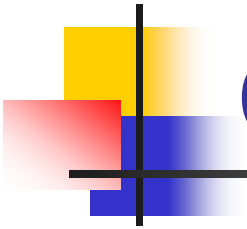
Tema 3: Herencia en C#

Programación Orientada a Objetos

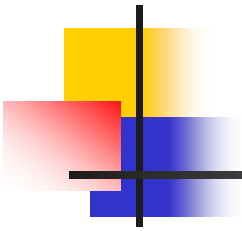
Curso 2008/2009

Begoña Moros Valle



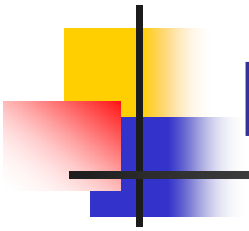
Contenido

- Herencia.
- Polimorfismo y ligadura.
- Clase object.
- Casting.
- Compatibilidad de tipos.
- Clases abstractas.
- Interfaces.
- Boxing y unboxing.
- Herencia múltiple.
- Genericidad.
- Acciones – Delegados.
- Iteradores.



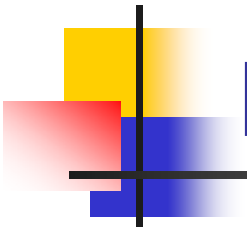
Herencia

- La herencia en C# comparte características tanto con Java como con C++:
  - Herencia **simple** (= Java)
  - Herencia **pública** (= Java)
  - Todos las clases heredan directa o indirectamente de **`object`** (= Java)
  - Por defecto, no está permitida la **redefinición** de métodos (= C++)
  - La aplicación de métodos puede resolverse por **ligadura** estática o dinámica (= C++)



Herencia y constructores

- **Los constructores no se heredan** (= Java y C++)
- El constructor de la clase hija tiene que invocar al de la clase padre utilizando la palabra clave **`base`**.
- Si no invoca al constructor del padre, el compilador añade **`base()`**.
- La llamada al constructor se realiza justo después de la lista de parámetros (= C++)



Redefinición de métodos

- Un **método o propiedad** sólo puede ser redefinido si se declara con el modificador `virtual` (= C++)

- Para redefinir un método en una subclase hay que utilizar el modificador `override`.

- En un **refinamiento**, se llama a la versión del padre con `base` (= `super` de Java)

Redefinición de métodos

- Si se define un método con la misma declaración que otro método **`virtual`** de la clase padre, podemos indicar que no lo redefinimos con el modificador **`new`**:
  - Se entiende que se define un método con la misma signatura, pero con **distinto significado**.
  - **No se aplicaría ligadura dinámica**.

Depósito 1/2

```
public class Deposito
{
    public Persona Titular { get; private set; }
    public virtual double Capital { get; protected set; }
    public int PlazoDias { get; private set; }
    public double TipoInteres { get; private set; }

    public Deposito(Persona titular, double capital,
          int plazoDias, double tipoInteres) {

       Titular = titular; Capital = capital;
       PlazoDias = plazoDias; TipoInteres = tipoInteres;
    }
…
```

Depósito 2/2

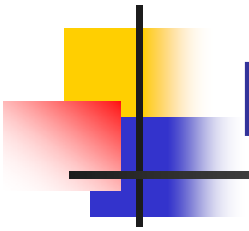
```
public class Deposito
{ …
    public virtual double Intereses
    {
      get
      {
          return (PlazoDias * TipoInteres * Capital) / 365;
      }
     }
     public double Liquidar()
     {
            return Capital + Intereses;
     }
 }
```

Depósito Penalizable 1/2

```
public class DepositoPenalizable : Deposito
{
    public bool Penalizado { get; set; }

    public override double Intereses
    {
        get
        {
            if (Penalizado)
                return base.Intereses / 2;
            else return Intereses;
        }
    }
…
```

Depósito Penalizable 2/2

```
public class DepositoPenalizable : Deposito
{   …

   public DepositoPenalizable(Persona titular, double capital,
                              int plazoDias, double tipoInteres):
      base(titular, capital, plazoDias, tipoInteres)
    {
         Penalizado = false;
    }
}
```

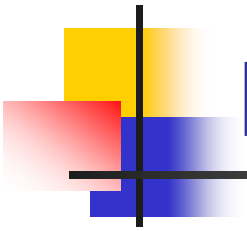


Redefinición y visibilidad

- Si el método redefinido es **`virtual`**:
  - No se puede modificar su nivel de visibilidad (distinto a Java y C++)
- Si el método **`no es virtual`** (no se redefine):
  - Podemos cambiar la visibilidad, aumentarla o reducirla.

