

# Fundamentos del Paradigma Funcional

Tema 3

# Ejemplos de Código

- Todos los ejemplos de **código** mostrados en estas transparencias están **disponibles para el alumno**
  - Siguen apareciendo en etiquetas como la que se muestra:

Consulta el código en:

*generics/inference*

- Para comprender los conceptos explicados, el alumno deberá **abrir** el código, **analizarlo**, **modificarlo**, **ejecutarlo** y **asegurarse** de que lo **entiende**

# Contenido

- [Cálculo Lambda](#)
- [Funciones como Entidades Primer Nivel](#)
- [Isomorfismo Curry-Howard](#)
- [Clausuras](#)
- [Currificación](#)
- [Aplicación Parcial](#)
- [Continuaciones](#)
- [Evaluación Perezosa](#)
- [Transparencia Referencial](#)
- [Pattern Matching](#)
- [Funciones de Orden Superior](#)
- [Listas por comprensión](#)
- [LINQ](#) (Language INtegrated Query)

# Paradigma Funcional

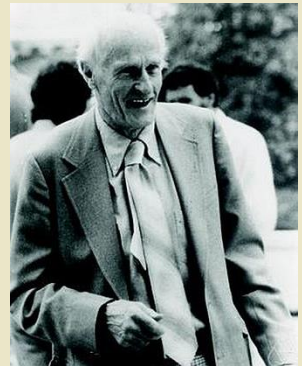
- Paradigma **declarativo** basado en la utilización de **funciones** que manejan datos inmutables
  - Los datos nunca se modifican
  - En lugar de cambiar un dato, se llama a una función que devuelve el dato modificado sin modificar el original
- Un **programa** se define mediante un **conjunto de funciones invocándose entre sí**
- Las funciones no generan **efectos (co)laterales** (secundarios):
  - el valor de una expresión depende exclusivamente de los valores de los parámetros
  - devolviendo siempre el mismo valor en función de éstos (paradigma funcional puro)

# Cálculo Lambda

- El origen de la programación funcional se remonta al **cálculo lambda** definido por *Alonzo Church* y *Stephen Kleene* en los años 30
- El cálculo lambda ( $\lambda$ -calculus) es un sistema formal basado en la **definición de funciones** (abstracción) y su **aplicación** (invocación)
  - Hace uso exhaustivo de la recursión
- El cálculo lambda se considera como el **lenguaje más pequeño universal de computación**
  - Es **universal** porque cualquier función computable puede ser expresada y evaluada usando este formalismo (lenguaje Turing-completo, tesis de Church-Turing)
- Es utilizado por investigadores y diseñadores de lenguajes para describir nuevos elementos de programación



Alonzo Church  
(1903-1995)



Stephen C. Kleene  
(1909-1994)

# Expresiones Lambda

- En el cálculo lambda, una **expresión lambda** (o término) se define como
  - Una **abstracción** lambda  $\lambda x.M$  ( $M, N, M_1, M_2...$ )  
donde  $x$  es una variable ( $x, y, z, x_1, x_2...$ ) –parámetro–  
y  $M$  es una expresión lambda –cuerpo de la función
  - Una **aplicación**  $M N$   
donde  $M$  y  $N$  son expresiones lambda
- Ejemplos de **abstracciones** (funciones)
  - La función identidad  $f(x) = x$  puede representarse como la expresión  $\lambda x.x$
  - La función doble  $g(x) = x+x$  puede representarse con la expresión  $\lambda x.x+x$

Nota:  $x+x$  no es realmente una expresión lambda, pero puede representarse con una expresión lambda (más larga, lo obviamos para ser más concisos)

# Aplicación (reducción- $\beta$ )

- La **aplicación** de una función representa su **invocación**

- La aplicación se define del siguiente modo

$$(\lambda x.M)N \rightarrow M[x:=N] \quad (\text{o } M[N/x])$$

Siendo  $x$  una variable y

$M$  y  $N$  expresiones lambda

Y  $M[x:=N]$  (o  $M[N/x]$ ) representa  $M$  donde todas las apariciones de  $x$  son sustituidas por  $N$  (sustitución)

- Esta **sustitución** (reducción de expresiones o términos) se denomina **reducción- $\beta$**  ( $\beta$ -reduction)
- Ejemplos de aplicación
  - $(\lambda x.x+x)3 \rightarrow 3+3$
  - $(\lambda x.x) \lambda y.y*2 \rightarrow \lambda y.y*2$

# Teorema de Church-Rosser

- En algunos términos lambda se pueden aplicar múltiples reducciones  $\beta$ 
  - $(\lambda x.x) (\lambda y.y*2) 3$
- El **teorema de Church-Rosser** establece que el orden en el que se hagan estas reducciones no afecta al resultado final:
  - $(\lambda x.x) (\lambda y.y*2) 3 \rightarrow (\lambda y.y*2) 3 \rightarrow 3*2$
  - $(\lambda x.x) (\lambda y.y*2) 3 \rightarrow (\lambda x.x) (3*2) \rightarrow 3*2$
- Por tanto, los paréntesis **se usan normalmente para delimitar términos** lambda, no indican precedencias
  - $(\lambda x.x) (\lambda y.y)$  es una aplicación de funciones, que se reduce al evaluarse a  $\lambda y.y$
  - $\lambda x.x \lambda y.y$  es una sola abstracción (función)

<http://www.cburch.com/proj/lambda/> (usa \ para representar  $\lambda$  y sintaxis prefija para operadores)



# Variables Libres y Ligadas

- En una abstracción  $\lambda x.xy$  se dice que
  - La variable  $x$  está **ligada** (*bound*)
  - La variable  $y$  es **libre** (*free*)
- En una sustitución, sólo se sustituyen las variables libres  
 $(\lambda x.x(\lambda x.2+x)y)M \rightarrow \mathbf{x}(\lambda x.2+x)y[x:=M] = M(\lambda x.2+x)y$ 
  - La segunda  $x$  no se sustituye (está ligada) ya que representa una variable distinta (aunque tiene el mismo nombre)
- Para evitar estos conflictos de nombres se creó la **conversión- $\alpha$**  ( *$\alpha$ -conversion*)  
*Todas las apariciones de una variable **ligada** en una misma abstracción se pueden renombrar a una **nueva** variable*  
 $(\lambda x.x(\lambda x.2+x)y)M \equiv (\lambda x.x(\lambda z.2+z)y)M \rightarrow$   
 $\mathbf{x}(\lambda z.2+z)y[x:=M] = M(\lambda z.2+z)y \equiv M(\lambda \mathbf{x}.2+\mathbf{x})y$

# Conversión- $\alpha$

- En cuales de las siguientes transformaciones el término resultante es  **$\alpha$ -equivalente?**

$\lambda x.xy$

$\lambda z.zy$

$\lambda x.2+x$

$\lambda y.2+y$

$xy(\lambda x.2+x)$

$xy(\lambda z.2+z)$

$\lambda x.xy$

$\lambda y.yy$

# Conversión- $\alpha$

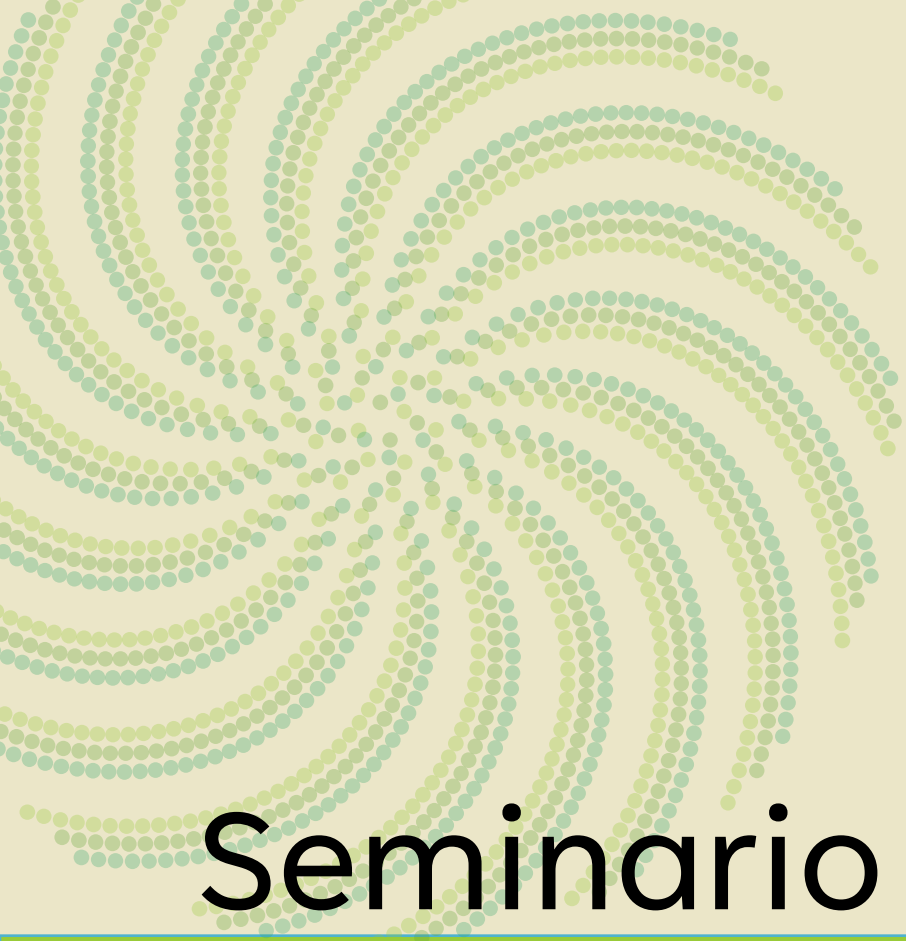
- Gracias a la conversión- $\alpha$  podemos aplicar funciones a sí mismas
- **Ejemplo 1.** La identidad de la función identidad es ella misma:  
$$(\lambda x . x) (\lambda x . x) \equiv (\lambda x . x) (\lambda y . y) \rightarrow \lambda y . y \equiv \lambda x . x$$

