

Preguntas detonadoras



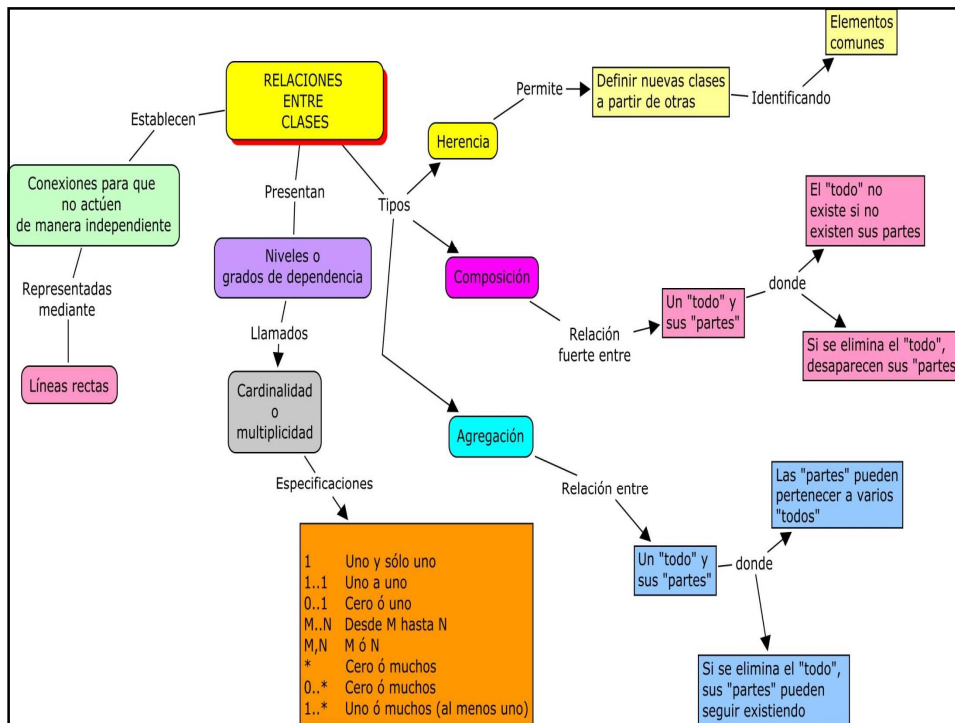
- ¿Qué es y para qué sirve una clase parametrizada?
- Una clase parametrizada, ¿Qué tipo de parámetro recibe?
- ¿Cuáles con las colecciones genéricas incluidas en el framework?
- ¿Es posible insertar uno o varios objetos dentro de otro? ¿Cómo?
- ¿En qué se parece una composición a una agregación? ¿En qué difieren?
- ¿Cuándo se recomienda implementar composición? ¿Cuándo agregación?
- Cuando una clase define una colección privada de objetos, ¿cómo pueden consultarse sus datos desde el exterior?
- ¿Se puede implementar un iterador por medio de un método? ¿y de una propiedad?

3

Relaciones entre clases:

Herencia, Composición y
Agregación

4

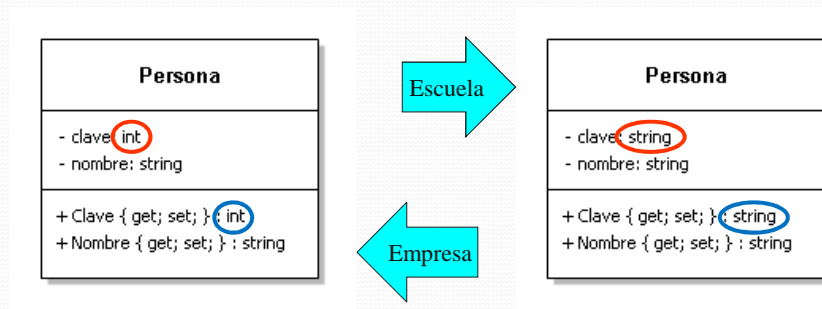


Clases parametrizadas o genéricas

- Ejemplo: Una empresa y una escuela desean almacenar la clave y nombre de sus personas:
 - *Clave: Entero ó String*
 - *Nombre: String*
- Pero en la empresa la clave es numérica entera y en la escuela es una cadena.

Clases parametrizadas o genéricas (cont.)

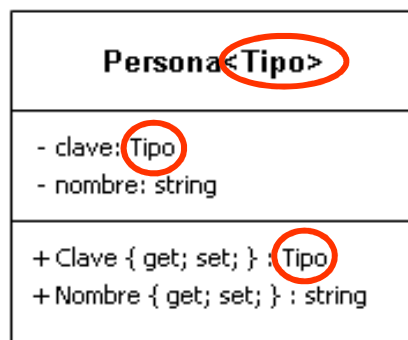
Clases iguales
(excepto en el tipo de dato de la clave)



7

Clases parametrizadas o genéricas (cont.)

- Diseñar una clase parametrizada que sirva para ambos casos:



8

Codificación de la clase parametrizada

```
class Persona<Tipo> // Clase parametrizada o genérica
{
    // Atributos privados
    private Tipo clave; // Se define el tipo de dato de la clave
    private string nombre;

    // Propiedades públicas
    public Tipo Clave // Se define el tipo de dato de la propiedad
    {
        get { return clave; }
        set { clave = value; }
    }

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }
}
```

9

Clases parametrizadas o genéricas (cont.)

// Declaración y creación de los objetos

```
Persona<int> miEmpleado = new Persona<int>();
```

```
Persona<string> miEstudiante = new Persona<string>();
```

// Uso de los objetos

```
miEmpleado.Clave = int.Parse(txtClave.Text);
```

```
miEstudiante.Clave = txtClave.Text;
```



10

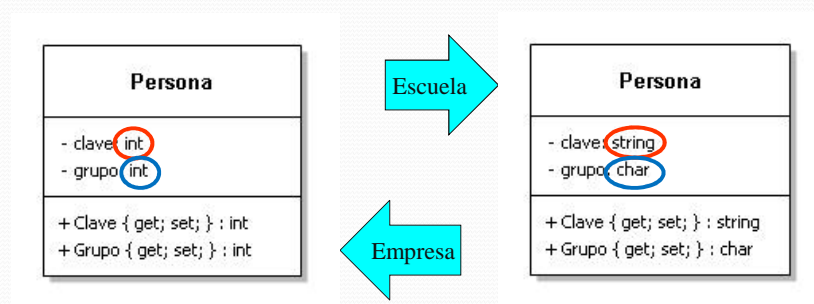
Clases parametrizadas con varios tipos

- Ejemplo: Una empresa y una escuela desean almacenar la clave y grupo de sus personas:
 - Clave: *Entero ó string*
 - Grupo: *Entero ó caracter*