Restringir la herencia

- Al redefinir un **método `virtual`**, podemos indicar que no se pueda redefinir en los subtipos con el modificador **`sealed`** (= `final` de Java)
- **Ejemplo**:
  - Podríamos definir como **`sealed`** la redefinición de `Intereses/get` en `DepositoEstructurado`.
  - Impediría que `DepositoGarantizado` pudiera cambiar la implementación.
- Una **clase** se puede definir como **`sealed`** indicando que no se puede heredar de ella (= `final` de Java)



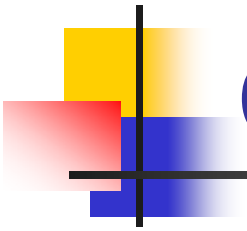
Polimorfismo y ligadura

- El **polimorfismo** está permitido sólo para **entidades de tipos referencia** (clases, interfaces).
- La **ligadura dinámica** sólo se aplica en tipos referencia y en métodos declarados con el modificador **`virtual`** (= C++)
  - Se aplica la versión del tipo dinámico, si la clase del objeto ha redefinido el método con **`override`**.
- La **ligadura estática** se aplica en el resto de casos.

Clase object

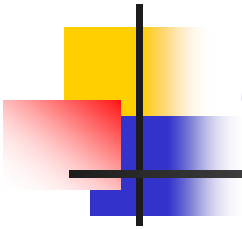
- La clase **`object`** representa la raíz de la jerarquía de tipos en C# y .NET

- Define **métodos básicos** para la plataforma:
  - `public virtual bool` **`Equals`**`(object otro)`
  - `public static bool` **`ReferenceEquals`** `(object obj1, object obj2)`
    - Comprueba siempre la identidad de objetos referencia y es aplicable a referencias nulas.



Clase object

- **Métodos básicos**:
  - `public virtual String` **`ToString`**`()`
  - `public Type` **`GetType`**`()`
    - Equivalente al `getClass()` de Java.
    - Para preguntar por el tipo de una variable se utiliza **`typeof`**`(var)`.
  - `public virtual int` **`GetHashCode`**`()`
  - `protected object` **`MemberwiseClone`**`()`
    - Realiza una copia superficial del objeto receptor de la llamada.



Casting

- Se puede aplicar un casting entre tipos compatibles:

```
estructurado = (DepositoEstructurado)deposito;
```

- Sin embargo, para los tipos referencia se define el operador **as**.

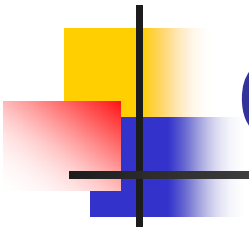
```
estructurado = deposito as DepositoEstructurado;
```

  - Devuelve `null` si la conversión no es correcta.
  - Similar al `dynamic_cast` de C++.

Compatibilidad de tipos

- Se define el operador **`is`** para consultar la compatibilidad de tipos (= `instanceof` de Java):

```
if (deposito is DepositoEstructurado)
{
     // El casting va a ser correcto
     estructurado = (DepositoEstructurado)deposito;
}
```

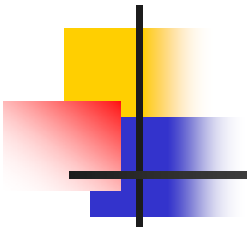


Clases abstractas

- Las clases pueden declararse como abstractas utilizando el modificador **`abstract`** .
- **Métodos y propiedades** se declaran abstractos con `abstract`.
- Si una **subclase** no implementa una declaración abstracta, debe declararse como abstracta.
- Una clase abstracta define un tipo, pero no se pueden construir objetos.
- Una clase es abstracta si define un **concepto abstracto** del cual no está permitido crear objetos.

Clases abstractas

```
public abstract class ProductoFinanciero
{
        public Persona Titular { get; private set; }

        public ProductoFinanciero(Persona titular) {
            Titular = titular;
        }
        public abstract double Beneficio { get; }

        public double Impuestos  {
            get {
                return Beneficio * 0.18;
            }
        }
}
```



Interfaces

- C# define el concepto de interfaz similar al de Java.
- Permite definir **propiedades y métodos**, pero no constantes.
- Una clase puede implementar múltiples interfaces.
- Una interfaz puede extender varias interfaces.
- Los miembros de una interfaz siempre son públicos.