Aplicad conversiones- $\alpha$ - para evitar que diferentes variables ligadas tengan el mismo nombre **antes** de aplicar una función
- **Ejemplo 2.** Definimos la función *doble*  $(\lambda x . x + x)$  y la función de *aplicación dos veces*  $(\lambda f . (\lambda x . f (f x) ) )$  y calculamos el *doble del doble de n*

# Conversión- $\alpha$

- Por convenio se hace una evaluación de expresiones lambda de izquierda a derecha

$$\begin{array}{l}
 \text{Conversión-}\alpha \text{ en 2ª expresión (x/y)} \\
 \hline
 (\lambda f. \lambda x. f(fx)) (\lambda \underline{x}. \underline{x} + \underline{x}) n \equiv (\lambda \underline{f}. \lambda x. \underline{f}(\underline{f}x)) (\lambda \underline{y}. \underline{y} + \underline{y}) n \rightarrow \\
 \text{reducción-}\beta \text{ [f:=}(\lambda y. y+y)\text{]} \quad \text{Conversión-}\alpha \text{ en 1ª expresión (y/z)} \\
 \hline
 (\lambda x. (\lambda \underline{y}. \underline{y} + \underline{y}) ((\lambda \underline{y}. \underline{y} + \underline{y}) x)) n \equiv (\lambda \underline{x}. (\lambda \underline{y}. \underline{y} + \underline{y}) ((\lambda \underline{z}. \underline{z} + \underline{z}) \underline{x})) \underline{n} \\
 \text{reducción-}\beta \text{ [x:=n]} \quad \text{reducción-}\beta \text{ [y:=}(\lambda \underline{z}. \underline{z} + \underline{z})n\text{]} \\
 \hline
 \rightarrow (\lambda \underline{y}. \underline{y} + \underline{y}) ((\lambda \underline{z}. \underline{z} + \underline{z}) \underline{n}) \rightarrow ((\lambda \underline{z}. \underline{z} + \underline{z}) n) + ((\lambda \underline{z}. \underline{z} + \underline{z}) n) \equiv \\
 \text{Conversión-}\alpha \text{ en 2ª expresión (z/x)} \quad \text{reducción-}\beta \text{ [z:=n]} \\
 \hline
 ((\lambda \underline{z}. \underline{z} + \underline{z}) \underline{n}) + ((\lambda \underline{x}. \underline{x} + \underline{x}) n) \rightarrow (\underline{n} + \underline{n}) + ((\lambda \underline{x}. \underline{x} + \underline{x}) n) \\
 \text{reducción-}\beta \text{ [x:=n]} \\
 \hline
 \rightarrow (\underline{n} + \underline{n}) + (\underline{n} + \underline{n})
 \end{array}$$

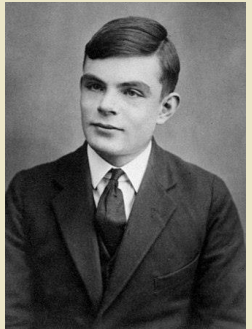


# Seminario 3

Cálculo lambda

# El Problema de la Parada

- **Problema de la Parada** (*Halting Problem*): Dada la especificación de un programa, demostrar si éste finalizará su ejecución o tendrá una ejecución infinita
- En 1936 Alan Turing probó que es **imposible** describir un **algoritmo** que resuelva el problema de la parada para todos los posibles programas y entradas de un lenguaje Turing completo  $\Rightarrow$  problema **no decidable**
  - Un lenguaje Turing completo es aquél capaz de computar lo mismo que la máquina de Turing (Lenguajes Recursivamente Enumerables o de Tipo 0)
- Hay lenguajes como Coq o Agda que permiten asegurar que un programa termina
- En cálculo lambda se pueden escribir programas que no terminan (que *divergen*)  
 $(\lambda x. xx)(\lambda x. xx) \rightarrow (\lambda x. xx)(\lambda x. xx)$



Alan Turing  
(1912-1954)

# Lógica como la base del Software

- La **lógica** es la **base formal** del desarrollo de **software**

$$\frac{\text{Lógica}}{\text{Ingeniería del Software}} = \frac{\text{Cálculo}}{\text{Ingeniería Civil}}$$

- La **lógica** es la ciencia formal que se dedica al estudio de las formas válidas de inferencia y su demostración
  - Estudio de los métodos y los principios utilizados para distinguir el razonamiento correcto del incorrecto
- Una **demostración** (*proof*) es una prueba convincente de que una **proposición** (**teorema**) es necesariamente cierta
  - Las demostraciones suelen realizarse siguiendo un **razonamiento deductivo**
- Las distintas demostraciones de que una proposición (teorema) es cierta se denominan **evidencias** (*evidence*)

# Curry-Howard

- El isomorfismo o correspondencia de **Curry-Howard** establece una **relación directa entre programas software y demostraciones matemáticas**
  - Estableciendo una correspondencia entre la lógica y la computación
  - En cálculo lambda, esta correspondencia es directa
- Esto implica que
  1. Existe una correspondencia entre **tipos** y **proposiciones** (teoremas o fórmulas)

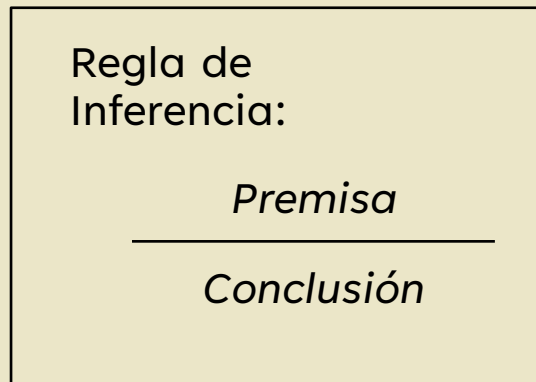
Lógica	Equivale al	Tipo
$\supset (\rightarrow)$		Función
$\wedge$		Tipo producto (Tupla o Registro)
$\vee$		Tipo suma (Unión)
true		Tipo Top (Object)
false		Tipo Bottom (void)
$\forall$		Genericidad



# Curry-Howard, Tipos

2. Existe una correspondencia entre **programas** y **evidencias que demuestran la proposición** (teoremas o fórmulas) **descrita por su tipo**

**Ejemplo:**  $A \wedge B \supset B \wedge A$  es una proposición (fórmula o teorema) cierta que puede demostrarse mediante deducción natural del siguiente modo:



$$\frac{\frac{A \wedge B}{B} \quad \frac{A \wedge B}{A}}{B \wedge A} \\ A \wedge B \supset B \wedge A$$

# Curry-Howard, Tipos

Continuación del Ejemplo: La **proposición**

$A \wedge B \supset B \wedge A$  se corresponde con el **tipo**  $A \times B \rightarrow B \times A$  y la demostración anterior se corresponde con la inferencia del tipo del programa:

$x : T$

Significa “*x tiene el tipo T*”

$x : A \times B$ <hr style="width: 100%;"/> $\text{second } x : B$  <hr style="width: 100%;"/>	$x : A \times B$ <hr style="width: 100%;"/> $\text{first } x : A$  <hr style="width: 100%;"/>
$\langle \text{second } x, \text{first } x \rangle : B \times A$	
<hr style="width: 100%;"/> $\lambda x:A \times B. \langle \text{second } x, \text{first } x \rangle : A \times B \rightarrow B \times A$	

# Correspondencia Curry-Howard

- El proceso deductivo ha sido igual para :
  - 1) Demostrar la proposición (teorema)  $A \wedge B \supset B \wedge A$
  - 2) Encontrar un programa cuyo tipo es  $A \times B \rightarrow B \times A$

1)

$$\begin{array}{c} \frac{A \wedge B}{B} \qquad \frac{A \wedge B}{A} \\ \hline B \wedge A \\ \hline A \wedge B \supset B \wedge A \end{array}$$

2)

$$\begin{array}{c} \frac{x : A \times B}{\text{second } x : B} \qquad \frac{x : A \times B}{\text{first } x : A} \\ \hline \langle \text{second } x, \text{first } x \rangle : B \times A \\ \hline \lambda x : A \times B. \langle \text{second } x, \text{first } x \rangle : A \times B \rightarrow B \times A \end{array}$$

# Demostración de Propiedades

- El resultado es que la **lógica** se puede utilizar como **mecanismo formal** para la **demostración de propiedades del software**

$$\frac{\text{Lógica}}{\text{Ingeniería del Software}} = \frac{\text{Cálculo}}{\text{Ingeniería Civil}}$$

- El mejor ejemplo es la **verificación de programas**: demostrar que un programa es correcto conforme a una especificación
  - Por **ejemplo**, un **algoritmo de ordenación** será correcto cuando se demuestre
    - Que, tras su invocación, todos los elementos están ordenados
    - Para cualquier entrada pasada al algoritmo (hay infinitas)

# Lógica y Paradigma Funcional

- El **paradigma funcional** es el **más utilizado** para realizar **demostraciones sobre programas** porque
  1. Toda computación se puede expresar en cálculo lambda (es universal)
  2. Hay una traducción directa con la lógica (isomorfismo Curry-Howard)
- Existen **asistentes de demostradores** que permiten demostrar propiedades (y corrección) de programas (Coq, Isabelle, Agda, Twelf...)
  - Se basan en lenguajes funcionales (ML y Haskell)
  - Añaden un lenguaje para realizar demostraciones mediante deducción natural
  - Permiten la extracción de programas: generar el código en distintos lenguajes (OCaml, Haskell, Scala...) una vez realizada la demostración

# Funciones, Entidades Primer Nivel

*NO: entidades de primer orden*

- El paradigma funcional identifica las funciones como **entidades de primer nivel**, igual que el resto de valores (Ejemplo: objetos): **Funciones de primera clase**
- Esto significa que **las funciones son un tipo más**, pudiéndose instanciar variables de tipo función para, por ejemplo:
  - Asignarlas en estructuras de datos (objetos, registros, árboles, tuplas...)
  - Pasarlas como parámetros a otras funciones
  - Retornarlas como valores de otras funciones
- Una función se dice que es una **función de orden superior** (*higher-order*) si:
  - O bien recibe alguna función como parámetro
  - O bien retorna una función como resultado

# Funciones de Orden Superior

- La función **doble aplicación**, que vimos anteriormente, es una función de orden superior  
 $\lambda f. (\lambda x. f(fx))$
- Recibe una función como parámetro (**f**) y, dada otra expresión (**x**), la utiliza para pasársela a la función (**fx**) y, su resultado, se lo vuelve a pasar a la función (**f(fx)**)

$(\lambda f. (\lambda x. f(fx))) (\lambda y. y+y) 3 \rightarrow 12$

El segundo parámetro es 3.  
En la aplicación será la  
sustitución de **x**.

