

Paradigma Orientado a Objetos

Tema 2

Material de la Asignatura

- Estas transparencias constituyen un **resumen** de las clases expositivas del tema del Paradigma Orientado a Objetos
- En él utilizaremos el **lenguaje C#**
¡Pero éste no se explicará!
- Los conceptos relativos al Paradigma Orientado a Objetos en C# serán adquiridos
 - Teniendo en cuenta el conocimiento de Java del alumno (Metodología de la Programación)
 - Mediante **actividades** a realizar de forma **autónoma** por parte del alumno
 - En los laboratorios
 - Haciendo las actividades (trabajo no presencial)

Material de la Asignatura

- **Actividad:** de forma **obligatoria**, el alumno debe empezar **ya** a leer las transparencias

Elementos del Paradigma Orientado a Objetos en C# (Actividades)

- En las transparencias donde se haga referencia a código, el alumno deberá **abrir** éste, **analizarlo**, **modificarlo**, **ejecutarlo** y **asegurarse** de que lo **entiende**
 - Este tipo de actividades aparecerán con etiquetas verdes como esta

Consulta el código en:

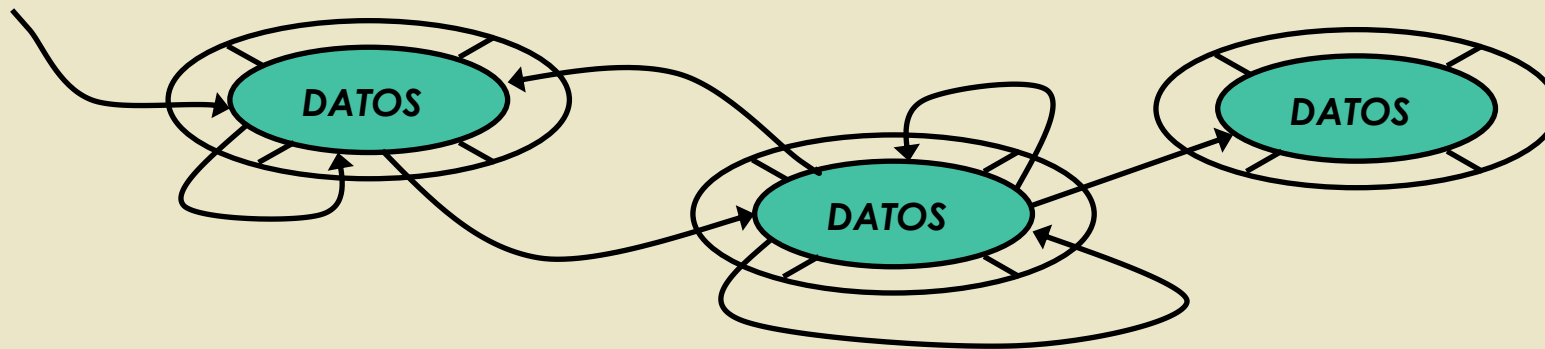
generics/inference

Contenido

- Paradigma Orientado a Objetos
- Encapsulamiento
- Modularidad
- Sobrecarga
- Herencia y Polimorfismo
- Clases Abstractas e Interfaces
- Excepciones
- Asertos
- Genericidad
- Inferencia de Tipos

Paradigma Orientado a Objetos

- Utiliza los **objetos**, unión de datos y métodos, como principal abstracción, definiendo **programas** como interacciones entre objetos
- Se basa en la idea de modelar **objetos reales**, o introducidos en diseño, mediante la codificación de **objetos software**
 - La idea es acercar el modelo del dominio al modelo del programa
- Un programa está constituido por un conjunto de objetos pasándose mensajes entre sí (interactuando)



Abstracción

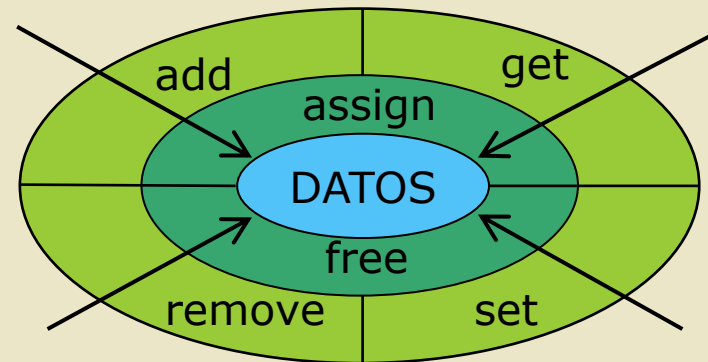
- **Abstracción:** expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás
[Booch, 1996]
 - El principal mecanismo de los lenguajes de programación para representar sus abstracciones son sus tipos

Encapsulamiento

- **Encapsulamiento** (encapsulación): Proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento **[Booch, 1996]**
 - Los objetos encapsulan en una misma entidad datos y comportamiento
- La **ocultación de información** permite discernir entre qué partes de la abstracción están disponibles al resto de la aplicación y qué partes son internas a la abstracción **[Meyer, 1999]**
 - Algunos autores incluyen el concepto de ocultación de información dentro del de encapsulamiento
- Para ello, los lenguajes de programación ofrecen diversos **niveles de ocultación** para sus miembros (atributos y métodos)
- Cada objeto está aislado del exterior y expone una **interfaz** a otros objetos que especifica cómo pueden interactuar con los objetos de la clase

Beneficios del Encapsulamiento

- Supongamos que implementamos una clase **Collection**, únicamente para enteros, con la siguiente interfaz
 - **add**
 - **set**
 - **get**
 - **remove**
- La implementación realizada es mediante una lista enlazada
- ¿Cómo podríamos aumentar su eficiencia?



Beneficios del Encapsulamiento (II)

- Cambiando la lista enlazada por otro tipo de implementación (vector), podríamos aumentar su eficiencia
- Si no modificamos su interfaz, ¡sólo tendríamos que cambiar la implementación de clase **Collection**!
 - El resto de la aplicación no sufriría cambio alguno
 - Encapsulamiento ⇒ **Mantenibilidad**
- El encapsulamiento ofrece una interfaz clara que puede ser empleada en cualquier escenario ⇒ **Reutilización**
- El único modo de manipular las estructuras de datos es mediante unas operaciones bien definidas, evitando así errores de inconsistencia ⇒ **Robustez**

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 (**mantenibilidad**)
- 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)