11

Clases parametrizadas con varios tipos

Clases iguales
(excepto en los tipos de datos)



12

Codificación de la clase parametrizada con varios tipos

```
class Persona<Tipo1, Tipo2> // Clase parametrizada con varios tipos
{
    // Atributos privados
    private Tipo1 clave;
    private Tipo2 grupo;

    // Propiedades públicas
    public Tipo1 Clave
    {
        get { return clave; }
        set { clave = value; }
    }

    public Tipo2 Grupo
    {
        get { return grupo; }
        set { grupo = value; }
    }
}
```

13

Clases parametrizadas con varios tipos (cont.)

```
// Declaración y creación de los objetos
Persona<int, int> miEmpleado = new Persona<int, int>();
Persona<string, char> miEstudiante = new Persona<string, char>();

// Uso de los objetos
miEmpleado.Clave = int.Parse(txtClave.Text);
miEmpleado.Grupo = int.Parse(txtGrupo.Text);

miEstudiante.Clave = txtClave.Text;
miEstudiante.Grupo = char.Parse(txtClave.Text);
```

14

Colecciones genéricas en C#

- Incluidas en el namespace `System.Collection.Generic`
- Incorporadas a partir del .NET Framework 2.0
- Contiene clases e interfaces que definen tipos genéricos para instanciar colecciones
- Permite modelar estructuras de datos

15

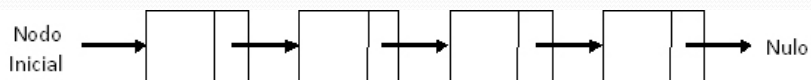
Estructuras de datos en C#

Colección (clase genérica)	Estructura de datos
<code>ArrayList</code>	Arreglos
<code>Stack</code>	Pilas
<code>Queue</code>	Colas
<code>List</code>	Listas enlazadas simples
<code>LinkedList</code>	Listas enlazadas dobles

16

Listas enlazadas simples

- Estructura de datos compuesta de nodos en secuencia enlazados a través de una referencia (apuntador).
- Cada nodo se compone de 2 partes:
 - Datos
 - Referencia al siguiente nodo
- Además, hay una referencia al primer nodo de la lista y el último nodo apunta a nulo



17

La clase genérica List

- Modela listas enlazadas en C#
- Requiere un parámetro adicional para definir el tipo de dato que almacena
- El parámetro se coloca entre < y >
- P. ejem.
 - `List <int> miListaSimpleEnteros;`
 - `List <double> miListaReales;`
 - `List <Empleado> miListaEmpleados;`

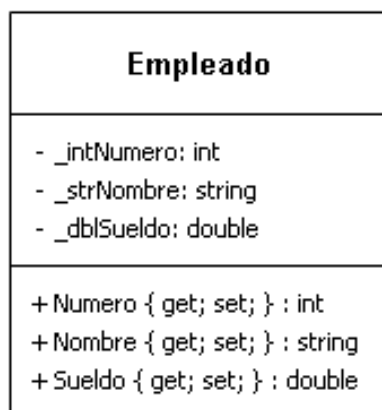
18

Principales métodos y propiedades de la clase genérica List

Método o propiedad	Uso
Clear()	Elimina todos los nodos de la lista
Add()	Agrega un nodo al final de la lista
Remove()	Elimina la primera ocurrencia de un nodo de la lista y devuelve un valor booleano para confirmar la operación
Contains()	Determina si un nodo se encuentra almacenado en la lista
Count	Devuelve la cantidad de nodos almacenados en la lista
Sort()	Ordena en forma ascendente los nodos de la lista
GetEnumerator()	Recorre los nodos de la lista y devuelve un enumerador.

19

Ejemplo de lista



- Crear una clase para modelar objetos con datos de empleados
- Crear una lista que almacene muchos empleados

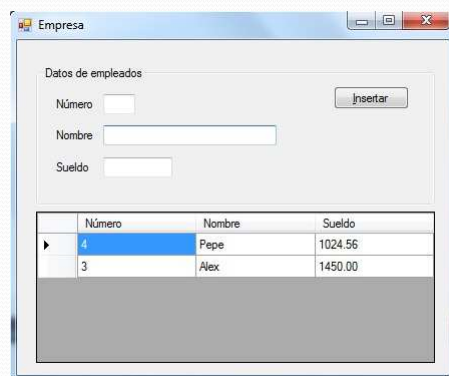
20

Creación de un objeto de una lista genérica

- Creación de un objeto de una lista genérica
 - `List<Empleado> miEmpresa = new List<Empleado>();`
- El objeto **miEmpresa** es una lista genérica que almacena objetos de la clase **Empleado**

21

Captura de datos de un nodo de la lista genérica



Número	Nombre	Sueldo
4	Pepe	1024.56
3	Alex	1450.00

- Capturar los datos de un empleado en *textBoxes*
- Al oprimir el botón insertar, crear un objeto de la clase **Empleado** e insertarlo en la lista genérica
- Mostrar los datos en un *dataGridView*

22

Creación de un empleado (nodo de la lista)

- Crear un objeto con los datos de un empleado para agregarlo a la lista
 - `Empleado miEmpleado = new Empleado();`
- Llenar los datos del empleado
 - `miEmpleado.Numero = int.Parse(textBox1.Text);`
 - `miEmpleado.Nombre = textBox2.Text;`
 - `miEmpleado.Sueldo = double.Parse(textBox3.Text);`

23

Insertar un empleado (agregar un nodo a la lista)

- Insertar el empleado en la lista
 - `miEmpresa.Add(miEmpleado);`
- Crear el dataGridView

```
dataGridView1.Columns.Add("Número", "Número");
dataGridView1.Columns.Add("Nombre", "Nombre");
dataGridView1.Columns.Add("Sueldo", "Sueldo");

// Inicializa las propiedades del dataGridView1
dataGridView1.ReadOnly = true;
dataGridView1.AllowUserToAddRows = false;
dataGridView1.AllowUserToDeleteRows = false;
dataGridView1.AutoSizeColumnsMode =
    DataGridViewAutoSizeColumnsMode.Fill;
```

24

Iterador

- Es un conjunto de instrucciones que devuelve una secuencia de valores del mismo tipo.
- Puede invocarse mediante un ciclo **foreach**
- P.ejem.

```
foreach(int Dato in X)
{
    .....
}
```

25

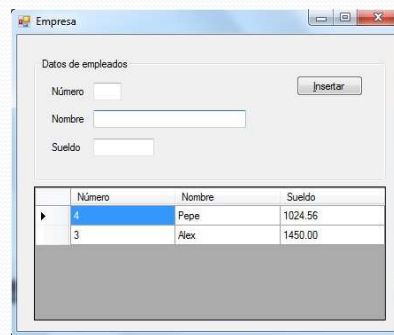
El ciclo foreach

- Se utiliza para recorrer los elementos de una colección y obtener la información deseada.
- No se debe utilizar para modificar el contenido de la colección.
- foreach repite un grupo de instrucciones incluidas en el cuerpo del ciclo para cada elemento de una estructura de datos.

