo, se puede hacer un método de ordenación donde se puedan ordenar objetos **Comparable<T>**



Consulta el código en:
[generics/bounded](#)

IEnumerable<T>

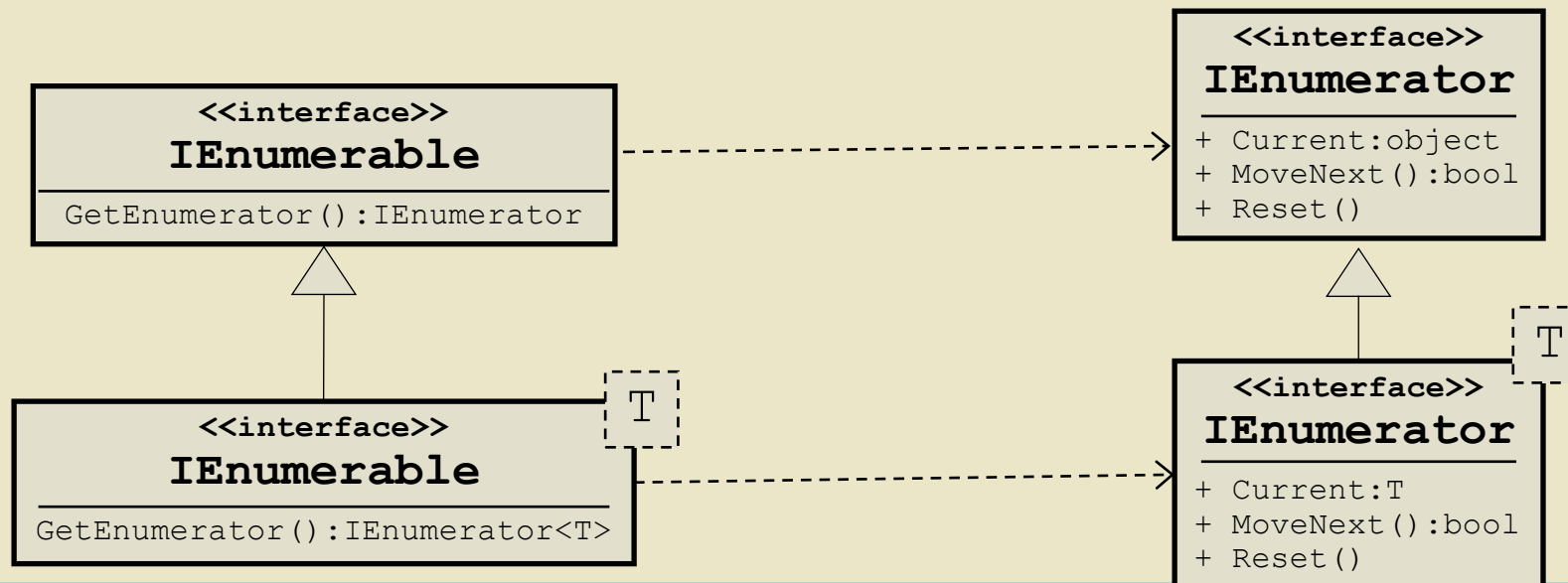
- Este interfaz representa una colección de elementos (genérica)
 - No tiene por qué ser un contenedor (por ejemplo, la generación de la serie de *Fibonacci*)
- Deriva del interfaz polimórfico (no genérico) **IEnumerable**
- Un objeto que implemente **IEnumerable** se puede recorrer con un **foreach**
- Los arrays derivan de **Array** e implementan **IEnumerable<T>**

```
int[] arrayEnteros = new int[] { 10, 99, 50 };  
Array a = arrayEnteros;  
IEnumerable enumerable = arrayEnteros;  
IEnumerable<int> enumerablei = arrayEnteros;
```

IEnumerable<T>

Consulta el código en:
[generics/enumerables](#)

- La interfaz **IEnumerable<T>** sólo posee un método **GetEnumerator** (también **IEnumerator**)
- El método **GetEnumerator** es un factory method (patrón de diseño) encargado de construir un iterador
 - El **IEnumerator** es un bridge (patrón de diseño) para ser independiente de la implementación del iterador
 - El iterador suele implementarse como una clase anidada de la colección



Tipos Anulables

- En ocasiones, se quiere representar que un tipo simple pueda **no** poseer valor (**null**)
- El mejor ejemplo es una base de datos con un campo de tipo simple anulable
- En C# estos tipos se representan añadiendo el sufijo **?** al tipo simple: **int?**, **char?**, **bool?**...
- Estos tipos derivan del *struct* **Nullable<T>**
- Sus dos principales miembros son las dos propiedades
 - **HasValue:bool** (de sólo lectura) Nos indica si el valor no es nulo
 - **Value:T** (de lectura y escritura) En el caso de no sea nulo, nos devuelve su valor; **default(T)** en caso contrario
- También se ha añadido el operador **??**

Consulta el código en:

[generics/nullable](#)

Colecciones

- En la mayoría de las aplicaciones es necesario tener una **abstracción que facilita el acceso de varios objetos** (vectores, pilas, colas, conjuntos, diccionarios...)
- Este tipo de abstracciones se suelen denominar “**contenedores**”
- C# ofrece:
 - Objetos de tipo *array* (vector)
 - Un conjunto de clases, cuyas instancias nos permiten coleccionar otros objetos: las colecciones (**System.Collections**)

Colecciones

- C# tiene dos tipos de colecciones:
 - Polimórficos (versión 1): Utilizan polimorfismo (**Object**) para coleccionar los elementos
System.Collections
 - Genéricos (versión 2): Coleccionan elementos mediante genericidad
System.Collections.Generic
- Cuando sea posible, mejor utilizar los genéricos porque
 - El código es más eficiente
 - Se producen menos errores en tiempo de ejecución
 - El código es más legible y se evitan numerosos casts

System.Collections.Generic

- Las clases más importantes son:
 - **List<T>**: Vector cuyo tamaño es variable dinámicamente
 - **Dictionary<Key, Value>**: Colección de pares clave/contenido organizados mediante *hashing* de la clave
 - **HashSet<T>**: Colección en la que los elementos no pueden estar repetidos (conjuntos)
 - **LinkedList<T>**: Lista doblemente enlazada
 - **Queue<T>**: Colección con política FIFO (Cola)
 - **Stack<T>**: Colección con política LIFO (Pila)
 - **SortedDictionary<T>**: Colección de pares clave/contenido ordenados por la clave
- Su documentación está disponible en el Microsoft Docs
 - <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic>



Material Laboratorio 4

El material ofrecido hasta esta transparencia debe ser **obligatoriamente estudiado** con anterioridad al **Laboratorio 4**