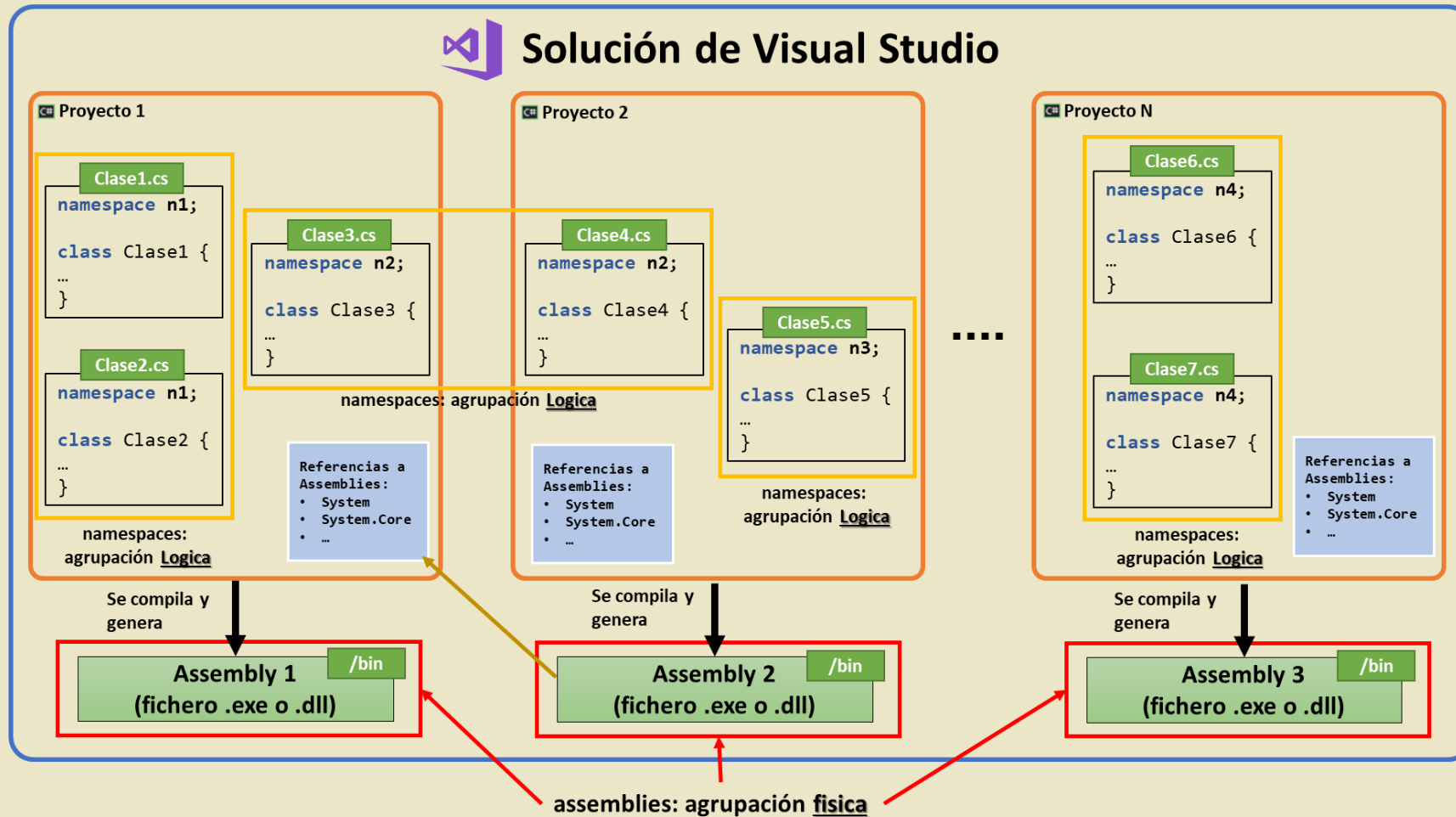
Propiedades

```
public class Circumference {  
    private int x;  
  
    public int X {  
        get { return x; } // Sólo lectura  
    }  
  
    public uint Radus { get; set; } // Lectura y escritura pública  
  
    public int Y { get; private set; } // Sólo lectura pública  
  
    public void Move(int relx, int rely) {  
        x += relx;  
        Y += rely;  
    }  
}
```

Modularidad

- Propiedad que permite subdividir una aplicación en partes más pequeñas (módulos), siendo cada una de ellas tan independiente como sea posible **[Booch, 1996]**
- Cada **módulo** ha de poder ser compilado por separado para ser utilizados en diversos programa (reutilización)
- Distintos elementos pueden constituir un módulo:
 - Funciones y métodos
 - Clases y tipos
 - Espacio de nombres y *packages*
 - Componentes