26

Uso de un iterador para limpiar los textBoxes

```
foreach(Control x in groupBox1.Controls)
    if(x is TextBox)
        x.Text="";
```



27

Ejemplo de un iterador en una colección genérica de una lista

- Para recorrer los nodos de una lista

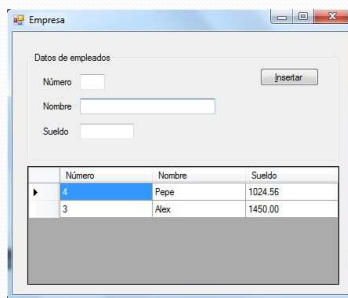
```
foreach(Empleado miEmpleado in miEmpresa)
```

- Donde:
- *Empleado*: Clase que define los datos de los empleados
- *miEmpleado*: Objeto de tipo *Empleado* que contiene los datos de un empleado
- *miEmpresa*: Objeto de la colección genérica

28

Mostrar los datos de una lista en un dataGridView

```
dataGridView1.Rows.Clear();  
foreach(Empleado miEmpleado in miEmpresa)  
{  
    dataGridView1.Rows.Add(miEmpleado.Numero,  
        miEmpleado.Nombre, miEmpleado.Sueldo)  
}
```



29

Uso de iteradores



Recuerde que no es recomendable incluir sentencias para desplegar datos en pantalla en un método de una clase, ya que se restringe el uso de dicha clase exclusivamente para la plataforma en la que se está trabajando y no se puede reutilizar en otras plataformas

30

La interfase IEnumerator

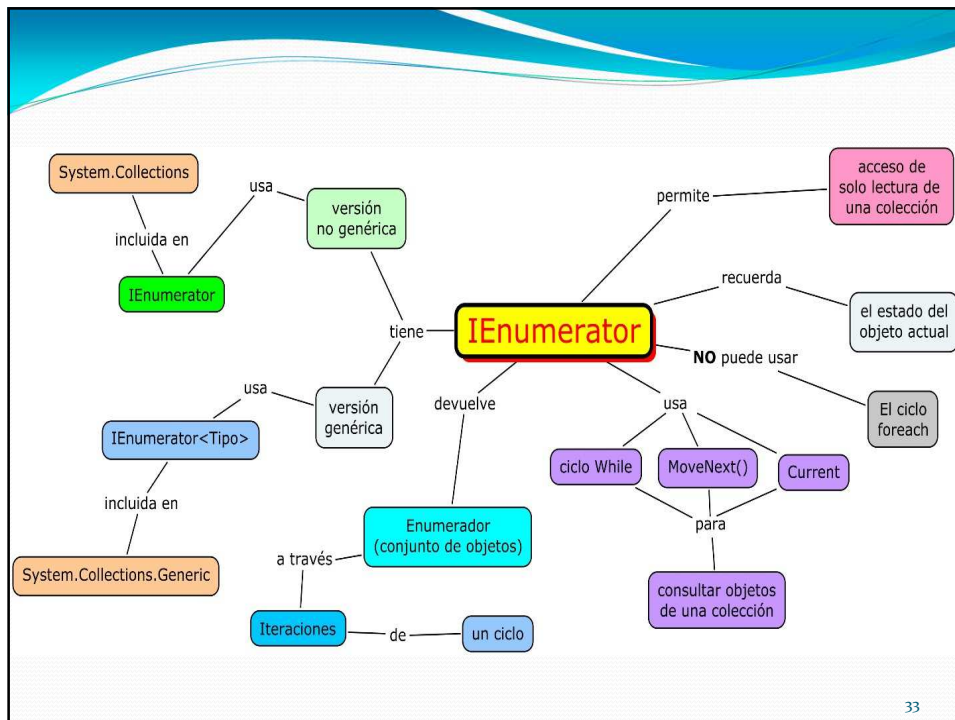
- Incluida en el espacio de nombres `System.Collections`
- Se utiliza para recorrer un conjunto de datos
- Acceso de solo lectura mediante iteraciones simples
- Proporciona dos métodos abstractos y una propiedad para consultar objetos de una colección
- IEnumerator es el fundamento de implementación de los iteradores

31

La interfase IEnumerator

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

32



33

La interfase IEnumerable

- Interfase parametrizada `IEnumerable<Tipo>`
- Incluida en el espacio de nombres `System.Collections.Generic`
- Se utiliza para recorrer un conjunto de datos
- Acceso de solo lectura mediante iteraciones simples a través de un ciclo `foreach`
- Utiliza el método `GetEnumerator()`