El primer parámetro es la función **doble**.  
En la aplicación será la sustitución de **f**.

Función **doble aplicación**.

# Delegados

- En C# las funciones (métodos) son entidades de primer nivel gracias a los **delegados** (y expresiones lambda)
- Un delegado en C# constituye **un tipo que representa un método** de instancia o de clase (**static**)
- Las variables de tipo delegado representan un modo de referenciar un método
- Por tanto,
  - El paso de estas variables implican el paso de métodos (funciones) a otros métodos (funciones)
  - La asignación permite parametrizar estructuras de datos (objetos) y algoritmos en base a otras funcionalidades (métodos o funciones)



# Delegados

- La siguiente línea define **el tipo del delegado**

```
public delegate int Comparacion(Persona p1, Persona p2);
```

- Definiendo el tipo **Comparacion** como un método que recibe dos **Persona** y devuelve un **int**
- Es posible declarar el siguiente método **OrdenarPersonas** independiente del criterio de ordenación

```
static public void OrdenarPersonas(  
    Persona[] vector, Comparacion comparacion) {
```

- Y hacer uso del delegado del siguiente modo

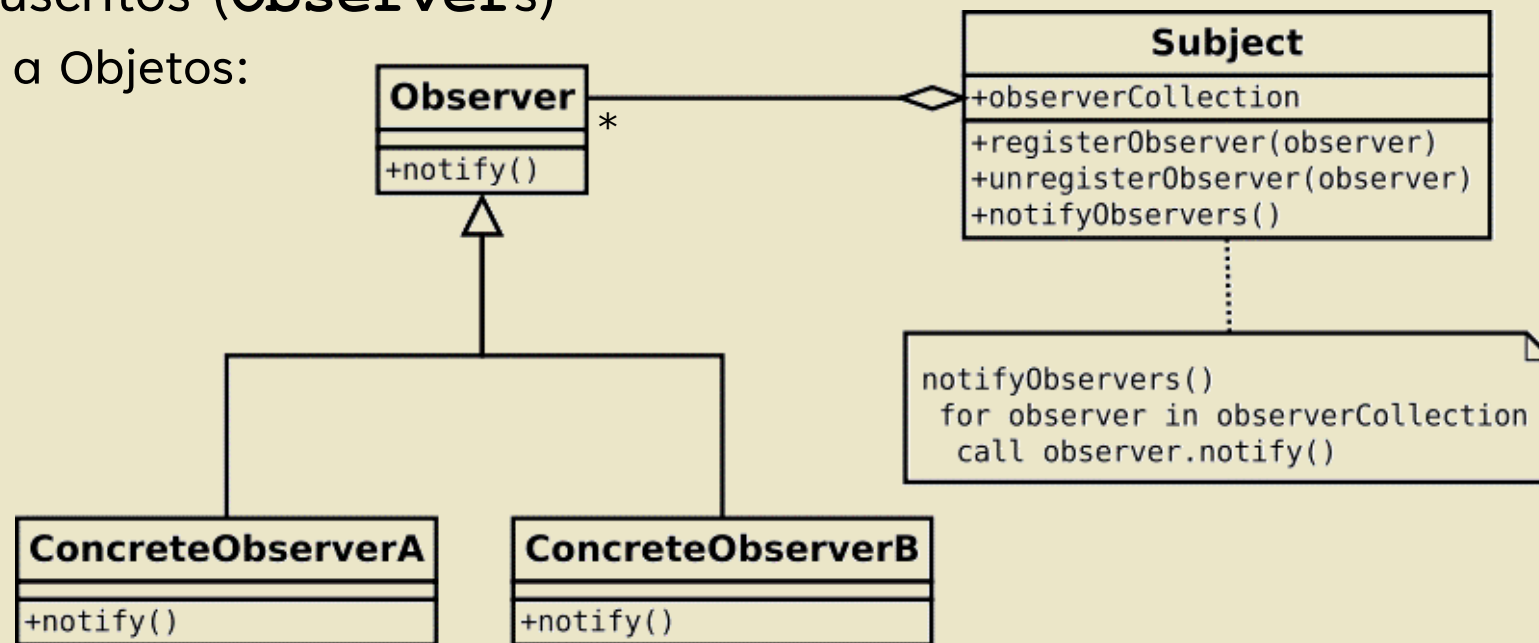
```
if (comparacion(vector[i], vector[j]) > 0) {  
    Persona aux = vector[i];  
    vector[i] = vector[j];  
    vector[j] = aux;  
}
```

Consulta el código en:

[\*delegates/delegates\*](#)

# Patrón *Observer*

- En .Net, un uso muy extendido de los delegados es el basado en el patrón de diseño *Observer*
  - Los suscriptores (*observer*) se registran y desregistran en un asunto (*subject*)
  - Cuando sucede un evento, el asunto (**Subject**) notifica de tal evento a los elementos suscritos (**Observers**)
- Diseño Orientado a Objetos:



# Patrón *Observer*

- En .Net una instancia de **un delegado** puede en realidad coleccionar un conjunto de métodos
  - Con el operador **+=** se añade un método(registro)
  - Con el operador **-=** se elimina un método (desregistro)
  - Con el operador **=** se asigna un único método
- Cuando se invoca a un delegado, se producirá una invocación a todos los subscriptores registrados en ese delegado
- Por tanto es posible utilizar delegados para implementar el patrón **Observer**

# Patrón *Observer*

```
class Boton {  
    public delegate void BotonClick();  
    private BotonClick metodosAInvocar;  
    public void AñadirDelegado(BotonClick metodo) {  
        metodosAInvocar += metodo;  
    }  
    public void EliminarDelegado(BotonClick metodo) {  
        metodosAInvocar -= metodo;  
    }  
    public void Click() {  
        metodosAInvocar();  
    }  
}
```

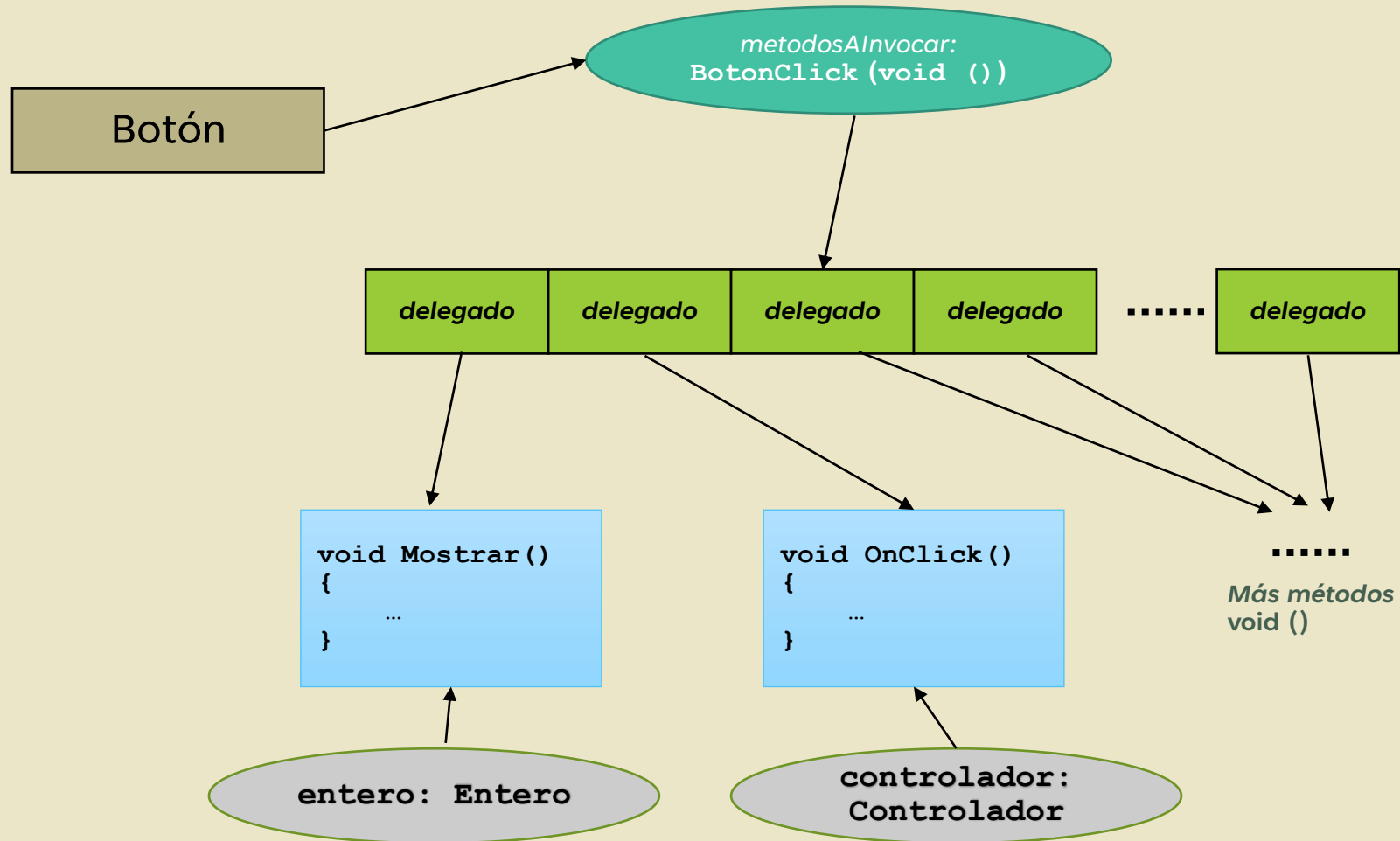
# Patrón *Observer*

```
static void Main() {  
    Boton boton = new Boton();  
  
    Entero entero = new Entero(3);  
    Controlador controlador = new Controlador();  
  
    boton.AñadirDelegado(entero.Mostrar);  
    boton.AñadirDelegado(controlador.OnClick);  
  
    boton.Click();  
}
```