Interfaces – Declaración

- **Declaración** de una interfaz:

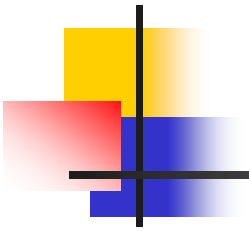
```
public interface Amortizable
{
    bool Amortizar(double cantidad);
}
```

- Una interfaz puede **extender múltiples interfaces**:

```
public interface Flexible : Amortizable, Incrementable
{
    void ActualizarTipoInteres(double tipo);
}
```

Interfaces – Implementación

```
public class DepositoPenalizable : Deposito , Amortizable
{
        …
          public bool Amortizar(double cantidad)
          {
              if (cantidad > Capital)
                  return false;

              Capital = Capital - cantidad;
              return true;
          }

}
```



Interfaces – Métodos repetidos

- Dos interfaces puede definir métodos o propiedades con la misma signatura (métodos repetidos)
- Si una clase implementa las dos interfaces con métodos repetidos, sólo podremos proporcionar una **única implementación** para esos métodos

  ➔ El mismo problema existe en Java.
- En cambio, en C# podemos resolverlo mediante la **implementación explícita de interfaces**.

Interfaces -Implementación explícita

- **Implementación explícita de una interfaz**:
  - El nombre del método va acompañado del nombre de la interfaz.
  - No se declara visibilidad. Se asume pública.

```
public class DepositoPenalizable : Deposito, Amortizable
{       …
      bool Amortizable.Amortizar(double cantidad)
      {  …  }
}
```

Interfaces -Implementación explícita

- La implementación explícita de interfaces tiene las siguientes **limitaciones**:
  - El método no puede ser utilizado dentro de la clase.
  - El método no puede ser aplicado sobre variables del tipo de la clase (en el ejemplo, `DepositoPenalizable`).
  - El método sólo puede ser aplicable sobre variables polimórficas del tipo de la interfaz:

```
DepositoPenalizable penalizable = new …;
penalizable.Amortizar(100); // error

Amortizable amortizable = penalizable;
amortizable.Amortizar(100);
```

Interfaces y estructuras

- Las estructuras pueden implementar interfaces.

```
public interface Reseteable
{
    void reset();
}

public struct Punto: Reseteable
{   …
    // Método Interfaz Reseteable
    public void reset()
    {
        x = 0;
        y = 0;
    }
}
```

Interfaces y estructuras

- Asignación a una interfaz:

```
Punto punto = new Punto(2, 3);;

Reseteable res = punto;
res.reset();

Punto otro = (Punto) res;
Console.WriteLine("Punto X: " + otro.X); // 0
```

- Una interfaz es un tipo referencia, ¿cómo puede apuntar a un tipo con semántica valor?

  ➔ **Boxing**

Boxing y unboxing

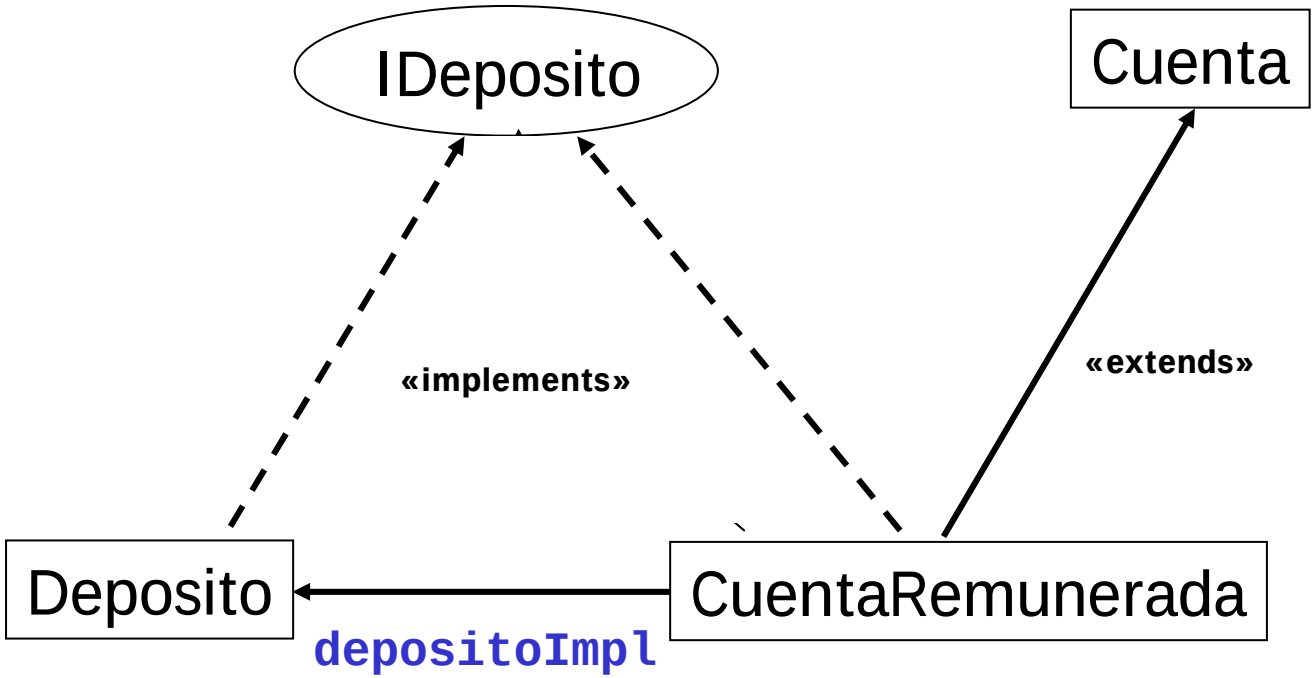
- **Boxing**: representación de tipos por valor como objetos por referencia.
- **Unboxing** realiza el proceso contrario.
- Con los tipos primitivos también se aplica el mecanismo de boxing:

```
int entero = 10;
Object objInt = entero; // Boxing
int otroEntero = (int)objInt; // Unboxing
```