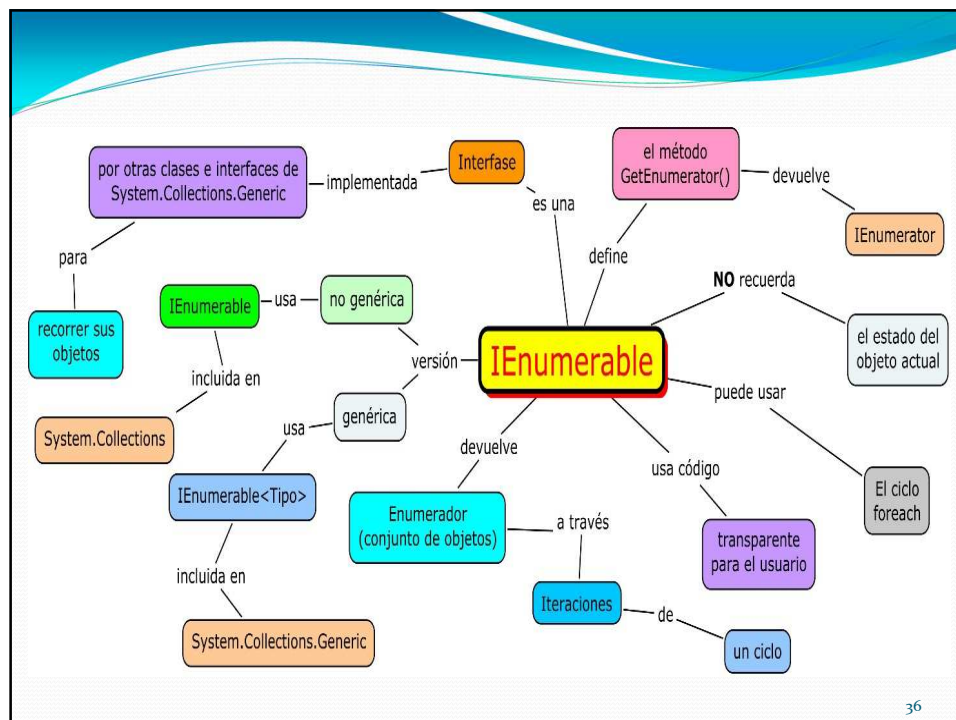
34

La interfase IEnumerable

```
// System.Collections.Generic
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

// System.Collections
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

35



36

El método GetEnumerator()

- Sirve para implementar iteradores
- El método GetEnumerator() devuelve un enumerador que recorre en iteraciones una colección.
- Lo contiene System.Collections
- Se utiliza por medio del ciclo *foreach*

37

Implementación de un iterador a través del método GetEnumerator()

- El método GetEnumerator() devuelve una secuencia de valores del mismo tipo IEnumerator<T>.
- Utiliza la instrucción *yield return* para devolver cada elemento
- Utiliza la instrucción *yield break* para finalizar la iteración (ciclo) cuando sea necesario.

38

Ejemplo de GetEnumerator()

```
public IEnumerator GetEnumerator()
{
    if(Arreglo.Tamaño == 0)
        yield break;
    for(i=0; i<Arreglo.Tamaño; i++)
        yield return Arreglo[i];
}
```



El método GetEnumerator() contiene la implementación de la lógica necesaria para recorrer todos los nodos de una colección, sin embargo, no se invoca directamente, sino a través del ciclo foreach

39

Diferencias entre IEnumerator e IEnumerable

	<i>IEnumerator</i>	<i>IEnumerable</i>
Internamente utiliza el ciclo foreach	x	✓
Recuerda el estado del objeto actual	✓	x
Internamente usa ciclo while	✓	x
Contiene la propiedad Current	✓	x
Contiene los métodos MoveNext() y Reset()	✓	x

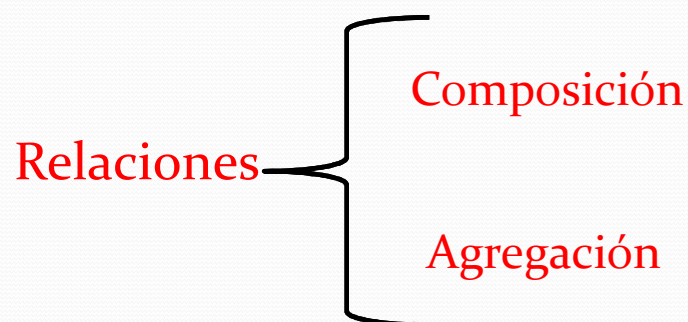
40

¿Cuándo usar IEnumerator y cuándo IEnumerable?

- Se recomienda **IEnumerator** cuando:
 - Se tiene una colección de objetos solamente con una forma de enumerar
 - Se implementa GetEnumerator()
- Se recomienda **IEnumerable** cuando:
 - Se requieren varios iteradores (varias estrategias para recorrer el conjunto de datos)
 - No se pueden implementar varios GetEnumerator()
 - Se implementan propiedades de solo lectura

41

Relaciones entre clases



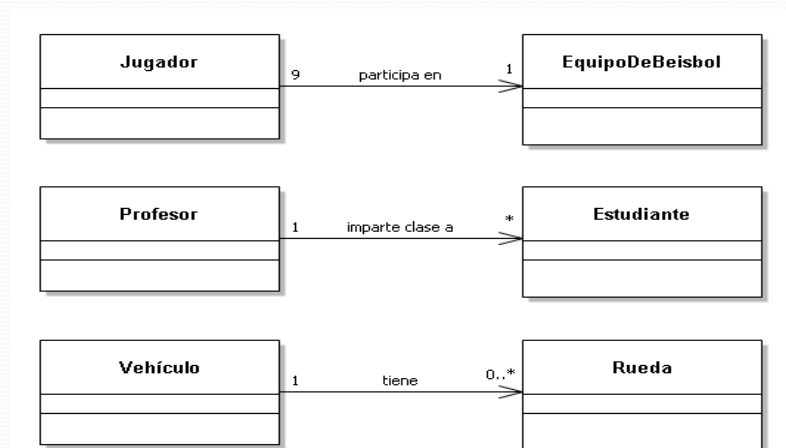
42

Grados de dependencia (cardinalidad o multiplicidad)

Representación	Descripción de la cardinalidad
1	Uno y sólo uno
1..1	Uno a uno
0..1	Cero ó uno
M..N	Desde M hasta N
M,N	M ó N
*	Cero ó muchos
0..*	Cero ó muchos
1..*	Uno ó muchos (al menos uno)

43

Ejemplos

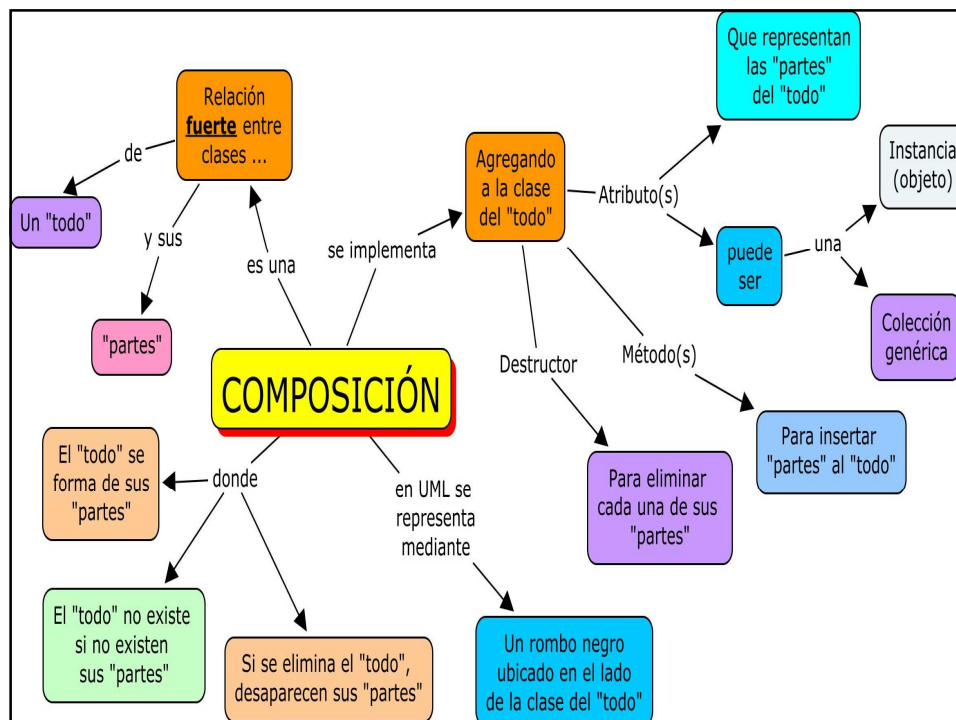


44

Composición

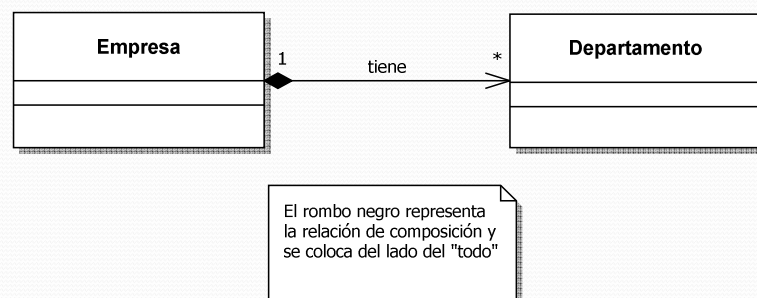
- Define una relación fuerte entre clases
- Se utiliza para modelar un "todo" y sus "partes" donde ...
 - El "todo" no puede existir si no existen sus "partes"
 - Las "partes" desaparecen cuando se elimina el "todo"

45



Representación de la composición

- Gráficamente se representa colocando un rombo negro en el extremo de la clase constituida (parte del “todo”).



47

Composición

- Un objeto puede tener como miembro a otro objeto definido previamente.
- Cuando un objeto se encuentra compuesto por otros objetos, se dice que hay “Composición”.
- La composición permite implementar relaciones del tipo “tiene un”.
 - Ejemplo: Un Auto “tiene un” Motor.
- Una característica importante de la composición es que la clase del “todo” regularmente contiene un destructor para eliminar sus “partes”

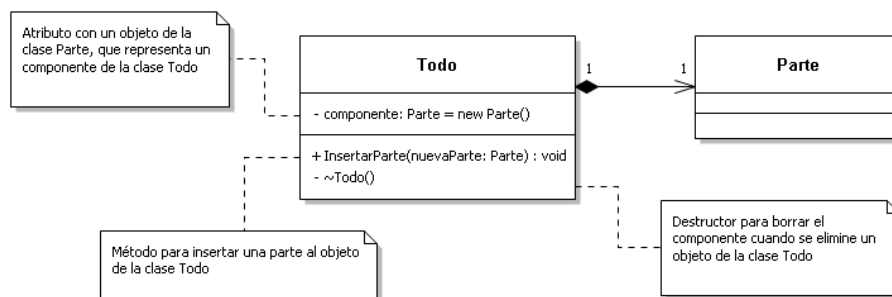
48

Reglas para que exista composición

- Existen tres reglas para que se presente la relación de composición entre dos clases:
 1. La clase del “todo” **DEBE** tener un atributo de tipo “parte”
 - a) Un objeto cuando es relación 1..1
 - b) Una colección genérica si es relación 1..*
 2. La clase del “todo” **DEBE** tener un método para insertarle objetos de tipo “parte”
 3. La clase del “todo” **DEBE** tener el destructor
 - a) Al eliminar el objeto del “todo”, también se deben eliminar sus “partes”

49

Relación 1..1 en Composición



50

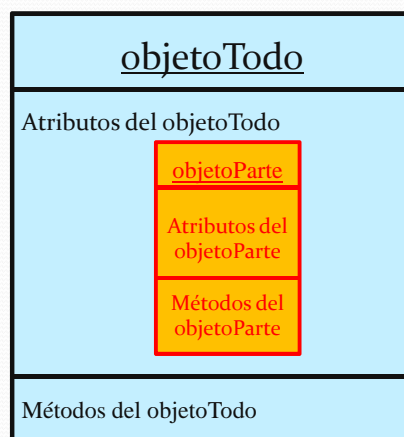
Implementación de relación 1..1 en composición