Modularidad en C#

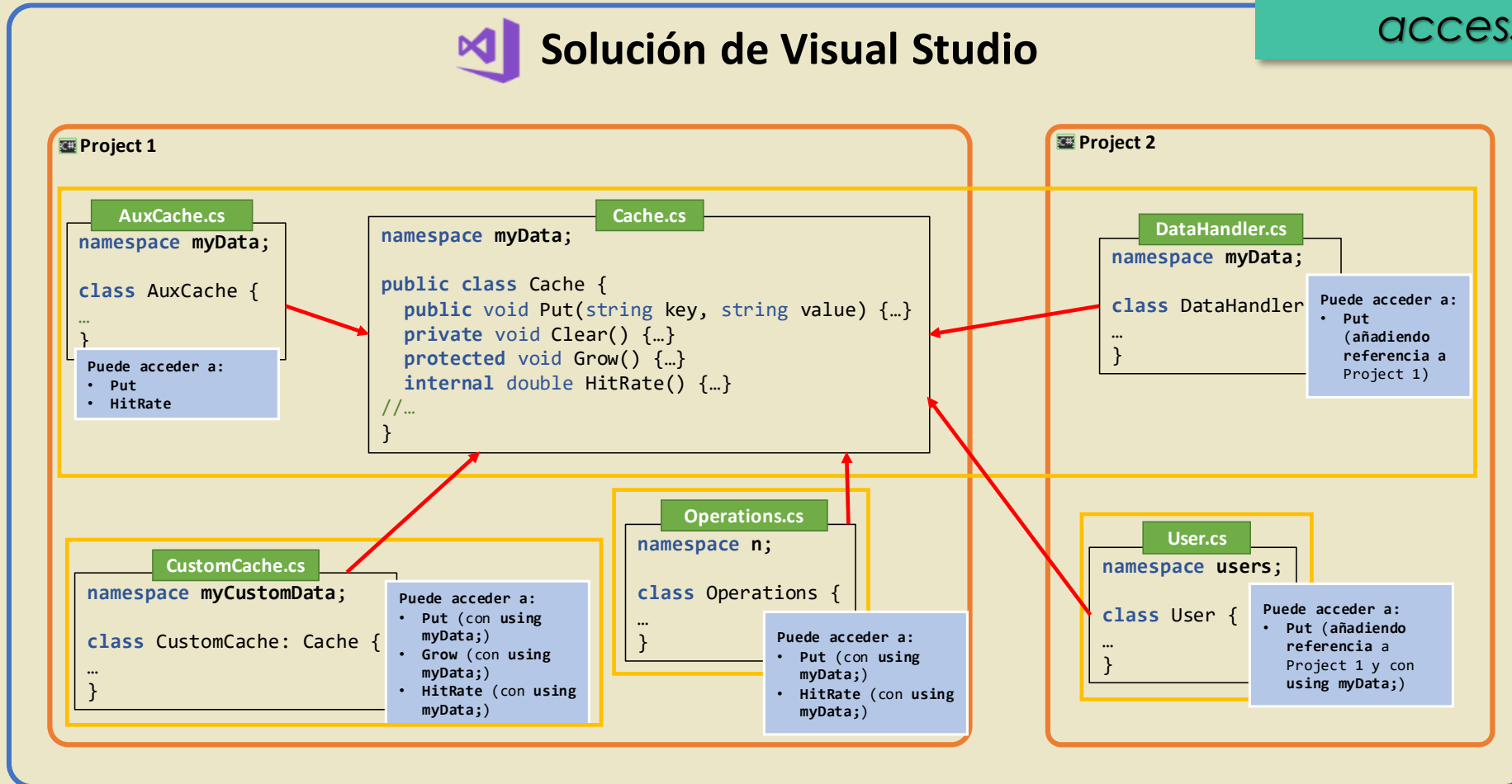


Ocultación de la Información

Consulta el código en:
access.levels



Solución de Visual Studio



Acoplamiento y Cohesión

- Bertrand Meyer enuncia cinco criterios, reglas y principios enunciados de modularidad **[Meyer, 2000]**
- Son comúnmente resumidos en dos
 - **Acoplamiento**: Nivel de interdependencia entre módulos
 - **Cohesión**: Nivel de uniformidad y relación que existe entre las distintas responsabilidades de un módulo
- En desarrollo software, el **bajo acoplamiento** y **elevada cohesión** favorecen la reutilización y mantenibilidad del software

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 que cada método sobrecargado difiera de los otros en al menos uno de los siguientes puntos
 - El número de parámetros
 - El tipo de alguno de sus parámetros
 - El paso de alguno de sus parámetros (valor, **ref** o **out**)
- En Java el último punto no es aplicable

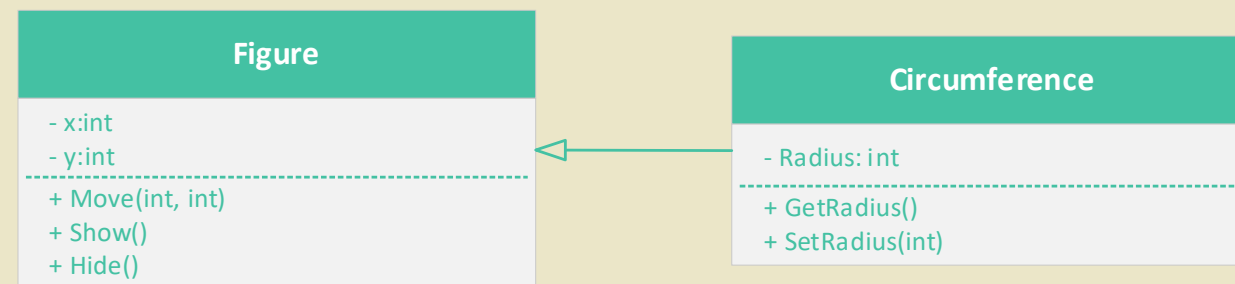
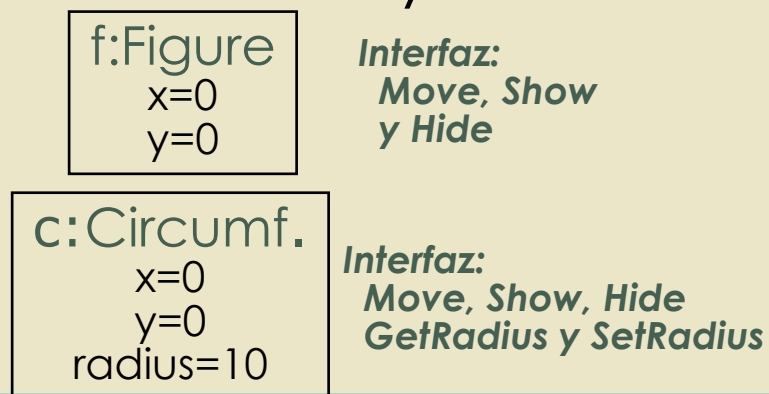
Sobrecarga de Operadores

- La sobrecarga de operadores permite **modificar la semántica de los operadores del lenguaje**
- C# ofrece sobrecarga de operadores incluyendo ++ (pre y post-fijo), -- (pre y post-fijo), [] (*indexers*), el cast y las conversiones implícitas
- Aunque C# ofrece sobrecarga de operadores, ésta apenas se usa

¿Por qué?

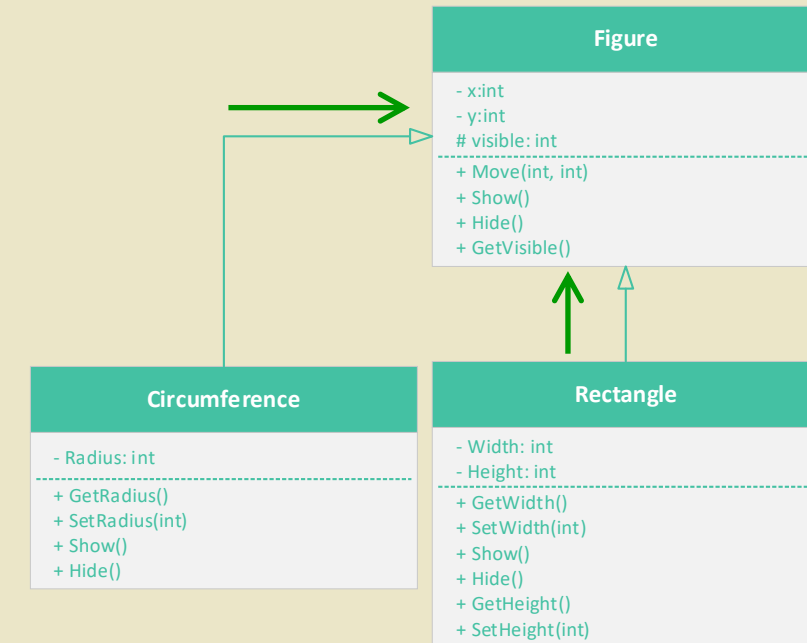
Herencia

- 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



Polimorfismo

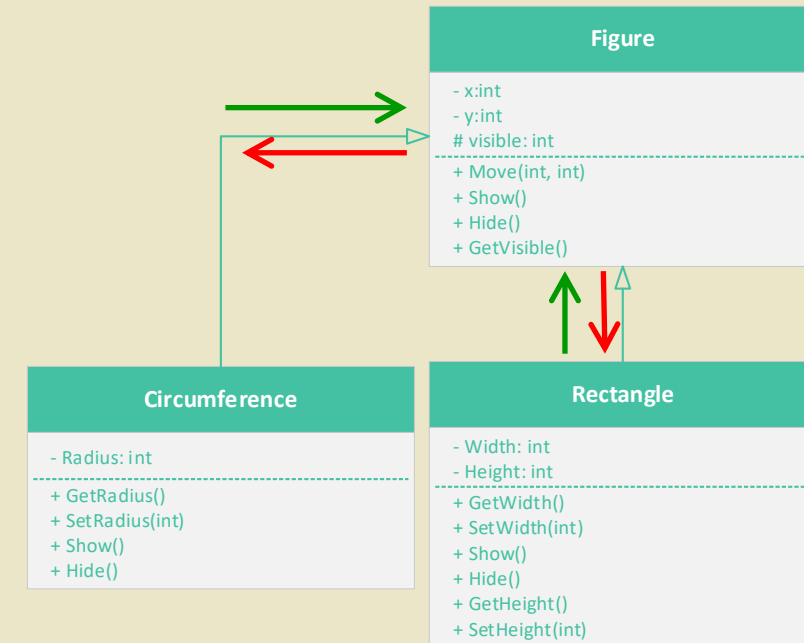
- Es un mecanismo de **generalización**, que hace que la abstracción más general pueda representar abstracciones más específicas
 - El tipo general representa, por tanto, varias formas (*poli morfismo*)
- Por ello, la conversión ascendente en la jerarquía es automática
 - **Las referencias derivadas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