- En C# **no hay clases envolventes** asociadas a tipos primitivos. Por tanto, en el casting del unboxing se puede utilizar el tipo primitivo.

Herencia múltiple

- C# es un lenguaje con herencia simple.
- Al igual que en Java, podemos **simular herencia múltiple** utilizando interfaces y relaciones de clientela.



Genericidad

- La genericidad en C# es parecida a Java, pero mejor implementada.

- **Ejemplo**: clase **`Contenedor`**

```
class Contenedor<T>
{
      public T Contenido
      {
          get; set;
      }
}
```

Genericidad

- Una clase genérica puede ser parametrizada a cualquier tipo:

```
Contenedor<Cuenta> contenedor = new Contenedor<Cuenta>();
contenedor.Contenido = cuenta;
Console.WriteLine(contenedor.Contenido);

Contenedor<int> contenedor2 = new Contenedor<int>();
contenedor2.Contenido = 10;
Console.WriteLine(contenedor2.Contenido);
```

Genericidad restringida

- Dentro de una clase genérica, sobre una entidad genérica sólo podemos aplicar:
  - Métodos disponibles en la clase `object`: `Equals`, `ToString`, etc.
  - Operadores de asignación (`=`) e igualdad (`==` y `!=`)
- Si queremos aplicar más operaciones debemos **restringir la genericidad**:
  - A una lista de **tipos compatibles** (= Java).
  - Para que sea una **clase** (tipo referencia): **`class`**.
  - Para que sea una **estructura** (tipo valor): **`struct`**.
  - A un tipo que tenga un **constructor sin parámetros**: **`new`**`()`.

Genericidad restringida

```
class Contenedor<T> where T : Deposito
{
        public T Contenido
        {
            get { return Contenido; }
            set
            {
                if (value.Titular is PersonaPreferente)
                    Contenido = value;
                else Contenido = null;
            }
        }
}
```

Genericidad restringida – Ejemplos

- El parámetro debe ser compatible con el tipo `Deposito` (clase) y el tipo `Amortizable` (interfaz):

```
class Contenedor<T> where T : Deposito, Amortizable
{ … }
```

- El parámetro debe ser compatible con `Amortizable` y debe ser una clase:

```
class Contenedor<T> where T : class, Amortizable
{ … }
```

Genericidad restringida – Ejemplos

- El parámetro debe ser compatible con `Amortizable` y debe ser una estructura:

```
class Contenedor<T> where T : struct, Amortizable
{ … }
```

- El parámetro debe ser compatible con la clase `Deposito` y la interfaz `Amortizable`, y debe proporcionar un constructor sin parámetros:

```
class Contenedor<T> where T : Deposito, Amortizable,
    new(){ … }
```

Genericidad – tipos compatibles

- Al igual que en Java, dos instancias de una clase genérica no son compatibles aunque los tipos de los parámetros sean compatibles:

```
Contenedor<Deposito> cDeposito =
                    new Contenedor<Deposito>();

Contenedor<DepositoEstructurado> cEstructurado =
                    new Contenedor<DepositoEstructurado>();

cDeposito = cEstructurado; // Error
```

Genericidad – tipos compatibles

- A diferencia de Java, no podemos utilizar una clase genérica sin parametrizar.

```
Contenedor contenedor; // Error
```

➔ No tenemos el problema de seguridad de tipos (tipo puro) de Java.