Consulta el código en:

*delegates/observer*

# Patrón Observer



# Tipos delegados predefinidos

Consulta el código en:

*delegates/predefined*

- La plataforma .Net tiene un conjunto de delegados predefinidos
  - Hacen uso de la potencia de la genericidad
- Los más utilizados son **Func**, **Action** y **Predicate**
  - **Func<T>**: Método sin parámetros que retorna un **T**
  - **Func<T1, T2>**: Método con un parámetro **T1** que retorna un **T2**
  - ...
  - **Action**: Método sin parámetros ni retorno
  - **Action<T>**: Método con un parámetro **T** sin retorno
  - ...
  - **Predicate<T>**: Método que retorna un **bool** y recibe un **T**
- Estos tres son muy utilizados (**¡no defines nuevos!**)

# Tipos delegados predefinidos

```
class Test1 {
    public float Abs(float param) { ... }
    public float Cos(float param) { ... } }
class Test2 {
    public float Sin(float param) { ... }
    public bool isPositive(float param) { ... }
    public static void Print(float param) { ... } }
public static void Main(string[] args) {
    Test1 t1 = new Test1();
    Test2 t2 = new Test2();
    Func<float, float> method1 = t1.Abs;
    float result = method1(3.4f); //Invocamos Abs de la instancia t1

    method1 = t1.Cos;
    result = method1(3.4f); //Invocamos Cos de la instancia t1, usando la misma referencia

    method1 = t2.Sin;
    result = method1(3.4f); //Invocamos Sin de la instancia t2, usando la misma referencia

    //Func<float, bool> method2 = t1.Abs; // Error
    Func<float, bool> method2 = t2.isPositive; // Ok
    Predicate<float> method3 = t2.isPositive; // También válido
    bool bResult = method2(3.4f);

    Action<float> method4 = Test2.Print;
    method4(5.6f); //Invocamos Print de la clase Test2 }
```



# Delegados Anónimos

- El tener que implementar un método para pasarlo como parámetro posteriormente es tedioso
- En la programación funcional es común poder escribir la función en el momento de pasar ésta
- La primera aproximación que ofreció C# para ello fueron los **delegados anónimos**
  - Sintaxis para definir una variable delegado indicando sus parámetros y su cuerpo (código)

```
Persona[] personas = ListadoPersonas.CrearPersonasAleatorias();
```

```
Persona[] mayoresEdad = Array.FindAll(personas,  
    delegate(Persona p) { return p.Edad >= 18; }  
);
```

- Sigue siendo una **sintaxis** un tanto **prolija**

Consulta el código en:

*[delegates/anonymous.methods](#)*

# Expresiones Lambda

- Las expresiones (funciones) lambda de C# permiten **escribir el cuerpo de funciones completas como expresiones** (siguiendo la aproximación del  $\lambda$ -cálculo)
  - Son una mejora de los delegados anónimos
- **Sintaxis**
  - Se especifican los parámetros separados por comas (si los hay)
    - Opcionalmente se pueden anteponer los tipos
    - Si hay más de un parámetro se deben utilizar paréntesis
    - Si hay cero parámetros, se indica con `()`
  - El símbolo `=>` indica la separación de los parámetros y el cuerpo de la función
  - Si el cuerpo tiene varias sentencias se separan con `;`
  - En el cuerpo se utiliza **`return`** para devolver valores
  - Si el cuerpo es una única sentencia, no es necesario escribir **`return`** ni llaves

# Expresiones Lambda

```
class Program {  
    public static bool MayorDe18 (Persona p)  
    { return p.Edad >= 18; }  
    // ...  
}
```

```
Persona[] personas =  
    ListadoPersonas.CrearPersonasAleatorias();  
Persona[] mayoresEdad;  
mayoresEdad = Array.FindAll(personas,  
    Program.MayorDe18);
```

```
mayoresEdad = Array.FindAll(personas,  
    delegate (Persona p) { return p.Edad >= 18; }  
);
```

*Delegado anónimo*

```
mayoresEdad = Array.FindAll(personas,  
    (Persona p) => { return p.Edad >= 18; });
```

```
mayoresEdad = Array.FindAll(personas,  
    (p) => { return p.Edad >= 18; });
```

```
mayoresEdad = Array.FindAll(personas,  
    p => { return p.Edad >= 18; });
```

```
mayoresEdad = Array.FindAll(personas,  
    p => p.Edad >= 18  
);
```

*Expresión Lambda*

# Expresiones Lambda

- Los tipos de las expresiones lambda promocionan a los tipos de delegados predefinidos en .Net: **Func**, **Predicate** y **Action**
- A partir de Java 8 **se incorporaron las expresiones lambda a Java**, siguiendo una aproximación similar (ver presentación “New Functional Features of Java”)

```
Func<Func<int, int>, int, int> dobleAplicacion =  
    (f, n) => f(f(n));
```

```
Console.WriteLine(dobleAplicacion( n => n+n, 3);
```

```
Persona[] personas = ListadoPersonas.  
    CrearPersonasAleatorias();
```

```
Persona[] mayoresEdad = Array.FindAll(  
    personas, persona => persona.Edad >= 18);
```

Consulta el código en:

*[delegates/lambda](#)*

# Expresiones Lambda

- Más ejemplos:

```
IDictionary <string, Func<int, int, int>> calculadoraFuncional
    = new Dictionary<string, Func<int, int, int>>();
calculadoraFuncional["add"] = (op1, op2) => op1 + op2;
calculadoraFuncional["sub"] = (op1, op2) => op1 - op2;
calculadoraFuncional["mul"] = (op1, op2) => op1 * op2;
calculadoraFuncional["div"] = (op1, op2) => op1 / op2;
```

```
calculadoraFuncional["add"](3, 4); // 7
```

```
IList<Predicate<string>> condiciones = new List<Predicate<string>>();
condiciones.Add(s => s.Length < 5);
condiciones.Add(s => !s.StartsWith("F"));
condiciones.Add(s => s.EndsWith("i"));
```

```
string str = "Hi";
foreach (var cond in condiciones) { if (!cond(str)) {...} }
```



# Seminario 4

Funciones de primera clase

# Bucles y Recursividad

- La programación funcional pura **no posee** el concepto de **iteración** (bucles)
  - En su lugar, hace uso de la **recursividad** (recursión)
- En C# (y en la mayor parte de lenguajes) las funciones tienen nombres, por lo que es sencillo hacer una llamada recursiva

```
static ulong fact(ulong n) {  
    return n==0 || n==1 ? 1 : n*fact(n-1);  
}
```

- Pero en el cálculo lambda las funciones no poseen nombres, dificultando así la recursividad
- Las funciones recursivas se han de expresar como funciones de orden superior que se reciben a sí mismas

$\lambda x. \text{ if } x=0 \text{ or } x=1 \text{ then } 1 \text{ else } x*?(x-1)$

*Los operadores if-then-else, =, \* y - también pueden codificarse en cálculo lambda*

- ¿Cómo resolver este problema?

# Iteración y Recursividad

- Las funciones recursivas deben implementarse como funciones de orden superior, pasándose a si mismas como parámetro  
 $\lambda f. \lambda x. \text{ if } x=0 \text{ or } x=1 \text{ then } 1 \text{ else } x * f(x-1)$
- La cuestión es: **¿cómo puedo pasar la propia función como parámetro?**



# Combinador de Punto Fijo

- El **combinador de punto fijo** (normalmente denominado `fix`) es una función de orden superior que cumple lo siguiente

$$\text{fix } f \rightarrow f \ (\text{fix } f)$$

- Es decir, que al aplicar `fix` a `f (fix f)`
  1. Se retorna `f`
  2. Pasando una nueva invocación a `fix f` como primer parámetro

# Ejemplo de invocación

- Recordemos...

a)  $\text{fact} \equiv \lambda f. \lambda x. \text{if } x=0 \text{ or } x=1 \text{ then } 1 \text{ else } x * f(x-1)$

b)  $\text{fix fact} \rightarrow \text{fact } (\text{fix fact})$

- Evaluemos pues **fix fact 3** (las expresiones evaluadas aparecen subrayadas)

1. `fix fact 3` b)  $\rightarrow$

2. `fact (fix fact) 3` a)  $\rightarrow$

3. `3 * (fix fact) 2` b)  $\rightarrow$

4. `3 * fact (fix fact) 2` a)  $\rightarrow$

5. `3 * 2 * (fix fact) 1` b)  $\rightarrow$

6. `3 * 2 * fact (fix fact) 1` a)  $\rightarrow$

7. `3 * 2 * 1`

# Combinador Y

- Una de las funciones **fix** más conocidas es el **combinador Y**  
 $\text{fix} \equiv \lambda f. (\lambda x. f (xx)) (\lambda x. f (xx))$

$\text{fix fact} \equiv$

$\lambda f. (\lambda x. f (xx)) (\lambda x. f (xx)) \text{ fact} \rightarrow$

$(\lambda x. \text{fact} (xx)) (\lambda x. \text{fact} (xx)) \equiv_{\alpha}$

$(\lambda y. \text{fact} (yy)) (\lambda x. \text{fact} (xx)) \rightarrow$

$\text{fact} ( (\lambda x. \text{fact} (xx)) (\lambda x. \text{fact} (xx)) ) \equiv$

$\text{fact} (\text{fix fact})$

# Clausura

- Una **clausura** (cierre o *closure*) es una función de primer nivel junto con su ámbito: una tabla que guarda las referencias a sus variables libres
- En cálculo lambda  $\lambda y. \dots x$  es una clausura

$\lambda x. \dots x \dots \quad \lambda y. \dots x$

$x$  es una variable libre en  $\lambda y. \dots x$ , pero guarda una **referencia** (no una copia) a la  $x$  externa de  $\lambda x. \dots x \dots$

- En C#

```
int valor = 1;
Func<int> dobleDeValor = () => valor * 2;
dobleDeValor(); // 2
valor = 7;
dobleDeValor(); // 14
```