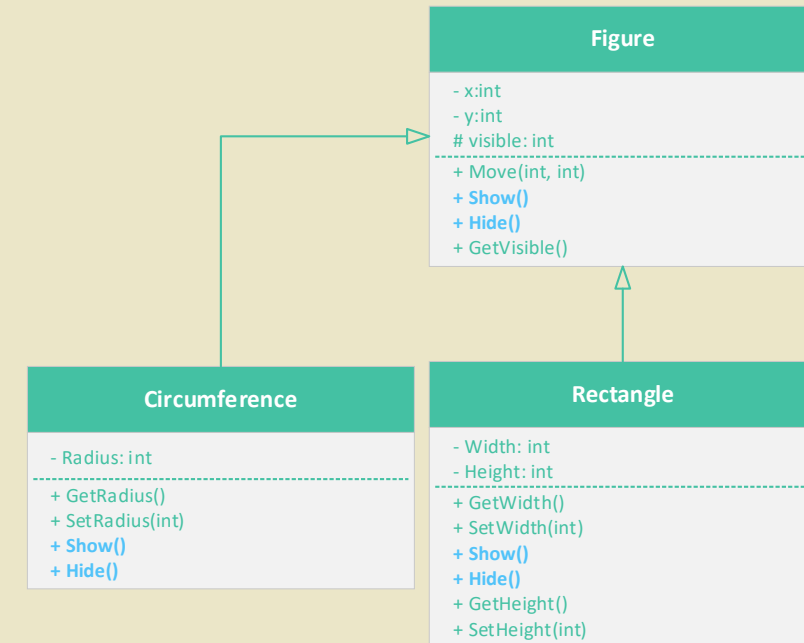
```
void Method(Figure f) {
    ...
}
```

Enlace Dinámico

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

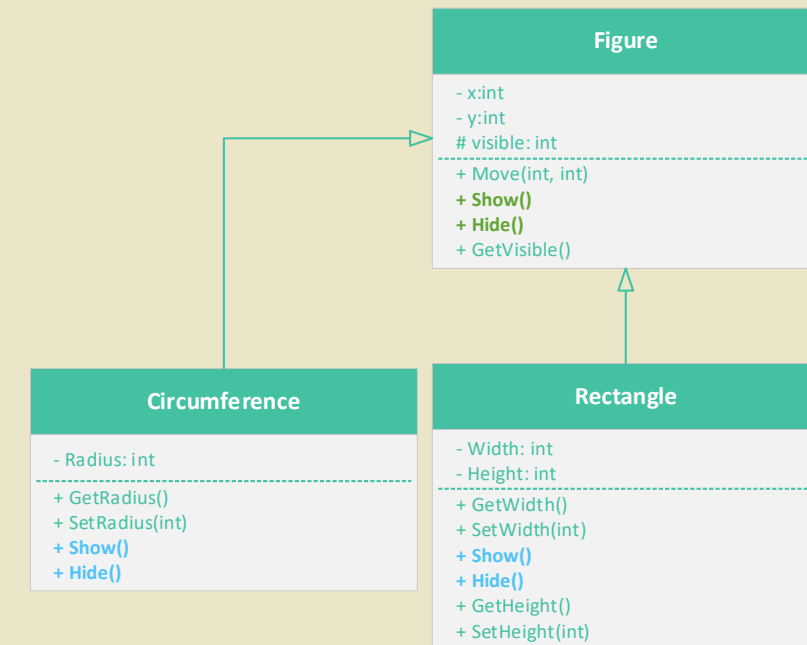
```
void Show(Figure f) {  
    f.Show();  
}
```

- 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
- **Pregunta: ¿Cómo se hace en Java?**



virtual
override

Enlace dinámico por defecto (Java)

- Java permite por defecto el enlace dinámico
- Las clases hijas con capacidad de llamar a un método siempre pueden redefinirlo y cambiar su comportamiento
- ¡Pero esto puede usarse para saltarse o corromper funcionalidades críticas!
- Se puede prohibir el enlace dinámico declarando el método (o la clase) `final`
- Pero esto obliga a elegir cuidadosamente cuáles no permiten hacerlo
- ¿Y si se olvida uno?
- **¡La redefinición ocurre salvo que se prohíba!**

```
public class UserDataController {
    //...
    public boolean CheckPassword(
        String userName, String sha512PwdHash)
    {
        ConnectToDMBS();
        boolean userCorrect = ValidateUser(userName);
        if (userCorrect) return ValidatePassword(sha512PasswordHash);
        return false;
    }
}
//...
public class MaliciousUserDataController
    extends UserDataController
{
    //...
    //Warning: This class destroys the parent ability to properly
    // check passwords!
    public boolean CheckPassword(
        String userName, String sha512PwdHash)
    {
        return true; //Bypass!!
    }
}
//...
public static void Main(String[] args) {
    //...
    //Expects a UserDataController, so assignment is compatible
    //CheckPassword is redefined: every login now returns true!
    Program.SetLoginController(new MaliciousUserDataController());
}
```



Enlace dinámico por autorización (C#)

- C# obliga a que los métodos redefinibles lo indiquen marcándolos como `virtual`
- Si no, métodos con la misma signatura en clases hijas deben marcarse como `new` (warning si no se hace)
- Incluso aunque el padre sea `virtual`, ¡los métodos hijos deben marcarse como `override` para permitir el enlace dinámico!
- Si no se hace, el comportamiento es el mismo del ejemplo
- **¡La redefinición no ocurre salvo autorización expresa!**
- **Así no hay “sorpresas”**

```
public class UserDataController {
    //...
    public bool CheckPassword(string userName, string sha512PwdHash)
    {
        ConnectToDMBS();
        bool userCorrect = ValidateUser(userName);
        if (userCorrect) return ValidatePassword(sha512PasswordHash);
        return false;
    }
}
//...
public class MaliciousUserDataController : UserDataController
{
    //...
    //No dynamic linking.
    //We should use new (public new bool ...)
    public bool CheckPassword(string userName, string sha512PwdHash)
    {
        return true; //Bypass!!
    }
}
//...
public static void Main(string[] args) {
    //...
    //Expects a UserDataController, so assignment is compatible
    //CheckPassword can't be redefined: no bypass is possible,
    //still uses parent method
    Program.SetLoginController(new MaliciousUserDataController());
}
```


Preguntas

- ¿Es correcto el siguiente código?

```
String s = "Hello";  
Console.WriteLine(s);  
Console.WriteLine(DateTime.Now);  
Console.WriteLine(new Angle(0));
```

- En el caso de ser válido,
 - ¿por qué es posible su implementación?
 - ¿qué tendría de positivo?
 - ¿cómo se tendría que haber desarrollado el método **WriteLine**?