```
class Todo
{
    // Atributo (objeto componente del Todo)
    private Parte componente = new Parte();

    // Método para insertar un componente
    public void InsertarParte(Parte nuevaParte) {
        componente = nuevaParte;
    }
    // Destructor (elimina el componente)
    ~Todo() {
        componente = null;
    }
}
```

51

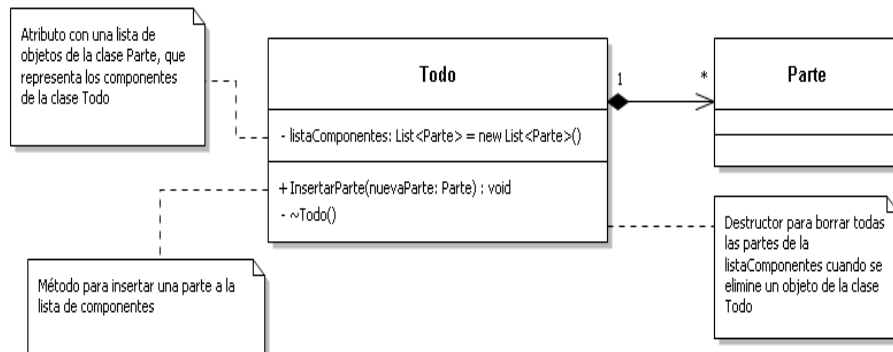
Representación de la relación 1..1



- Un objeto de la clase del “todo” tiene dentro un objeto de la clase “parte”
- El objetoTodo tiene un objetoParte como atributo

52

Relación 1..* en Composición



53

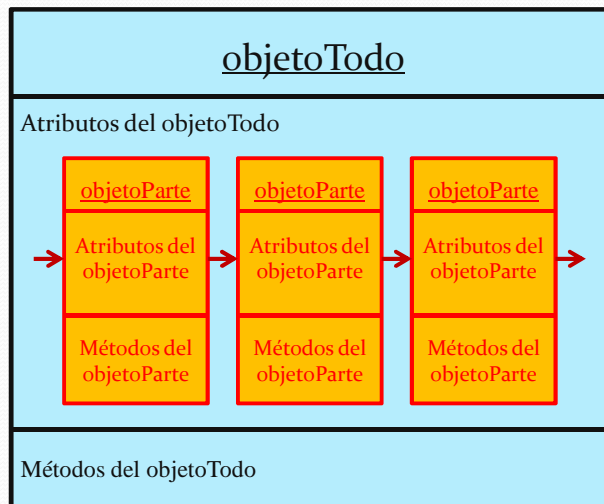
Implementación de relación 1..* en composición