# Clausura

- Las variables libres de una clausura representan **estado**
  - Este estado **puede**, además, **estar oculto** cuando el ámbito de la variable finaliza
- Por tanto, pueden representar **objetos**

```
static Func<int> RetornarContador() {  
    int contador = 0;  
    return () => ++contador;  
}
```

# Clausura

- Las clausuras también pueden representar **estructuras de control**

```
void BucleWhile(Func<bool> condicion, Action cuerpo) {  
    if (condicion()) {  
        cuerpo();  
        BucleWhile(condicion, cuerpo);  
    }  
}  
...  
int i = 0;  
BucleWhile(    () => i < 10,  
            () => { Console.Write(i); i++; }  
            );
```

Consulta el código en:

*[closures/closures](#)*

# Clausura

- Puesto que las clausuras permiten
  1. Representar estructuras de control iterativas
  2. Objetos y ocultación de información
- Estamos viendo cómo se pueden traducir estos mecanismos imperativos a elementos equivalentes del paradigma funcional
  - No existiendo pérdida de expresividad
  - Sino **un modo distinto** (paradigma) **de expresar las abstracciones**

# Clausura - Ejemplo

```
long limit = 10;
int num = 0;
long resultAction1 = 1, resultAction2 = 2; //Variables para guardar el resultado de las Actions
del while
double resultAction3 = 20000; // Podríamos meter los resultados en arrays o listas (si son del
mismo tipo o no)

Func<bool> test = () => num < limit; //Clausura con la condición del while
List<Action> whileActions = new List<Action>();
//Actions para usar como cuerpo del while. Son todas clausuras y gracias a ello no necesitamos
pasarles parámetros
whileActions.Add(() => {resultAction1 *= 2; num++;});
whileActions.Add(() => {resultAction2 += 2; num++;});
whileActions.Add(() => {resultAction3 = Math.Cos(resultAction3); num++;});

//Ejecutamos el while con distintos cuerpos :)
foreach (var action in whileActions)
{
    WhileLoop(test, action); //En cada iteración ejecutamos un código distinto
    /* Dado que las clausuras comparten una referencia a num, debe re-inicializarse en cada
    iteración. Si no, ¡las iteraciones 2ª y 3ª no se ejecutarán, al ser la condición falsa
    (num==limit)!*/
    num = 0;
}
//Imprime 1024, 22, 0,736924884186011
Console.WriteLine(resultAction1 + ", " + resultAction2 + ", " + resultAction3);
```



# Curificación

- El cálculo lambda es un lenguaje de programación **universal**: cualquier función computable puede representarse en él
  - Números y operaciones enteras y booleanas (*Church encoding*)
- No obstante, la definición de abstracción sólo define un único parámetro para las funciones lambda
- La **curificación** (*currying*) es la técnica para transformar una función de varios parámetros en una función que recibe un único parámetro
  - La función recibe un parámetro y retorna otra función que se puede llamar con el segundo parámetro
  - Esto puede repetirse para todos los parámetros de la función original
  - La invocación se convierte en llamadas a funciones encadenadas (Ej.  $f(1)(2)(3)$  vs.  $f(1, 2, 3)$ )
- El nombre es en honor al matemático Haskell Curry (aunque lo inventó Moses Schönfinkel)

# Curricación

- En cálculo lambda

$$\lambda x. \lambda y. x+y \equiv \lambda xy. x+y$$

- En C#

```
static Func<int, int> SumaCurricada(int a) {  
    return b => a + b;  
}  
static Predicate<int> MayorQue(int a) {  
    return b => a > b;  
}  
...  
SumaCurricada(3)(1)  
Predicate<int> esNegativo = MayorQue(0);
```

Consulta el código en:

[closures/currying](#)

- Pregunta: ¿qué característica se ha usado para implementar la currificación?
- Su principal beneficio es la **aplicación parcial** (siguientes transparencias)

# Aplicación Parcial

- Cuando las **funciones** están “**currificadas**” es posible realizar su aplicación (invocación) parcial
  - En lenguajes como Haskell o ML, todas las funciones están currificadas
- Al **aplicación parcial** consiste en pasar un número menor de parámetros en la invocación de la función
  - El resultado es otra función **con un número menor en su aridad** (número de parámetros)
- Esto le da gran potencia al lenguaje, especialmente cuando los **operadores son funciones** (Haskell)
- En cálculo lambda
$$\begin{aligned}(\lambda x. \lambda y. x+y) 1 &\equiv \lambda y. 1+y \\ (\text{sumar}) 1 &\equiv \text{incrementar}\end{aligned}$$

# Aplicación Parcial en C#

- En C# la **sintaxis** para aplicación parcial es **prolija**
  1. Las funciones no están currificadas por omisión
  2. Los operadores no son funciones
- Esto hace que este mecanismo en C# no sea tan potente como en lenguajes como Haskell (ofrece ambos)
- Un ejemplo de su utilidad:

```
static Func<int, int> Suma(int a) {  
    return b => a + b; }  
static Predicate<int> MayorQue(int a) {  
    return b => a > b; }  
static Predicate<int> SegundoParametro(  
    Func<int, Predicate<int>> predicado, int parametro) {  
    return a => predicado(a)(parametro); }
```

# Aplicación Parcial en C#

```
static int[] Map(int[] origen, Func<int, int> funcion) {
    int[] destino = new int[origen.Length];
    for (int i = 0; i < origen.Length; i++)
        destino[i] = funcion(origen[i]);
    return destino;
}

static IEnumerable<int> Filtrar(IEnumerable<int> origen, Func<int, bool> predicado) {
    IList<int> destino = new List<int>();
    foreach (int elemento in origen)
        if (predicado(elemento)) destino.Add(elemento);
    return destino;
}

static void Main() {
    int[] enteros = new int[10];
    int[] inc = Map(enteros, Suma(1));
    var positivos = Filtrar(enteros, SegundoParametro(MayorQue, 0));
}
```

(1+) en Haskell

(>0) en Haskell

Consulta el código en:

[closures/partial.application](https://github.com/franciscoortin/closures/partial.application)

# Partial Application in C#: Example

- La aplicación parcial se puede utilizar para escribir menos código
- En lugar de crear multiples elementos con una parte común, podemos definir esa parte común una vez y reutilizarla
  - Definimos la parte comun gracias a la aplicación parcial
  - Lo guardamos en un delegado (función parcialmente-aplicada)
  - Invocamos la función parcialmente-aplicada con el resto de parametros (parte variable)

```
Email BuildCombinedEmail(string signature,
DateTime date, string text, string origin,
string greeting, string destination) {
    return new Email(signature,
        date, text, origin, greeting,
        destination).ToString();
}
```



*Versión currificada*

```
Func<string, Func<DateTime, Func<string,
Func<string, Func<string, Func<string,
Email>>>>>>
BuildCombinedEmail() {
    return signature => date => text =>
origin => greeting => destination =>
    new Email(signature,
        date, text, origin, greeting,
        destination);
}
```

# Aplicación Parcial en C#: Ejemplo

```
//Creación "Traditional" de un email
// (función con 6 parametros)
var nonCurried = new NonCurriedCombinedEmail();
var johnLetter = nonCurried.BuildCombinedEmail(
    "Hans Gruber, Department of Dark Arts",
    DateTime.Today,
    "You are fired",
    "HansGruber@F.org",
    "Dear John McLane",
    "diehard@F.org");
```

**Llamada "Tradicional"**

```
//Creación "currificada" de un email
//(secuencia de funciones con 1 parametro).
//Builder pattern
var curried = new CurriedCombinedEmailBuilder();
var letterForJohnCurried =
    curried.BuildCombinedEmail()
        ("Hans Gruber, Department of Dark Arts")
        (DateTime.Today)
        ("You are fired")
        ("HansGruber@F.org")
        ("Dear John McLane")
        ("diehard@F.org");
```

**Llamada "currificada"**

```
string[] employees = { "John McLane", "Patricia McLane", "Angus Dumbledore" };
//Podemos pre-construir parte de un email gracias a la aplicación parcial
var preBuiltFiringLetter = curried.BuildCombinedEmail() ("Hans Gruber, Department of Dark Arts")
    (DateTime.Today) ("You are fired") ("HansGruber@F.org");
```

**Aplicación parcial**

```
//Posteriormente construimos la parte "variable" del correo
foreach (var employee in employees)
    Console.WriteLine(preBuiltFiringLetter("Dear " + employee)
        (employee.Replace(" ", "_") + "@F.org"));
```

Consulta el código en:

**[closures/curryingBuilder](#)**

# Continuaciones

- Una continuación representa el **estado de computación** en un momento de ejecución
- El estado de computación normalmente está compuesto por, al menos
  1. El **estado de la pila** de ejecución
  2. La **siguiente instrucción a ejecutar**
- Los lenguajes que ofrecen continuaciones son capaces de almacenar su estado de ejecución y recuperarlo posteriormente
  - Scheme y Ruby ofrecen continuaciones
  - C# no las ofrece de un modo directo



# Continuaciones

- Los lenguajes que ofrecen continuaciones añaden una función **call/cc** (*call with current continuation*) para obtener el estado de computación
- Lo siguiente es un ejemplo en sintaxis C# (**no válido en C#**)

```
static Func<int> continuacion = null;
static int Test() {
    int i = 0;
    // Asigna a continuación el estado actual
    // (estado dinámico en la línea actual)
    callcc(estado => continuacion = estado);
    i = i + 1;
    return i;
}
```

```
static void Main(string[] args) {
    Console.Write(Test());           // 1
    Console.Write(continuacion());   // 2
    Console.Write(continuacion());   // 3
    Func<int> otraContinuacion =
continuacion;
    // Resetea la continuacion
    Console.Write(Test());           // 1
    Console.Write(continuacion());   // 2
    // Sigue con la anterior
    Console.Write(otraContinuacion()); // 4
}
```

# Generadores

- Un **generador**
  - es una función
  - que simula la devolución de una colección de elementos
  - sin construir toda la colección
  - devolviendo un elemento cada vez que la función es invocada
- Una implementación de generadores es **mediante continuaciones**
- El hecho de no construir toda la colección hace que sea **más eficiente**
  - Requiere menos memoria
  - El invocador obtiene el primer elemento de forma inmediata
  - Sólo se generan los elementos que se usan
- Un generador es una función que se comporta como un **iterador**