Polimorfismo = Código Mantenable

- La implementación de **WriteLine** ha sido desarrollada de diversas formas

WriteLine(int), **WriteLine(char)**, **WriteLine(String)** ...
WriteLine(Object)

- En .Net 1.0, cuando se quiere hacer algo que funcione para todos los objetos, se utiliza **Object**
 - En la versión 2.0 se añadió genericidad al lenguaje y plataforma (no es necesario utilizar **Object**)
- Por tanto, **WriteLine** permite mostrar cualquier objeto
 - ¡Se ha desarrollado un método válido para cualquier abstracción, incluso desarrollada posteriormente!
- ¿Qué método de **Object** utiliza **WriteLine**?

La Clase Object

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

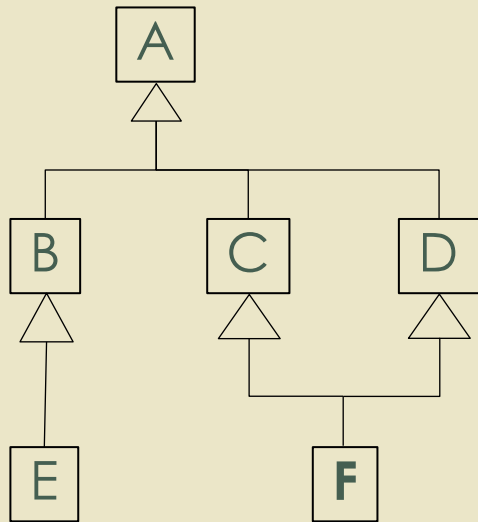
- El método ToString representa un objeto como cadena de caracteres
- Se implementa una vez, y se utiliza en múltiples contextos:
 - Mostrar objetos en consola
 - Para mostrar los mensajes de las excepciones no capturadas
 - Para mostrar los elementos de ComboBoxes
 - Al concatenar (+) a una cadena de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

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**)
- Los métodos abstractos deberían ser redefinidos (recordad poner **override** al hacerlo)
 - Todo método abstracto ofrece enlace dinámico
 - Hay que recordar usar **override** cuando se redefina
- Toda clase que posea, al menos, un método abstracto, será una **clase abstracta**
 - Habrá que declarar ésta como **abstract**
- Una clase abstracta no tiene por qué tener algún método abstracto (se emplearía para reutilizar código)

Herencia Múltiple

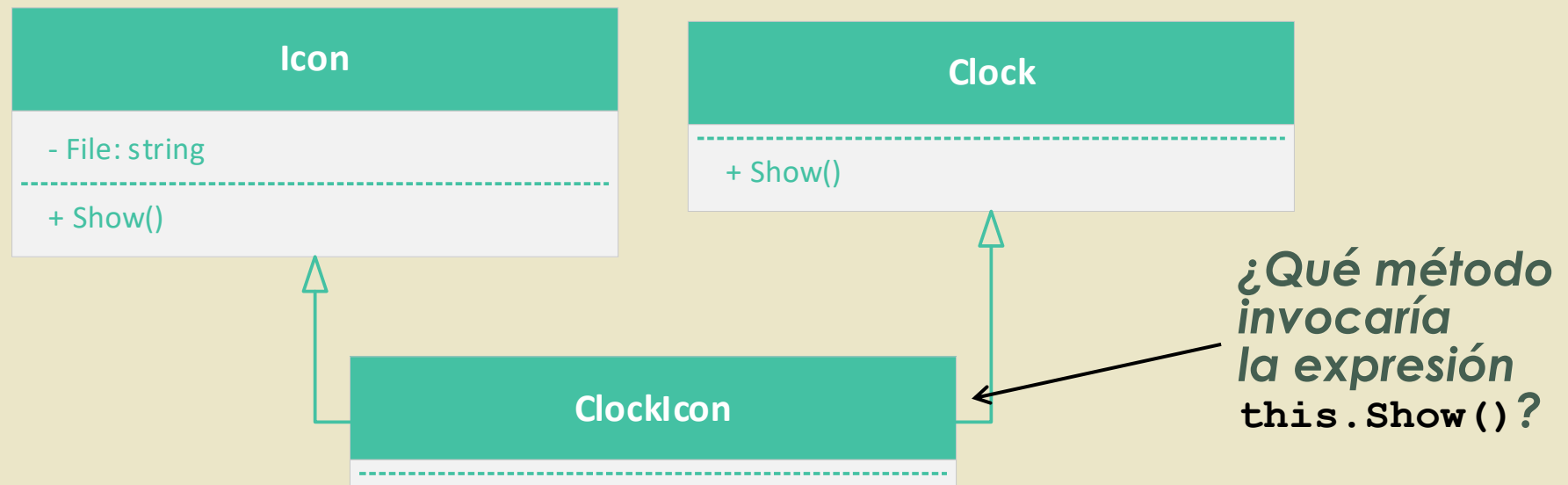
- Se produce cuando una clase hereda, **directamente**, de **más de una clase**
 - Recordemos que la clase es transitiva, por lo que una clase puede derivar indirectamente de muchas clases
- La clase derivada hereda todos los métodos y atributos de sus clases base
- C++, Eiffel o Python son ejemplos de lenguajes que ofrecen herencia múltiple
 - Java y C# no ofrecen herencia múltiple



$$\begin{aligned} \text{miembros}(\text{instancia}(F)) \\ = \\ \text{miembros}(F) \cup \text{miembros}(C) \cup \text{miembros}(D) \cup \text{miembros}(A) \end{aligned}$$

Herencia Múltiple

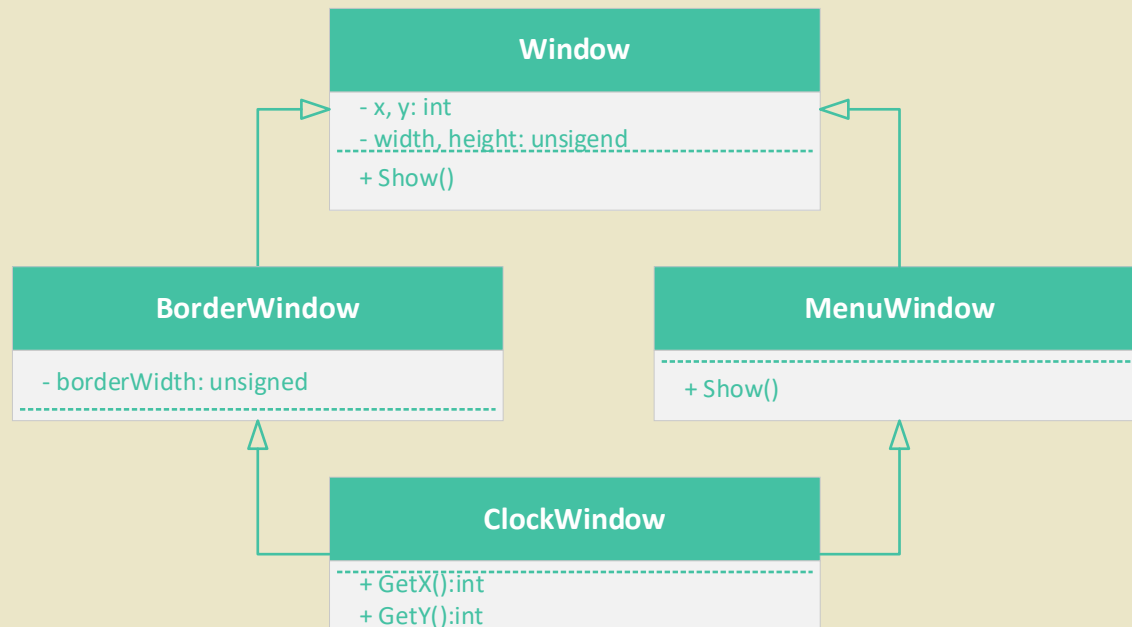
- La herencia múltiple produce dos conflictos:
 - 1. Coincidencia de nombres:** Se produce cuando se hereda de dos o más clases un miembro con igual identificador
Se produce una ambigüedad en su acceso