```
class Todo
{
    // Atributo (lista de componentes del Todo)
    private List<Parte> listaComponentes = new List<Parte>();

    // Método para insertar una parte a la lista
    public void InsertarParte(Parte nuevaParte) {
        listaComponentes.Add(nuevaParte);
    }
    // Destructor (elimina el componente)
    ~Todo() {
        listaComponentes.Clear();
    }
}
```

54

Representación de la relación 1..*

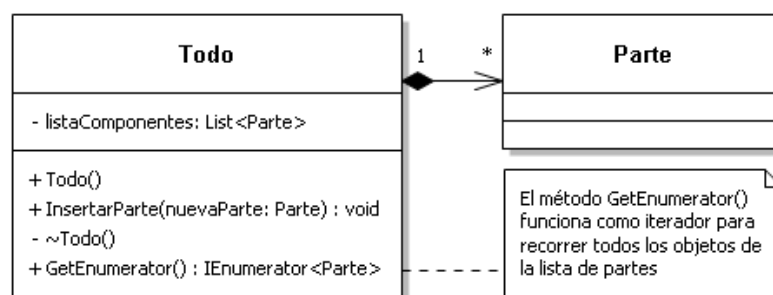


- Un objeto de la clase del "todo" tiene dentro una colección de objetos de la clase "parte"
- El objetoTodo tiene una lista de objetoParte como atributo

55

¿Cómo recorrer todas las partes de la ListaComponentes?

- Implementar un iterador por medio del método GetEnumerator()



56

Implementación del iterador

```
class Todo
{
    . . .
    . . .
    . . .
    public IEnumerator<Parte> GetEnumerator()
    {
        foreach (Parte p in listaComponentes)
            yield return p;
    }
}
```

57

Otra forma de implementación del iterador

```
class Todo
{
    . . .
    . . .
    . . .
    public IEnumerator<Parte> GetEnumerator()
    {
        return listaComponentes.GetEnumerator();
    }
}
```

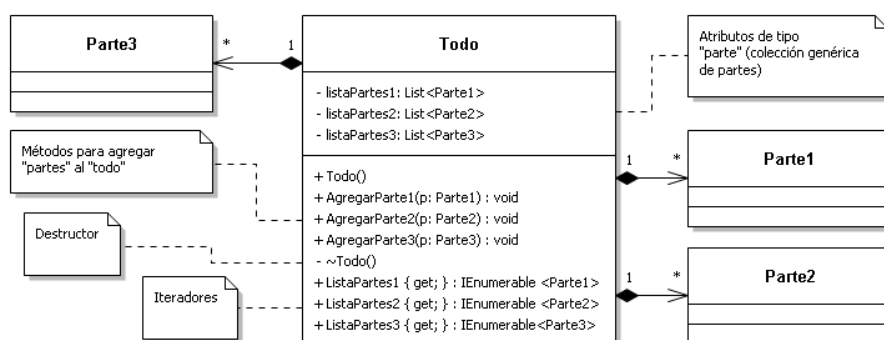
58

Modelo UML con varias composiciones

- Un objeto de tipo “todo” puede componerse de objetos de diferentes tipos de “partes”
- El “todo” tiene un atributo de cada tipo de “parte”
- El “todo” tiene un método para agregar cada tipo de “parte”
- El “todo” tiene un destructor para eliminar todas sus “partes”
- El “todo” tiene varios iteradores (uno para cada tipo de “parte”).

59

Modelo UML con varias composiciones (cont.)



60

Diseño de una clase con varios iteradores

- Cuando una clase requiere varios iteradores, estos no pueden implementarse a través del método GetEnumerator().
- Una clase no puede implementar varios métodos GetEnumerator().
- Se recomienda implementar una propiedad de solo lectura para cada iterador.

61

Implementación de iteradores a través de propiedades