# Generadores en C#

- C# implementa los generadores mediante **yield** una especie de continuaciones *ad hoc*

```
static IEnumerable<int> FibonacciInfinito() {  
    int primero = 1, segundo = 1;  
    while (true) {  
        yield return primero;  
        int suma = primero + segundo;  
        primero = segundo;  
        segundo = suma;  
    }  
}  
...  
foreach (int valor in Fibonacci.FibonacciInfinito()) {  
    Console.WriteLine("Término {0}: {1}.", i, valor);  
    if (i++ == numeroTerminos) break;  
}
```

# Generadores en C#

```
IEnumerable<int> FibonacciFinito(int términoMáximo) {  
    int primero = 1, segundo = 1, término = 1;  
    while (true) {  
        yield return primero;  
        int suma = primero + segundo;  
        primero = segundo;  
        segundo = suma;  
        if (término++ == términoMáximo)  
            yield break;  
    }  
}  
...  
foreach (int valor in Fibonacci.FibonacciFinito(10))  
    Console.Write(valor);
```

Consulta el código en:

*[continuations/generators](#)*

# Evaluación Perezosa

- La **evaluación perezosa** (*lazy*) es la técnica por la que se demora la evaluación de una expresión hasta que ésta es utilizada
  - Lo contrario es la evaluación **ansiosa** (*eager*) o **estricta** (*strict*)
- Relativo al paso de parámetros, la mayoría de lenguajes ofrecen paso ansioso
  - Haskell y Miranda ofrecen paso perezoso de argumentos
  - Ejemplo en sintaxis C# (código no válido en C#)

```
int Eager(int n) { return 0; }
int Lazy(int n) { return 0; }
int a = 1, b = 1;
Eager(a++); // a == 2 tras la invocación
Lazy(b++);  // b == 1 tras la invocación
```

# Evaluación Perezosa

- Los beneficios de la evaluación perezosa son
  1. Un menor **consumo de memoria**
  2. Un mayor **rendimiento**
  3. La posibilidad de poder crear **estructuras de datos potencialmente infinitas**
- La implementación de un procesador de lenguaje (compilador o intérprete) que soporte evaluación perezosa es más compleja

# Evaluación Perezosa en C#

*Cuidado con el Reset.  
Solo se consumen una vez*

- **C# no ofrece evaluación perezosa de un modo directo**
  - A excepción de las continuaciones implementadas en los generadores
- Puesto que la generación de elementos es perezosa, podemos generar colecciones de un número infinito de números con **yield**  
y hacer uso de ellas con los siguientes métodos extensores
  - **Skip** para desoír un conjunto de elementos de una secuencia, retornando los restantes
  - **Take** para retornar un nº concreto de elementos contiguos desde el principio de una secuencia

# Evaluación Perezosa en C#

```
static private IEnumerable<int> GeneradorLazyNumerosPrimos() {  
    int n = 1;  
    while (true) {  
        if (EsPrimo(n))  
            yield return n;  
        n++;  
    }  
}  
  
static internal IEnumerable<int> NumerosPrimosLazy(  
    int desde, int númeroDeNúmeros) {  
    return GeneradorLazyNumerosPrimos()  
        .Skip(desde).Take(númeroDeNúmeros);  
}
```

Consulta el código en:

*[continuations/lazy](#)*



# Transparencia Referencial

- Una expresión se dice que es **referencialmente transparente** si ésta se puede sustituir por su valor sin que cambie la semántica (significado) del programa
  - Lo contrario es la opacidad referencial
- Mientras que en matemáticas todas las funciones son referencialmente transparentes, en programación no
- La transparencia referencial es uno de los **pilares del paradigma funcional**
- Cuando es ofrecida por un lenguaje, se dice que es **funcional puro**: Haskell, Clean y Charity
- **Elementos** de un lenguaje que hacen que **no ofrezca transparencia referencial** son:
  1. Variables globales mutables
  2. Asignaciones destructivas (`=`, `+=`, `*=`, `++`, `--...`)
  3. Funciones impuras (E/S, *random*, *datetime...*)

# Transparencia Referencial

**Pregunta: Si un lenguaje ofrece clausuras, ¿puede ofrecer transparencia referencial?**

# Variables Globales y Asignaciones

1. Las **variables mutables fuera del ámbito** de una función hacen que no se obtenga transparencia referencial
  - Vimos en el ejemplo de clausuras cómo éstas guardaban un valor como estado
  - La clausura **RetornaContador** devolvía el número de veces que había sido invocada (depende de su “historia”)  $\Rightarrow$  No puede sustituirse por un valor

```
static Func<int> RetornarContador() {  
    int contador = 0;  
    return () => ++contador;  
}
```

2. Con las **asignaciones** sucede lo mismo
  - La evaluación de una variable depende de sus asignaciones previas

# Funciones Puras

## 3. La utilización de **funciones** que **no** son **puras**, implican **opacidad referencial**

- Una función es **pura** cuando
  1. Siempre devuelve el mismo valor ante los mismos valores de los argumentos  
No depende de un estado dinámico, ni de un dispositivo de entrada salida
  2. La evaluación de una función no genera efectos secundarios ((co)laterales)  
No actualiza variables globales, ni dispositivos de entrada y salida, ni estados dinámicos
- Ejemplos de funciones **no puras**: `DateTime::Now`, `Random::Random`, `Console::ReadLine`
- Ejemplos de funciones **puras**: `Math::Sin`, `String::Length`, `DateTime::ToString`

# Beneficios

- Los beneficios de la transparencia referencial son:
  1. Se puede **aplicar el razonamiento matemático** a los **programas** para, por ejemplo:
    - Demostrar su **corrección** (*correctness*)  $\Rightarrow$  Demostrar que hace lo descrito en una especificación (cálculo de un valor, cumplimiento de invariantes y postcondiciones...)
    - **Razonamiento** acerca de su **comportamiento**  $\Rightarrow$  Identificación de propiedades (terminación, errores de ejecución, consumo de memoria...)
  2. Se pueden realizar **transformaciones** en los **programas** para, por ejemplo:
    - **Simplificación** y **paralelización** de algoritmos
    - **Optimización** de programas  $\Rightarrow$  memoización, eliminación de subexpresiones repetidas, paralelización...

# Memoización

- A modo de ejemplo, veremos una **técnica de optimización** que puede ser aplicada sobre expresiones con **transparencia referencial**: la **memoización** (memoization)
  - Si una expresión posee transparencia referencial, ésta puede sustituirse por su valor
  - Si la expresión es una invocación a una función (pura), ésta puede sustituirse por el valor de retorno
  - La primera vez que se invoca se retorna el valor guardándolo en una caché
  - En sucesivas invocaciones, se retornará el valor de la caché sin necesidad de ejecutar la función

# Memoización

```
static class FibonacciMemoizacion {  
  
    private static IDictionary<int, int> valores =  
        new Dictionary<int, int>();  
  
    internal static int Fibonacci(int n) {  
        if (valores.Keys.Contains(n))  
            return valores[n];  
        int valor = n <= 2 ? 1 :  
            Fibonacci(n - 2) + Fibonacci(n - 1);  
        valores.Add(n, valor);  
        return valor;  
    }  
}
```

Consulta el código en:

*[continuations/memoization](#)*

## Preguntas:

¿Ventajas?

¿Inconvenientes?

# Evaluación Perezosa (Revisitada)

- Recordemos que en el **paso de parámetros perezoso**
  - se demora la evaluación de un parámetro hasta que éste sea utilizado
- Este comportamiento se puede conseguir
  - Haciendo que los **parámetros sean funciones** (siendo las funciones, por tanto, de orden superior)
  - Memoizando su **evaluación**
- Actividad Obligatoria: Analice, ejecute y comprenda este ejemplo de implementación

Consulta el código en:

*[continuations/lazy.simulated](#)*



# Pattern Matching

- **Pattern matching** (coincidencia de patrones) es el acto de comprobar si (la secuencia de) un conjunto de elementos siguen algún patrón determinado
- Los **elementos** suelen ser
  - Tipos (clases) de variables (objetos)
  - Listas
  - Cadenas de caracteres (*strings*)
  - Tuplas
  - Arrays
  - ...
- Los **patrones** suelen ser
  - Secuencias (generalmente mediante expresiones regulares)
  - Estructuras de árboles (listas bidimensionales)
- Lenguajes que ofrecen *pattern matching*: Haskell, ML, Mathematica, Prolog

# Pattern Matching de Tipos en C#

- C# **no** ofrecía *pattern matching* hasta la versión 7.0
- La única forma de ofrecer comprobación de patrones era haciendo uso de **introspección**: operador **is** o método **GetType**

```
static double AreaIs(Object figura) {  
    if (figura is Circulo)  
        return Math.PI * Math.Pow(((Circulo)figura).Radio, 2);  
    if (figura is Cuadrado)  
        return Math.Pow(((Cuadrado)figura).Lado, 2);  
    if (figura is Rectangulo) {  
        Rectangulo rectangulo = figura as Rectangulo;  
        return rectangulo.Alto * rectangulo.Ancho;  
    }  
    if (figura is Triangulo) {  
        Triangulo triangulo = figura as Triangulo;  
        return triangulo.Base * triangulo.Altura / 2;  
    }  
    throw new ArgumentException("El parámetro no es una figura");  
}
```

**NUNCA** escribais código de esta forma!!!  
**Es completamente inmantanible**

Consulta el código en:

[\*pattern.matching/types\*](#)

# Pattern Matching de Tipos

- La utilización de introspección para este escenario posee dos inconvenientes
  1. Es costosa en rendimiento en tiempo de ejecución
  2. Los errores no son detectados en tiempo de compilación (se detectan en tiempo de ejecución)
- Pregunta: En C#, lenguaje orientado a objetos,
  - ¿cuál es la técnica más apropiada para obtener los beneficios *pattern matching* de tipos (como en el ejemplo anterior)?