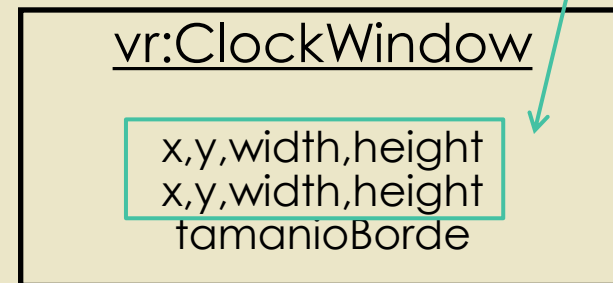
Herencia Múltiple

2. Herencia repetida: Se produce cuando se hereda más de una vez de una clase por distintos caminos

Puede conllevar una duplicidad de miembros



Miembros repetidos

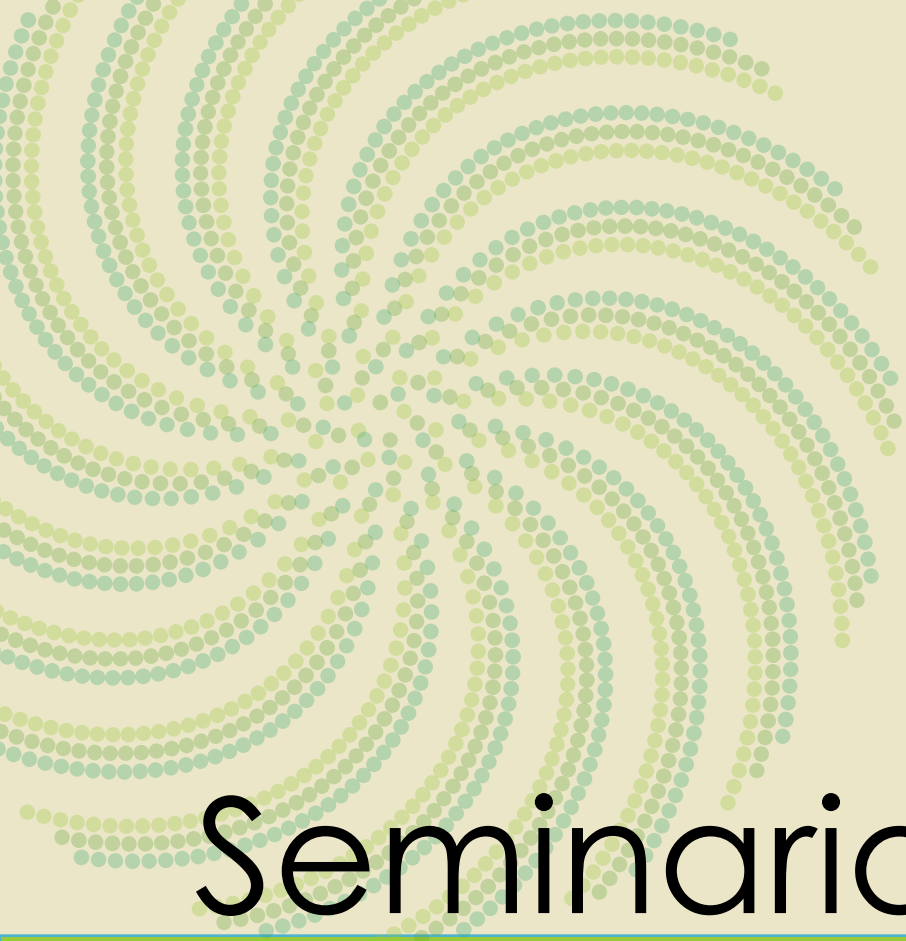


Interfaces

- Debido a los inconvenientes de la herencia múltiple, se analizó el uso de la misma
 1. En la mayor parte de los casos, no se utilizaba **herencia múltiple de implementación** (utilización de herencia para reutilizar implementaciones, sin buscar el polimorfismo)
 2. En la mayoría de los casos, se buscaba **polimorfismo**: que una clase tuviese (promocionase a) múltiples formas (tipos), haciendo uso extensivo de métodos abstractos
- Para el primer caso (poco frecuente), se relegó la herencia a la utilización de **composición**
- Para el segundo caso (mayoritario), se incluyó al lenguaje el concepto de interfaz (**interface**)

Interfaces

- En ocasiones necesitamos que un tipo sea un subtipo de dos o más supertipos
 - Necesitamos **polimorfismo múltiple**, pero
 - No existe una relación “real” general / específico
 - El lenguaje no ofrece herencia múltiple
- Una interfaz (*interface*) es un conjunto de mensajes (y/o propiedades) públicos, que ofrecen un conjunto de clases
- En C#, este concepto se ofrece **como un tipo**
 - Los interfaces se usan para proporcionar **polimorfismo múltiple**
 - Una clase o interface puede derivar (implementar) uno o más interfaces



Seminario 1

Polimorfismo y enlace dinámico

Necesidad de Control Dinámico

- Un compilador no es capaz de detectar la totalidad de los errores de un programa
 - Existen errores que pueden producirse en función del contexto dinámico (en tiempo de ejecución) de un programa
 - Ejemplos: acceso fuera de rango, memoria insuficiente, división por cero, **precondiciones** no cumplidas...
- Es necesario dotar a los **lenguajes de programación** de un mecanismo de control dinámico de errores
 - Históricamente esta gestión se ha hecho con código ad hoc, sin apoyarse en un mecanismo específico de los lenguajes

Objetivos del Manejo de Excepciones

- Los posibles errores en tiempo de ejecución generados por una abstracción no deberían evitar el desarrollo de software
 1. **Reutilizable**: En los diversos contextos en los que se emplee la abstracción, el manejo (y recuperación) de errores puede ser totalmente distinto
 2. **Robusto**: El sistema deberá obligar al programador a gestionar el posible error de forma distinta al flujo general de ejecución
 3. **Extensible**: En función del uso de una abstracción, un error
 - puede transformarse en otros errores
 - puede manejarse, corrigiendo el posible error