- C# tampoco permite saltar el control de tipos ni siquiera a través de `Object`:

```
object obj = cEstructurado;
// Error en ejecución
cDeposito = (Contenedor<Deposito>)obj;
```

Genericidad – tipos compatibles

- El **operador `is`** se puede aplicar sobre el tipo genérico parametrizado:

```
object obj = cEstructurado;

if (obj is Contenedor<Deposito>)
   Console.WriteLine(“Deposito");

if (obj is Contenedor<DepositoEstructurado>)
   Console.WriteLine(“Deposito Estructurado");
```

- **El CLR de .NET maneja tipos genéricos.**

Genericidad – Métodos

- Al igual que en Java, la compatibilidad de tipos limita el paso de parámetros:

```
public double PosicionGlobal(List<Deposito> depositos){
        double valor = 0;
        foreach (Deposito deposito in depositos)
        {
            valor += deposito.Capital;
         }
         return valor;
}
```

- El método no podría ser parametrizado a una lista de depósitos estructurados.

Genericidad – Métodos

- C# no tiene tipos comodín, pero permite **restringir la genericidad de un método**:

```
public double PosicionGlobal<T>(List<T> depositos)
    where T: Deposito { … }
```

- Al igual que en Java, la genericidad aplicada a los métodos hace **inferencia de tipos**.
- Sin embargo, si se conoce el tipo se puede indicar en la llamada:

```
banco.PosicionGlobal<DepositoEstructurado>(estructurados);
```

Acciones – Delegados

- **Acción**: representación de una referencia a un método como un objeto.

- Las acciones permiten establecer código como parámetro de un método.

- Para la definición de acciones, C# declara el concepto de **delegado**:
  - Similar a un puntero a función de C/C++.
  - Incluye **control de tipos** y permite definir **referencias a métodos de instancia**.

Delegados

- **Ejemplo**:
  - Clase **`Banco`** almacena una colección de cuentas.
  - Declara el método **`Buscar`**`( )` parametrizado con una condición de búsqueda (delegado).

- Declaración del **tipo delegado `Test`**:

```
public delegate bool Test(Cuenta cuenta);
```

  - Representa un tipo (`Test`).
  - Cualquier método que tenga como parámetro una `Cuenta` y devuelva un booleano es compatible con el delegado `Test`.

Delegados

- Declaración del método **`Buscar`**`()`:

```
public Cuenta Buscar(Test test)
{
    foreach (Cuenta cuenta in cuentas) {
        if (test(cuenta))
            return cuenta;
    }
    return null;
}
```

- El método declara como parámetro un delegado de tipo `Test`.
- En el código utiliza el nombre del parámetro para invocar al método.

Delegados

- Definimos una clase con varias **condiciones de búsqueda**:

```
public class CondicionesBusqueda
{

    public bool CapitalAlto(Cuenta cuenta)
    {
        return cuenta.Saldo > 100000;
    }

    public bool SaldoCero(Cuenta cuenta)
    {
        return cuenta.Saldo == 0;
    }
}
```

Delegados

- Aplicamos el método **`Buscar`**( ) con las condiciones de búsqueda:

```
Cuenta resultado;

CondicionesBusqueda condiciones = new CondicionesBusqueda();

resultado = banco.Buscar(condiciones.CapitalAlto);

resultado = banco.Buscar(condiciones.SaldoCero);
```

Delegados genéricos

- Los delegados también pueden declararse genéricos:

```
public delegate bool Test<T>(T elemento);
```

  - El ejemplo representa un delegado que acepta cualquier tipo como parámetro y retorna un booleano.

- Al utilizar el tipo delegado se indica el tipo del parámetro:

```
public Cuenta Buscar(Test<Cuenta> test) { … }
```

Delegados anónimos

- **Motivación**:
  - Resulta inconveniente tener que definir un método por cada condición de búsqueda.