# Pattern Matching en F#

- F# es una implementación del lenguaje ML sobre .Net
- ML hace uso exhaustivo de *pattern matching*
- Lo siguiente es un ejemplo de implementación de la función *Fibonacci* en ML, utilizando *pattern matching* (con patrones de secuencia)

```
let rec fib n =  
    match n with  
    | 1 -> 1  
    | 2 -> 1  
    | x -> fib(x - 2) + fib(x - 1)  
;;
```

Consulta el código en:

[pattern.matching/fsharp](https://pattern.matching/fsharp)

# Pattern Matching de Tipos en F#

- F# también ofrece *pattern matching* (con patrones de árbol)

```
type Figura =  
| Circulo of double  
| Rectangulo of double * double  
| Cuadrado of double  
| Triangulo of double * double  
;;  
  
let area figura =  
    match figura with  
    | Circulo(radio) -> 3.141592 * radio * radio  
    | Rectangulo(ancho, alto) -> ancho * alto  
    | Cuadrado(lado) -> lado * lado  
    | Triangulo(labase, altura) -> labase*altura/2.0  
    ;;
```

Consulta el código en:

[pattern.matching/fsharp](https://pattern.matching/fsharp)

# Pattern Matching de Tipos en F#

- Pregunta: Identifique cómo representar las dos secciones de código en un lenguaje orientado a objetos como Java/C#

```
type Figura =  
| Circulo of double  
| Rectangulo of double * double  
| Cuadrado of double  
| Triangulo of double * double  
;;  
  
let area figura =  
    match figura with  
    | Circulo(radio) -> 3.141592 * radio * radio  
    | Rectangulo(ancho, alto) -> ancho * alto  
    | Cuadrado(lado) -> lado * lado  
    | Triangulo(labase, altura) -> labase*altura/2.0  
    ;;
```

Sección 1

Sección 2

# Patrones con condiciones

- En ML (F#) es posible poner **condiciones** a los patrones
  - Para ello se utiliza la palabra reservada **when** y el carácter comodín **\_**

```
let regular figura =  
    match figura with  
    // _ es un comodín: implica que puede valer cualquier cosa  
    | Cuadrado (_) -> true  
    // cierto si el ancho y el alto son iguales  
    | Rectangulo(ancho, alto) when ancho = alto -> true  
    // falso para el resto de rectángulos  
    | Rectangulo(_, _) -> false  
    // cierto si es isósceles  
    | Triangulo(labase, altura) -> altura =  
        labase*System.Math.Sqrt(3.0)/2.0  
    | _ -> false // falso para el resto de figuras (el círculo)  
;;
```

- En Haskell, los patrones condicionales se llaman *Guards*

Consulta el código en:

[pattern.matching/fsharp](https://pattern.matching/fsharp)

# Pattern Matching de Listas en F#

- ML (F#) también ofrece *pattern matching* de listas
- Para ello se utiliza el **operador ::**
- El patrón **head :: tail** indica que
  - **head** es el primer elemento de la lista
  - Y **tail** la lista resultante de quitar el primer elemento

```
let lista = ["hola"; "mundo"; "adios"; "amigos"]
let rec concatenar lista =
    match lista with
    | cabeza :: cola -> cabeza + " " + concatenar cola
    | [] -> ""
;;
```

- Se aprecia un estilo **declarativo** (no imperativo)

Consulta el código en:

[pattern.matching/fsharp](https://pattern.matching/fsharp)



# Preguntas

Consulta el código en:

[pattern.matching/fsharp](https://pattern.matching/fsharp)

- Responda las siguientes preguntas acerca del siguiente código escrito en lenguaje ML (F#)

*Que función de orden superior es?*

*Cual es la principal diferencia con su implementación en C#?*

*Qué característica del lenguaje se usa?*

```
let rec higherOrder list f =  
  match list with  
  | head :: tail -> f head :: higherOrder tail f  
  | [] -> []  
  ;;  
printfn "%A" (higherOrder [1; 2; 3; 4; 5] ((+) 1))
```

*Qué se muestra en la consola?*

*Qué característica del lenguaje se usa?*

# Actividad Obligatoria

- En el reconocimiento de patrones **aparecen tres elementos**:
  1. El **elemento** sobre el cuál se aplica el patrón
  2. El **patrón**
  3. La **acción** a realizar si el patrón se evalúa cierto
- Los dos últimos elementos se pueden representar mediante funciones
  - Si el elemento es de tipo **T**, entonces
    - El patrón será **Predicate<T>**
    - La acción **Func<T, TResultado>**
- Por tanto, es posible implementar una clase **PatternMatch** que, recibiendo los tres elementos simule esta característica
- Analice, ejecute y comprenda el código de la etiqueta

Consulta el código en:

[pattern.matching/pattern.matching](https://pattern.matching/pattern.matching)

# Pattern Matching de Tipos en C#

- C# 7.0 incorporó un subconjunto de las funcionalidades de *pattern matching* de F#

```
static double Area(Figura figura)
{
    switch (figura)
    {
        case Circulo c:
            return Math.PI * c.Radio * c. Radio;
        case Cuadrado u:
            return u.Lado * u.Lado;
        case Rectangulo r:
            return r.Alto * r.Ancho;
        case Triangulo t:
            return t.Base * t.Altura / 2.0;
        default:
            throw new InvalidCastException("Tipo de figura desconocido");
        case null:
            throw new ArgumentNullException (nameof(figura));
    }
}
```

```
static bool EsRegular(Figura figura)
{
    switch (figura)
    {
        case Cuadrado c:
            return true;
        case Rectangulo r when (r.Alto == r.Ancho):
            return true;
        case Triangulo t:
            return t.Altura == t.Base * Math.Sqrt(3.0)/2.0;
        default:
            return false;
        case null:
            throw new ArgumentNullException(nameof(figura));
    }
}
```

Siempre se ejecuta si ningún otro patrón encaja, aunque físicamente no sea el último case

# Funciones de Orden Superior Típicas

- Recordemos que una **función de orden superior** es una función que
  - O bien recibe alguna función como parámetro
  - O bien retorna una función como resultado
- Existen numerosas funciones de orden superior, aunque las más típicas son:
  - **Filter**: Aplica un predicado a todos los elementos de una colección, devolviendo otra colección con aquellos elementos que satisfagan el predicado
  - **Map**: Aplica una función a todos los elementos de una colección, devolviendo otra nueva colección con los resultados obtenidos
  - **Reduce (Fold, Accumulate, Compress o Inject)**: Se aplica una función a todos los elementos de una lista, dado un orden, devolviendo un valor

# Funciones de Orden Superior en C#

- En el .Net Framework 4.0 se han añadido múltiples **funciones de orden superior genéricas** en **System.Linq**
  - Reciben colecciones de tipo **IEnumerable** utilizando métodos extensores
  - Reciben funciones de orden superior mediante los tipos delegado predefinidos **Predicate**, **Func** y **Action**
- Los nombres que han dado para las funciones de orden superior mencionadas son
  - **Filter**           ⇒       **Where**
  - **Map**             ⇒       **Select**
  - **Reduce**          ⇒       **Aggregate**

Consulta el código en:

*higher.order*

# Funciones de Orden Superior en .Net

Función	Recibe	Devuelve	Qué cambia	Ejemplos
Map (Select)	<code>IEnumerable&lt;T&gt;</code>	<code>IEnumerable&lt;Q&gt;</code>	<b>T</b> no tiene porque ser del mismo tipo que <b>Q</b> . Cambia el <b>tipo</b> de los elementos	<ul style="list-style-type: none"> <li>• “Dame los nombres de todos los clientes” (<code>Cliente-&gt;string</code>)</li> <li>• “Dame la Dirección y la Ciudad de todos los alumnos” (<code>Alumno -&gt; {Direccion, Cuidad}</code>)</li> </ul>
Filter (Where)	<code>IEnumerable&lt;T&gt;</code>	<code>IEnumerable&lt;T&gt;</code>	Cambia el <b>Nº</b> de elementos devueltos	<ul style="list-style-type: none"> <li>• “Dame los alumnos que viven en Oviedo”</li> <li>• “Dame los clientes que sean mayores de edad y más de 10.000 euros ahorrados”</li> </ul>
Reduce (Aggregate)	<code>IEnumerable&lt;T&gt;</code>	<b>Q</b>	<b>Q</b> puede ser <b>cualquier tipo</b> ( <code>int</code> , <code>List</code> , <code>Dictionary</code> , ...)	<ul style="list-style-type: none"> <li>• “Dame la suma de todas las edades de los alumnos” (<b>Q</b>: <code>int</code>)</li> <li>• “Dame una distribución de nombres de los clientes y cuantos clientes hay por cada nombre” (<b>Q</b>: <code>Dictionary&lt;string, int&gt;</code>)</li> </ul>

# Ejemplos de Select (Map)

- “Dame los nombres de todos los clientes”

```
IEnumerable<string> names = clients.Select(client => client.Name);
```

- “Dame la Dirección y la Ciudad de todos los alumnos”

// 1ª opción: Usando una clase creada a propósito para este fin

```
IEnumerable<FullAddress> fullAddresses = students.Select(  
    student => new FullAddress {  
        Address = student.Address,  
        City = student.City,  
    });
```

/\* 2ª opción: Usando tipos anónimos. En este caso solo podemos usar var, ya que un tipo anónimo no tiene nombre, y por tanto no podemos declarar una variable de su tipo directamente\*/

```
var fullAddresses2 = students.Select(student =>  
    new {  
        Address = student.Address,  
        City = student.City,  
    });
```

# Ejemplos de Where (Filter)

- *“Dame los alumnos que viven en Oviedo”*

```
IEnumerable<Student> studentsFromOviedo = students.Where(student =>  
student.City.ToLower().Equals("oviedo"));
```

- *“Dame los clientes que sean mayores de edad y más de 10.000 euros ahorrados”*

```
IEnumerable<Client> oviedoRichClients = clients.Where(client => client.Age  
>= 18 && client.Savings > 10000);
```



# Ejemplos de Reduce (Aggregate)

- “Dame la suma de todas las edades de los alumnos”

```
int sumOfAges = students.Aggregate(0, (previousSum, currentElement) => previousSum + currentElement.Age);
```

- O también existe una función Sum que se usa para estos casos particulares:

```
int sumOfAges2 = students.Sum(currentElement => currentElement.Age);
```