```
class Todo
{
    . . .
    public IEnumerable<Parte1> ListaPartes1
    {
        get
        {
            foreach(Parte1 p in listaPartes1)
                yield return p;
            yield break;
        }
    }
}
```

62

Composición vs. Herencia

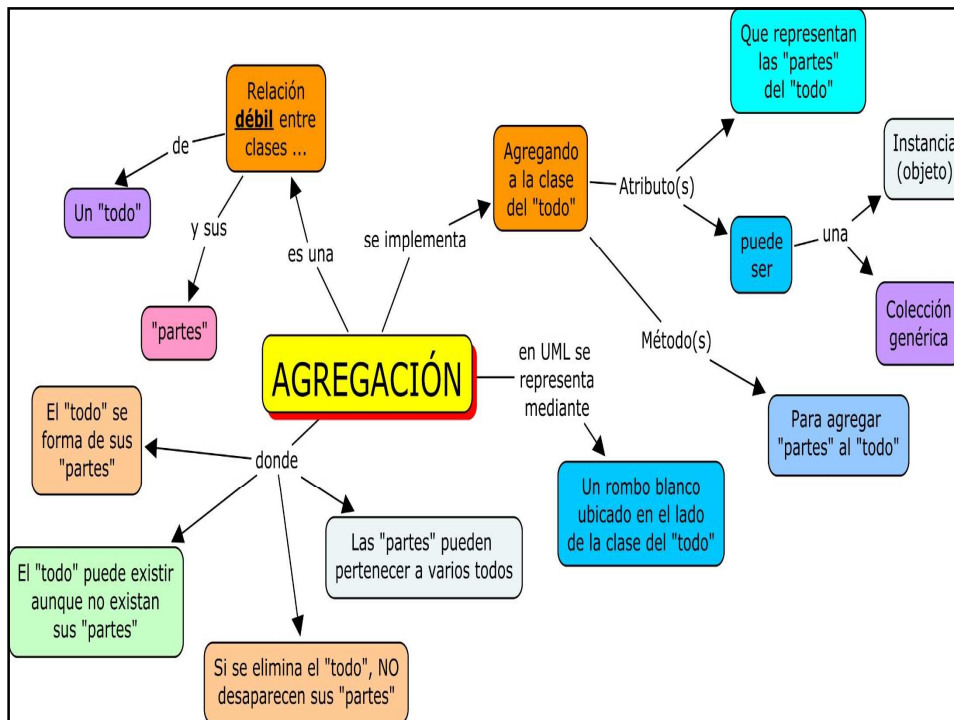
Usar...	Cuando...
Herencia (Es un...)	<ul style="list-style-type: none">• Se desee incorporar en la clase las variables, propiedades y métodos de otra clase.• Se desee especializar una clase (agregando características específicas).
Composición (Tiene un o unos...)	<ul style="list-style-type: none">• Se desee ocultar, o encapsular un objeto bajo una nueva interfaz.• Un objeto contenga otro objeto.

63

Agregación

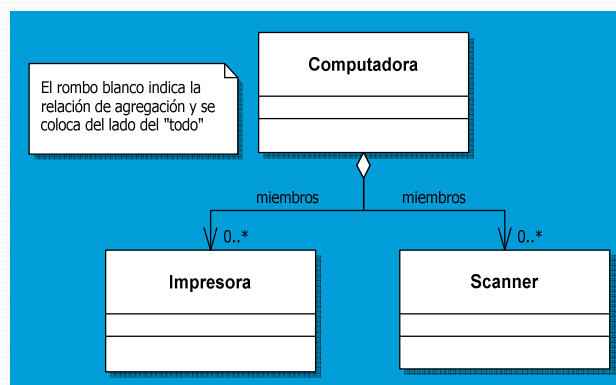
- Define una relación donde una clase se puede formar de otras clases
- Sin embargo, la existencia de objetos de dichas clases es independiente
- Se utiliza para modelar un "todo" y sus "partes" donde ...
 - El "todo" se forma agregando sus "partes"
 - Las "partes" pueden pertenecer a varios "todos"
 - Si se elimina el "todo" pueden seguir existiendo sus "partes"

64



Representación de la agregación

- Gráficamente se representa colocando un rombo blanco en el extremo de la clase constituida (parte del “todo”).



66

Reglas para que exista agregación

- Existen dos reglas para que se presente la relación de agregación entre dos clases:
 1. La clase del “todo” **DEBE** tener un atributo de tipo “parte”
 - a) *Un objeto cuando es relación 1..1*
 - b) *Una colección genérica si es relación 1..**
 2. La clase del “todo” **DEBE** tener un método para insertarle objetos de tipo “parte”

67

Modelo UML con varias agregaciones

- A un objeto de tipo “todo” pueden agregarse objetos de diferentes tipos de “partes”
- El “todo” tiene un atributo de cada tipo de “parte”
- El “todo” tiene un método para agregar cada tipo de “parte”
- El “todo” tiene varios iteradores (uno para cada tipo de “parte”).
- Los iteradores se implementan a través de propiedades de solo lectura

68

Reglas para que exista agregación (cont.)

- En la agregación, la clase del “todo” **NO** es necesario que tenga el destructor, ya que si se elimina un objeto del “todo” sus partes siguen existiendo porque pueden pertenecer a otros “todos”.