- **Solución**: **delegados anónimos**

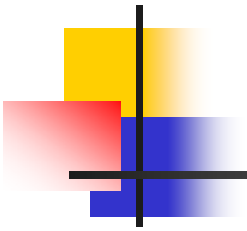
```
banco.Buscar(delegate (Cuenta c) { return c.Saldo < 0;});
```

Expresiones lambda

- Las expresiones lambda representan una **simplificación de los delegados anónimos**:

```
banco.Buscar( c => c.Saldo < 0 );
```

- Una expresión lambda tiene dos partes separadas por =>:
  - **Parte izquierda**: lista de parámetros separados por comas. No se indica el tipo, ya que se deduce de la definición del tipo delegado (`Cuenta` en el ejemplo)
  - **Parte derecha**: expresión que se evalúa al tipo de retorno del tipo delegado (booleano en el ejemplo)



Iteradores

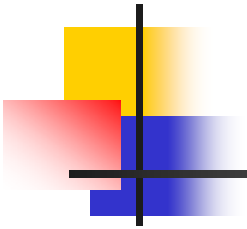
- El modelo de iteradores de C# es similar al de Java.
- Cualquier clase que quiera ser iterable debe implementar la **`IEnumerable`**:

```
public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

Iteradores

- Interfaz **`IEnumerator`**:
  - Diferente a un iterador de Java. No hay método `remove()`
  - Método **`MoveNext`**: avanza hasta el siguiente elemento, indicando si ha conseguido avanzar.
  - Propiedad **`Current`**: elemento actual.
  - Método **`Reset`**: sitúa el iterador en estado inicial, justo antes del primer elemento.

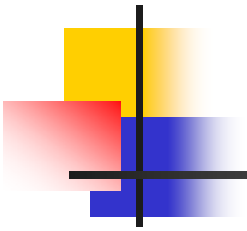
```
public interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```



Iteradores

- Ejemplo de uso de un iterador:

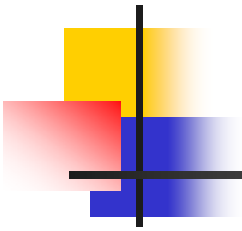
```
public Cuenta Buscar(Test test)
{
    IEnumerator<Cuenta> enumerador = cuentas.GetEnumerator();

    while (enumerador.MoveNext())
    {
        Cuenta cuenta = enumerador.Current;
        if (test(cuenta))
            return cuenta;
    }
    return null;

}
```



Iteradores

- Al igual que en Java, se puede omitir el uso de un iterador con un recorrido **`foreach`**:

```
public Cuenta Buscar(Test test)
{
    foreach (Cuenta cuenta in cuentas)
    {
        if (test(cuenta))
            return cuenta;
    }
    return null;
}
```



Bloques de iteración

- En C# es posible definir métodos con **bloques de iteración**:
  - Un método que retorna un objeto iterable.
  - La ejecución del método se detiene cada vez que se llama a **`yield return`** para retornar un elemento del recorrido.
  - Cuando se solicita el siguiente elemento, continúa la ejecución del método hasta alcanzar el siguiente `yield`.
  - La iteración finaliza al terminar el método o ejecutar **`yield break`**.

Bloques de iteración

- **Ejemplo**:
  - Método de búsqueda que permite recorrer los elementos que cumplen una condición de búsqueda.

```
public IEnumerable<Cuenta> Buscar2(Test test)
{
    foreach (Cuenta cuenta in cuentas)
    {
        if (test(cuenta))
            yield return cuenta;
    }
    yield break;
}
```

Bloques de iteración

- Uso del método de iteración:

```
foreach (Cuenta cuenta
        in banco.Buscar2(elemento => elemento.Saldo > 400))
{
    Console.WriteLine("Cuenta " + cuenta);
}
```

Implementación iteradores

- Los bloques de iteración son utilizados como implementación de las clases iterables.
- **Ejemplo**:
  - La clase **Banco** es iterable. En un recorrido retorna las cuentas que contiene.
  - La implementación de la interfaz se apoya en un método que usa un bloque de iteración.
  - C# obliga a implementar dos versiones del método `getEnumerator()`.

```
foreach (Cuenta cuenta in banco) {
    Console.WriteLine("Cuenta " + cuenta);
}
```

Implementación iteradores

```
class Banco: IEnumerable<Cuenta> {
  private List<Cuenta> cuentas = new List<Cuenta>();

  private IEnumerator<Cuenta> getEnumerador() {
      foreach (Cuenta cuenta in cuentas) {
          yield return cuenta;
      }
  }
  IEnumerator<Cuenta> IEnumerable<Cuenta>.GetEnumerator() {
      return getEnumerador();
  }
  IEnumerator System.Collections.IEnumerable.GetEnumerator() {
      return getEnumerador();
  }
```