- “Dame una distribución de nombres de los clientes y cuantos clientes hay por cada nombre”

```
Dictionary<string, int> dictOfNameFrequency = clients.Aggregate(new Dictionary<string, int>(),  
    (dictSoFar, currentClient) => {  
        if (dictSoFar.ContainsKey(currentClient.Name))  
            dictSoFar[currentClient.Name]++;  
        else dictSoFar[currentClient.Name] = 1;  
        return dictSoFar;  
    });
```

# Map, Filter, Reduce en otros lenguajes

- La programación funcional se ha incorporado a multiples lenguajes de programación populares
  - **Python**
    - Map: `map(lambda x: x**2, number_list)`
    - Filter: `filter(lambda x: x < 0, number_list)`
    - Reduce: `reduce((lambda x, y: x * y), number_list)`
  - **Javascript**
    - Map: `number_list.map((value, index, array) => {return value * value; });`
    - Filter: `number_list.filter((value, index, array) => {return value < 0; });`
    - Reduce: `number_list.reduce((acc, currValue, currIndex, array) => {return acc * currValue; }, 1);`
  - **Java:** Ver presentación “New Functional Features of Java 8”
  - **Otros lenguajes**
    - Programación funcional: [https://en.wikipedia.org/wiki/Functional\\_programming](https://en.wikipedia.org/wiki/Functional_programming)
    - Soporte de funciones de orden superior: [https://en.wikipedia.org/wiki/Higher-order\\_function](https://en.wikipedia.org/wiki/Higher-order_function)
    - Map: [https://en.wikipedia.org/wiki/Map\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Map_(higher-order_function))
    - Filter: [https://en.wikipedia.org/wiki/Filter\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Filter_(higher-order_function))
    - Reduce (Fold): [https://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

# Otras funciones

Función	Recibe	Devuelve	Qué cambia
SelectMany	<code>IEnumerable&lt;T&gt;</code>	<code>IEnumerable&lt;Q&gt;</code>	Similar al <b>Select</b> pero obtiene una colección de elementos de tipo <b>Q</b> por cada elemento de tipo <b>T</b> . El resultado se concatena en una única colección.
Take	<code>IEnumerable&lt;T&gt;</code>	<code>IEnumerable&lt;T&gt;</code>	Toma 'n' elementos. Menor tamaño que la original.
Skip	<code>IEnumerable&lt;T&gt;</code>	<code>IEnumerable&lt;T&gt;</code>	Omite los primeros 'n' elementos de la original.
Range	<code>int n, int m</code>	<code>IEnumerable&lt;int&gt;</code>	Genera una colección de 'm' enteros comenzando en 'n'.
Repeat	<code>T element, int n</code>	<code>IEnumerable&lt;T&gt;</code>	Genera una colección con 'n' repeticiones de 'element'.
Reverse	<code>IEnumerable&lt;T&gt;</code>	<code>IEnumerable&lt;T&gt;</code>	Misma colección en orden inverso.
Any	<code>IEnumerable&lt;T&gt;</code>	<code>bool</code>	Comprueba si alguno de los elementos cumple el predicado
All	<code>IEnumerable&lt;T&gt;</code>	<code>bool</code>	Comprueba si todos los elementos cumplen el predicado
SequenceEqual	<code>IEnumerable&lt;T&gt;, IEnumerable&lt;T&gt;</code>	<code>bool</code>	Comprueba si las dos colecciones son iguales comparando sus elementos. Se le puede pasar un criterio de comparación.
Zip	<code>IEnumerable&lt;T1&gt;, IEnumerable&lt;T2&gt;</code>	<code>IEnumerable&lt;(T1, T2)&gt;</code>	Empareja cada elemento de la primera colección con el elemento de la segunda colección en su misma posición. Devuelve una tupla con cada pareja de elementos.
Zip	<code>IEnumerable&lt;T1&gt;, IEnumerable&lt;T2&gt;</code>	<code>IEnumerable&lt;TResult&gt;</code>	Empareja cada elemento de la primera colección con el elemento de la segunda colección en su misma posición. Devuelve el resultado de aplicar la función a cada pareja creada.



# Seminario 5

## Clausuras

# Listas por comprensión

- Las **listas por comprensión** son una característica de un lenguaje que permite crear listas basándose en listas existentes
- Usa la notación de *creación de conjuntos* de la teoría de conjuntos (usada en matemáticas y la lógica)
- Ejemplo:  
$$S = \{ 2x \mid x \in \mathbb{N}, x < 10 \wedge x \% 2 = 0 \}$$
- Lenguajes como Haskell, OCaml, F# o Python soportan la creación de listas por comprensión
- Ejemplo en Python:  

```
S = [2*x for x in range(10) if x%2==0]
```
- La pregunta es: Las soporta C#?

# LINQ

- LINQ es bastante parecido a las listas por comprensión:

```
public static IEnumerable<uint> NumeroNatural (uint max) {  
    uint n = 0;  
    while (n < max)  
        yield return n++;  
}  
  
static IEnumerable<uint> SinAzucarSintactico () {  
    return NumeroNatural(10)  
        .Where(x => x % 2 == 0)  
        .Select(x => 2 * x);  
}
```

Consulta el código en:

*[list.comprehensions](#)*

# LINQ Syntax Sugar


- LINQ ofrece **azúcar sintáctico** para poder crear listas por comprensión de forma similar a la que usan otras aproximaciones:

```
static IEnumerable<uint> SinAzucarSintactico () {  
    return NumeroNatural(10)  
        .Where(x => x % 2 == 0)  
        .Select(x => 2 * x);  
}  
  
static IEnumerable<uint> ConAzucarSintactico () {  
    return  
        from x in NumeroNatural(10)  
        where x % 2 == 0  
        select 2 * x;  
}
```

Consulta el código en:

*list.comprehensions*

- No obstante, no hay una equivalencia para todas las posibles expresiones expresadas en la primera sintaxis mostrada



# Programación orientada a objetos vs funcional

*Como cambian ciertas operaciones comunes gracias a las características adicionales de la programación funcional*



# Ordenación (POO)

- Dada la siguiente clase:

```
class Person {  
    public string Name { get; set; }  
    public string NIF { get; set; }  
    public override string ToString() { return Name + "; " + NIF; }  
}
```

- Si queremos ordenar personas por un criterio (Nombre) debemos hacer un IComparer:

```
class PersonByNameComparator : IComparer<Person> {  
    public int Compare(Person x, Person y) {  
        return x.Name.CompareTo(y.Name);  
    }  
}
```

- Para finalmente ordenar:

```
Array.Sort(persons, new PersonByNameComparator());
```

# Ordenación (Funcional)

- Basta con indicar el criterio de ordenación con una función

```
var sorted = persons.OrderBy(person => person.NIF);
```

# Generación de colecciones (POO)

- Generar una colección de números desde 0 hasta limit:

```
int [] OOPNumberGenerator (int limit) {  
    int counter = 0;  
    int [] temp = new int[limit];  
    while (counter < limit)  
        temp[counter] = counter++;  
    return temp;  
}
```

# Generación de colecciones (Funcional)

- Generar una colección de números desde 0 hasta `limit`, de forma lazy:

```
IEnumerable<int> LazyNumberGenerator (int limit) {  
    int counter = 0;  
    while (counter < limit) yield return counter++;  
}
```

# Procesar elementos de una colección (POO)

- Convertir números < 100 en string:

```
var numbers = OOPNumberGenerator(100);  
var temp = new string[numbers.Count()];  
  
for(int i = 0; i < 100; i++)  
    temp[i] = numbers[i].ToString();  
Show(temp);
```

# Procesar elementos de una colección (Funcional)

- Convertir números < 100 en string:

```
Show(LazyNumberGenerator(100).Select(number =>
    number.ToString()));
```

# Filtrar elementos de una colección (POO)

- Calcular los n<sup>º</sup>s primos <100:

```
numbers = OOPNumberGenerator(100);  
var tempInt = new int[numbers.Length];  
int counter = 0;  
foreach (var number in numbers)  
    if (IsPrime(number))  
        tempInt[counter++] = number;  
Array.Resize(ref tempInt, counter);  
Show(tempInt);
```

# Filtrar elementos de una colección (Funcional)

- Calcular los n<sup>º</sup>s primos <100:

```
Show(LazyNumberGenerator(100).Where(number =>  
IsPrime(number)));
```



# Hacer cálculos con elementos de una colección (POO)

- Calcular la suma de todos los primos <100:

```
numbers = OOPNumberGenerator(100);  
var result = 0;  
foreach (var number in numbers)  
    if (IsPrime(number))  
        result += number;  
Console.WriteLine(result);
```

# Hacer cálculos con elementos de una colección (Funcional)

- Calcular la suma de todos los primos <100:

```
Console.WriteLine(LazyNumberGenerator(100).  
    Aggregate((accum, number) => {  
        if (IsPrime(number))  
            return accum + number;  
        return accum;  
    }));
```

- O bien...

```
Console.WriteLine(LazyNumberGenerator(100).Where(n =>  
    IsPrime(n)).Sum());
```

# IEnumerable (POO)

- Para hacer la clase MyList enumerable...

```
class MyList<T> : IEnumerable<T>{  
    public IEnumerator<T> GetEnumerator() {  
        return new MyListEnumerator<T>(this);  
    }  
    IEnumerator IEnumerable.GetEnumerator() {  
        return GetEnumerator();  
    }  
}
```

- Es necesario además crear una clase que implemente IEnumerator<T>:

```
class MyListEnumerator<T> : IEnumerator<T> {  
    public MyListEnumerator(MyList<T> listToEnumerate) {...}  
    public void Dispose() {...}  
    public bool MoveNext() {...}  
    public void Reset() {...}  
    public T Current { get; }  
    object IEnumerator.Current { get { return Current; }}  
}
```

# IEnumerable (Funcional)

- Con programación funcional, ya no es necesario hacer una clase aparte que implemente `IEnumerator<T>`:

```
class MyListEnumeratorFunctional<T> : IEnumerable<T> {  
    public T GetElement(int pos) { ...}  
    public int Length { get; private set; }  
  
    public IEnumerator<T> GetEnumerator() {  
        for (int i = 0; i < this.Length; i++)  
            yield return GetElement(i);  
    }  
    IEnumerator IEnumerable.GetEnumerator() {  
        return GetEnumerator();  
    }  
}
```