Excepciones

- Una **excepción** es un **evento** que se produce en un momento de ejecución y que impide que la ejecución prosiga por su flujo normal
- El mecanismo de manejo de excepciones se basa en la **separación** de:
 - La abstracción que **detecta** el error y “lanza” la excepción (proveedor)
 - Las distintas abstracciones que “**manejan**” la excepción del modo que más les interese (clientes)
- En C#
 - Todas las excepciones son **unchecked** (**RuntimeException** en Java)
 - No se especifican las excepciones lanzadas por un método (**throws** en Java)

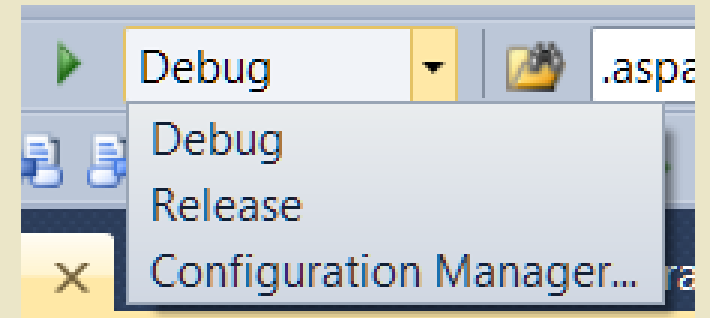
Excepciones y Asertos

- **Asertos** (aserciones) son condiciones que se han de cumplir en la correcta ejecución de un programa
 - Si se producen, se detiene la ejecución del programa
- Los asertos **no se deben utilizar para detectar errores en tiempo de ejecución**
 - No son reutilizables ni extensibles
- Entonces, ¿cuándo se pueden / deben utilizar los asertos? **[Steve McConnell, 2004]**
 - Para detectar aquellas situaciones que nunca deberían ocurrir (postcondiciones, invariantes...)
 - Si ocurren, se trataría de un error de implementación que debemos corregir
 - Deberían deshabilitarse una vez la aplicación haya sido probada exhaustivamente
- Excepciones y asertos son la forma clásica de implementar programación “por contrato”
 - Hay otras APIs como **Code Contracts**: <https://docs.microsoft.com/es-es/dotnet/framework/debug-trace-profile/code-contracts>

Asertos en C#

- Se encuentran en la clase **Debug**, dentro del namespace **System.Diagnostics**
`Debug.Assert(bool condition, string mensaje)`
- Habilitados en modo *Debug* y deshabilitados en modo *Release*
- La técnica más utilizada para implementar asertos está basada en **compilación condicional**
 - Si la macro **DEBUG** no está definida, el código del aserto no se compila
- C# ofrece compilación condicional

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



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());
}
```

Genericidad

- La **genericidad** es la propiedad que permite construir abstracciones modelo para otras abstracciones
- Ofrece **dos beneficios** principales
 - Una mayor robustez (mayor detección de errores en tiempo de compilación)
 - Una mayor rendimiento (bien implementada)
- En **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

Métodos Genéricos

```
class Generics {  
    public static T ConvertReference<T>(Object reference) {  
        if (!(reference is T))  
            //default value of T type (0 for int)  
            return default(T);  
        return (T)reference;  
    }  
  
    public static void Main() {  
        Object myString = "hello", myInteger = 3;  
        // Correct conversions  
        Console.WriteLine(ConvertReference<String>(myString));  
        Console.WriteLine(ConvertReference<int>(myInteger));  
        // Wrong conversions  
        Console.WriteLine(ConvertReference<int>(myString));  
        Console.WriteLine(ConvertReference<String>(myInteger));  
    }  
}
```

Genera, sin que nosotros lo veamos:

```
public static String ConvertReference(Object reference) {  
    if (!(reference is String))  
        return default(String); // null  
    return (String)reference;  
}
```

Genera, sin que nosotros lo veamos:

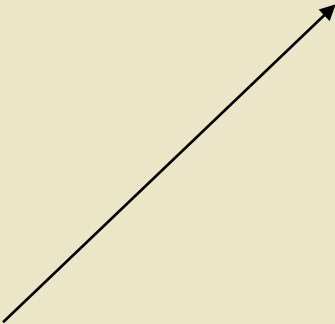
```
public static int  
ConvertirReferencia (Object  
referencia) {  
    if (!(referencia is int))  
        return default(int); // 0  
    return (int)referencia;  
}
```

Clases Genéricas

```
class GenericClass<T> {  
    private T field;  
    public GenericClass(T field) {  
        this.field = field;  
    }  
    public T get() {  
        return field;  
    }  
    public void set(T field) {  
        this.field = field;  
    }  
}  
  
class Run {  
    public static void Main() {  
        GenericClass<int> myInteger = new GenericClass<int>(3);  
        Console.WriteLine(myInteger.get());  
        GenericClass<string> myString = new GenericClass<string>("hello");  
        Console.WriteLine(myString.get());  
    }  
}
```

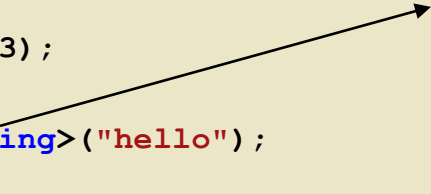
Genera, sin que nosotros lo veamos:

```
class GenericClass {  
    private int field;  
    public GenericClass(int atributo) {  
        this.field=field;  
    }  
    public int Get() {  
        return field;  
    }  
    public void Set(int field) {  
        this. field = field;  
    }  
}
```



Genera, sin que nosotros lo veamos:

```
class GenericClass {  
    private String field;  
    public GenericClass(String field) {  
        this. field = field;  
    }  
    public String Get() {  
        return field;  
    }  
    public void Set(String field) {  
        this. field = field;  
    }  
}
```



Genericidad Acotada

- Cuando tenemos un método genérico

```
T Método<T>(T parámetro) { ... }
```

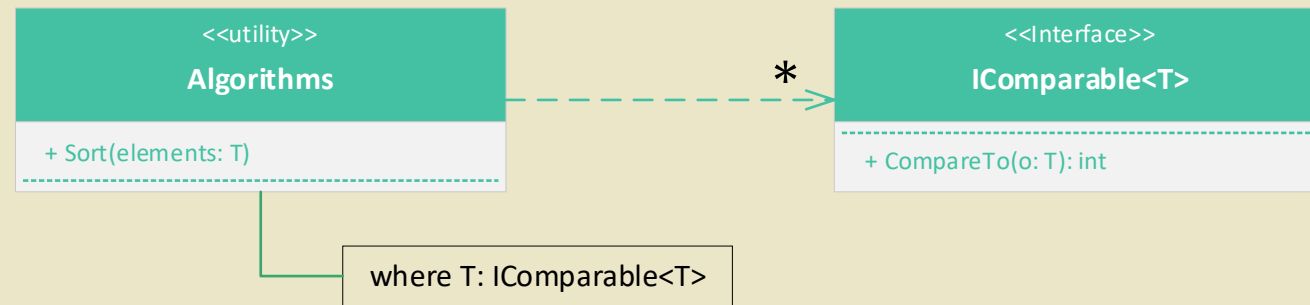
o una clase genérica