- En pocas palabras...

iii Una agregación es una composición sin destructor !!!

69

Composición vs. Agregación

Criterio de comparación	Composición	Agregación
Es una relación entre clases de un «todo» y sus «partes»	✓	✓
El «todo» depende de la existencia de sus «partes»	✓	✗
Las «partes» pueden pertenecer a varios «todos»	✗	✓
Si se elimina el «todo» también se eliminan sus «partes»	✓	✗
La clase del «todo» contiene atributo(s) que representa(n) las «partes»	✓	✓
La clase del «todo» contiene método(s) para insertar sus «partes»	✓	✓
La clase del «todo» tiene un destructor para eliminar sus «partes»	✓	✗

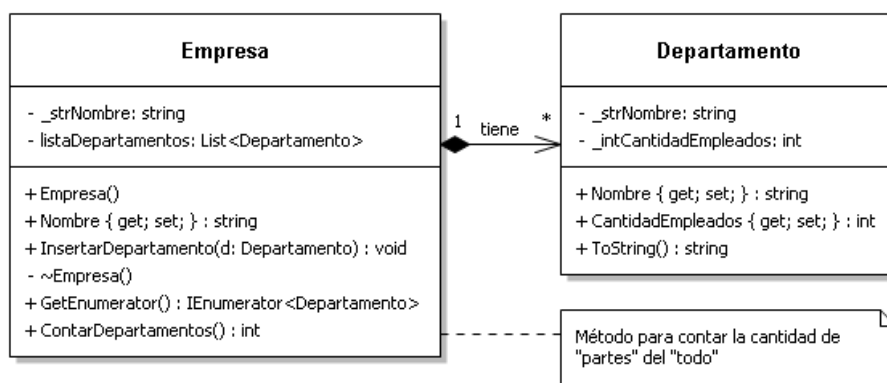
70

Composición vs. agregación (cont.)

iii Si a la clase del todo en una relación de composición se le elimina el destructor, entonces la relación se convierte en agregación !!!

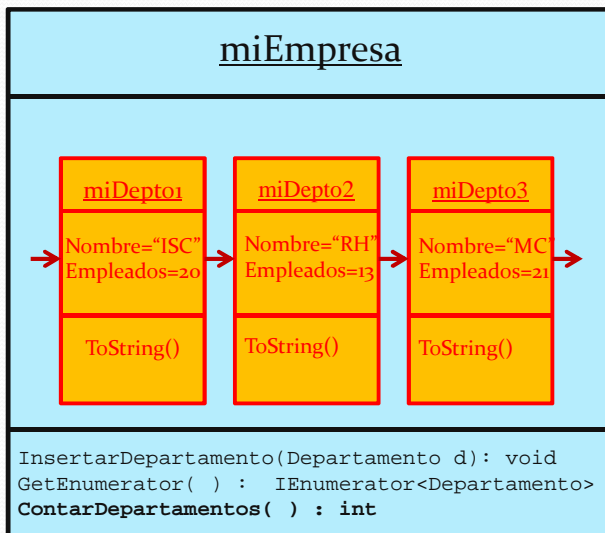
71

¿Cómo contar la cantidad de “partes” de un “todo” ?



72

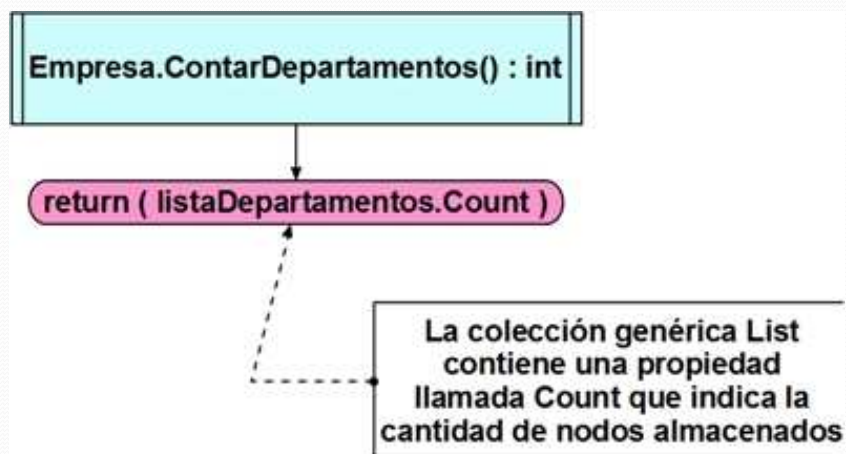
Contar los departamentos de una empresa



- Un objeto de la clase del "todo" (**miEmpresa**) tiene dentro una colección de objetos de la clase "parte" (**miDepto**)
- En este caso, el método devuelve el valor 3.

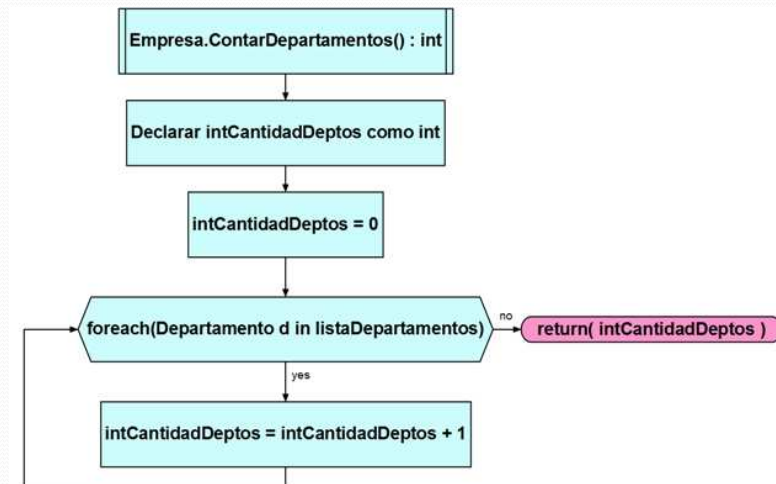
73

Diagrama de flujo para contar los departamentos de una empresa (opción 1)



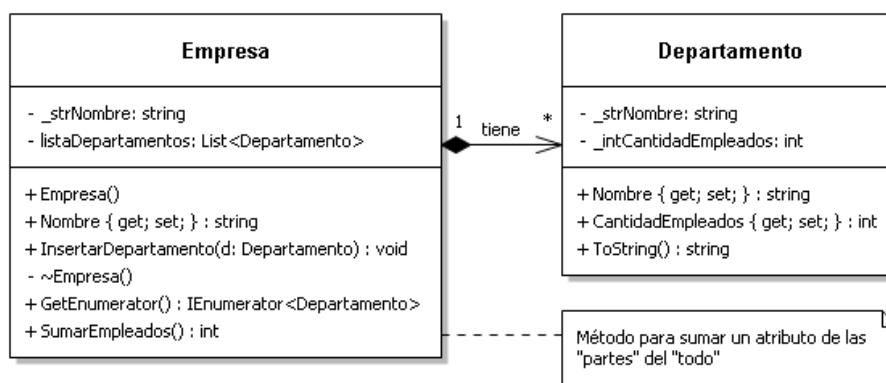
74

Diagrama de flujo para contar los departamentos de una empresa (opción 2)



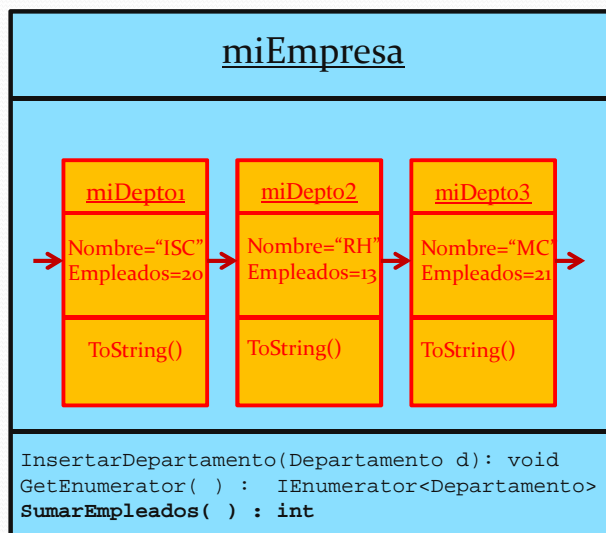
75

¿ Cómo sumar un atributo de las “partes” de un “todo” ?



76

Sumar la cantidad de empleados de los departamentos de la empresa

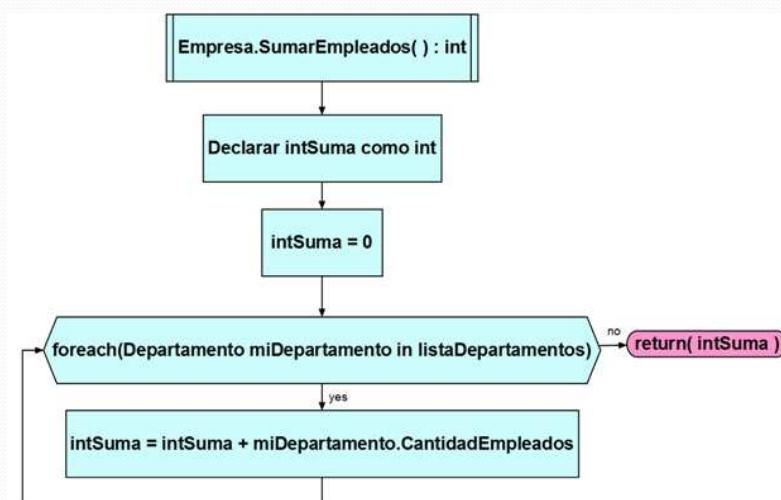


- El método recorre la lista de departamentos para acumular la cantidad de empleados

- En este caso, el método devuelve el valor 54

77

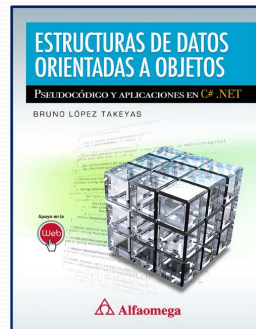
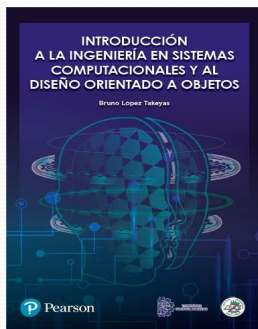
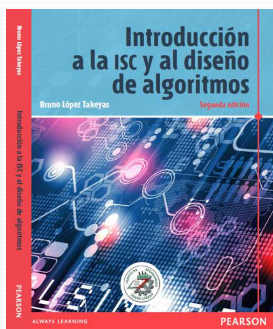
Diagrama de flujo para sumar la cantidad de empleados de los departamentos de la empresa



78

Otros títulos del autor

<http://www.itnuevolaredo.edu.mx/Takeyas/Libro>



✉ bruno.lt@nlaredo.tecnm.mx

 Bruno López Takeyas