```
class Clase<T> {  
    ...  
}
```

- ¿Qué puede hacerse con los elementos de tipo genérico (**T** en nuestro ejemplo)?
- Sólo puedo pasar los mensajes de **Object**
 - Por omisión, los elementos genéricos son **Objects**
- ¿Cómo podría, entonces, implementar un método genérico **Ordenar**?

Genericidad Acotada

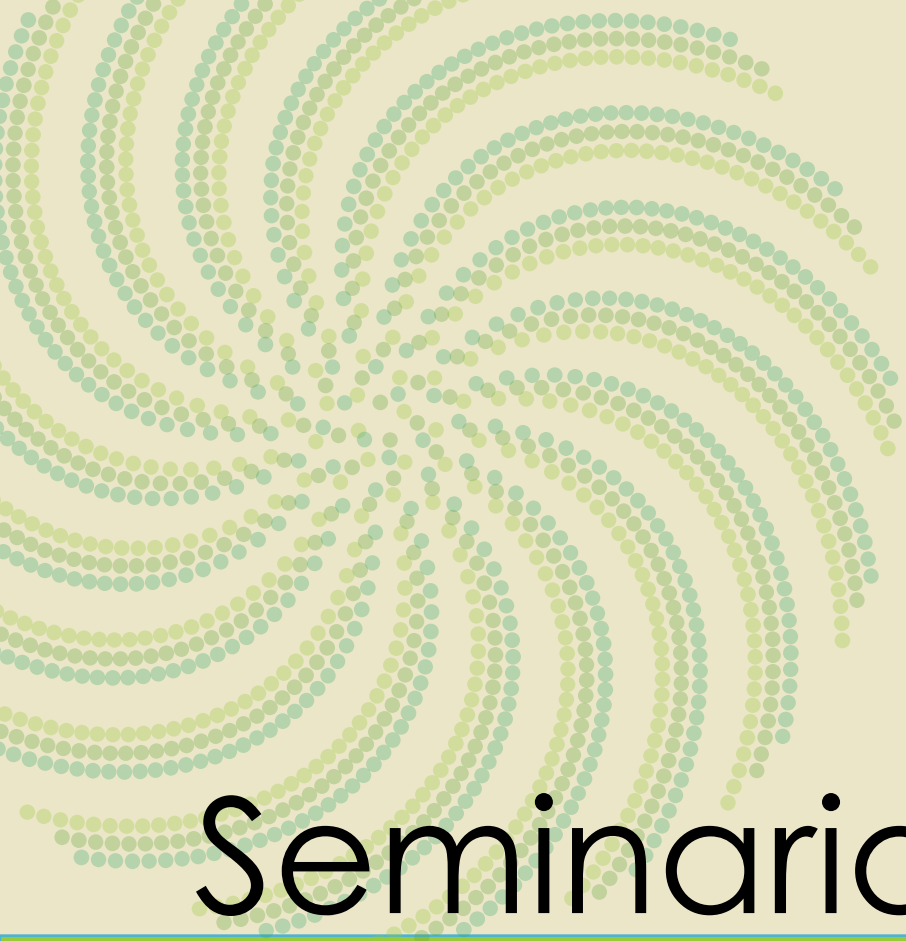
- La **genericidad acotada** (*bounded*) permite hacer que los tipos genéricos sean más específicos
 - Limitan (acotan) su genericidad
 - El beneficio es que se permite un mayor paso de mensajes
- Por ejemplo, se puede hacer un método de ordenación donde se puedan ordenar objetos **IComparable<T>**



```
static T[] Sort<T>(this T[] vector) where T: IComparable<T> {...}
```

- Pregunta: ¿Cuál es la diferencia con?

```
static IComparable<T>[] Sort<T>(this IComparable<T>[] vector) {...}
```



Seminario 2

Genericidad

Genericidad en C#

- El API del *.Net Framework* hace uso intensivo de la genericidad
- A partir de ahora, utilizaremos colecciones genéricas tales como
 - `IEnumerable<T>`
 - `ICollection<T>` y `List<T>`
 - `IDictionary<TKey, TValue>` y `Dictionary<TKey, TValue>`
- Su explicación y ejemplos están en las **actividades**
- Sin conocer estas estructuras de datos **no será posible seguir las explicaciones ni hacer los seminarios y laboratorios**

Genericidad en Java

- Java **1.5+** incorporó la genericidad en el lenguaje de programación
- No obstante, la Java virtual machine (JVM) **no** soporta genericidad
 - El compilador de Java traduce los tipos genericos (Ej., *T*) a `Object`
 - Por tanto, a nivel de la JVM **se usa polimorfismo** en su lugar
- Pero esta técnica tiene varias **limitaciones**
 - Menor capacidad para hacer optimizaciones de rendimiento en tiempo de ejecución
 - No se pueden usar los tipos primitivos como tipos genéricos (Ej. Un `ArrayList` de `int`)
 - No se pueden crear instancias de tipos genéricos (new)
 - No se pueden usar tipos genéricos static
 - No se pueden hacer cast o aplicar instanceof a tipos genéricos
 - No se pueden crear arrays de tipos genéricos
 - No se puede usar sobrecarga con diferentes tipos genéricos instanciados

Inferencia de Tipos

- La **inferencia de tipos** (también llamada reconstrucción de tipos) es la capacidad para deducir automáticamente el tipo de una expresión
- Cuanta menos información de tipos provea el programador (por ejemplo en las declaraciones de variables), más **avanzada** será la inferencia de tipos
- Por ejemplo, el siguiente código ML (F#), infiere el tipo de la función `f a int f (int a, int b)`

```
let f a b =  
    a + b + 100
```

Inferencia de Tipos en C#

- C# ofrece **inferencia de tipos** en tres escenarios principales
 1. Los métodos genéricos
 2. Las variables locales declaradas implícitamente (**var**)
 3. Las funciones lambda (próximo capítulo)

Inferencia en Métodos Genéricos

- En determinados escenarios, la inferencia de tipos permite no especificar el tipo de los métodos genéricos en su invocación

```
static void Swap<T>(ref T lhs, ref T rhs) {  
    T temp; temp = lhs;  
    lhs = rhs; rhs = temp;  
}  
  
static void Main() {  
    int a = 1, b = 2;  
    Swap(ref a, ref b);  
    double c = 3.3, d=4.4;  
    Swap(ref c, ref d);  
    Swap(ref a, ref d); // Compiler Error  
}
```

Consulta el código en:

generics/inference

Variables Declaradas Implícitamente

- En C# es posible no declarar explícitamente el tipo de las variables locales
 - Para ello se utiliza la palabra reservada **var** en lugar de su tipo
 - Hay que asignar una expresión en su declaración

```
var vector = new[] { 0, 1, 2 }; // vector is int[]
foreach(var item in vector) {   // item is 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 (tipos anónimos)
- La inferencia de tipos para variables locales (con **var**) se añadió también a Java a partir de **Java 10**