TEMA 2

Clases y Objetos
(2ª parte)

Facultad de Informática

Universidad de Murcia

Índice

1.- Clases en Eiffel, C++ y C#

MARCO DE COMPARACIÓN (Modelo OO):

- ***Abstracción de tipos***:
  - Definición de atributos y métodos
  - Atributos de clase vs. Atributos de instancia
- ***Ocultación de información***:
  - Niveles de visibilidad
- ***Modularidad***:
  - Tipos de módulos que permite el lenguaje

Ejemplo definición de clase en **Eiffel**

```
class CUENTA
    creation abrir
    feature {ALL}                                          -- caracteristicas públicas
        titular : PERSONA;
        saldo : INTEGER;
        codigo: INTEGER;

        abrir (quien: PERSONA) is do                       -- rutina de creación
                    saldo:=0;
                    titular:=quien;
                    codigo:= codigos.nuevo_valor;          -- codigos función ONCE
                    !!ultOper
        end;

        reintegro (suma: INTEGER) is do                    -- rutina para sacar dinero
                    if puedo_sacar(suma) then saldo:=saldo-suma;
        end;

        ingreso (suma: INTEGER) is do                      -- rutina para ingresar dinero
                    saldo:=saldo+suma
        end;

        ver_ult_oper (n: INTEGER) is do … end;             -- visualiza las n ultimas oper.
        ….

    feature {NONE}                                         -- at y rutinas privadas
        ultOper: LIST[INTEGER];
        puedo_sacar (suma: INTEGER): Boolean is do
                    Result:= saldo>=suma
        end;
end
```

Clases en Eiffel

- **Abstracción de tipos**
  - **Atributos**: `saldo:INTEGER`
    - exportados en modo consulta (Principio de Acceso Uniforme)
    - Sólo modificables por los métodos de la clase aunque sean públicos
  - **Rutinas**:
    - procedimientos: `ingreso(suma:INTEGER)is do ...end`
    - funciones:

      `puedo_sacar(suma:INTEGER):BOOLEAN is do ...end`
  - **Variables de clase**:
    - Eiffel no tiene variables globales
    - Funciones **`once`** = El objeto se crea sólo una vez pero puede cambiar su valor

Función once

```
codigos: Contador is
  once     --devuelve siempre el mismo objeto Contador
      !!Result   --crea un objeto contador
  end
```

- El objeto contador que devuelve se puede modificar utilizando los métodos de la clase contador. Por ejemplo:

```
codigos.nuevo_valor
```

Siendo `nuevo_valor` un método de la clase `Contador` que incrementa el valor del contador (de tipo INTEGER) y devuelve ese nuevo valor.

Clases en Eiffel

- **Ocultación de información**

  Especificación de acceso a un grupo de características:
  - públicas: (por defecto) `feature {ALL}`
  - privadas: `feature {NONE}/feature{}`
  - exportadas de forma selectiva. `feature {A,B, ...}`

- **Modularidad**
  - El único módulo son las clases
  - **`Cluster`** = Agrupación de clases relacionadas pero no es parte del lenguaje sino que depende del entorno de desarrollo
  - Para hacer uso de un cluster se debe decir al entorno Eiffel

Eiffel y Ocultación de Información

```
class ICE1 feature
   at1: INTEGER;   //Público
   ...
end
class TEC1 feature
   atrib1: ICE1;
   atrib2: INTEGER;
   una_rutina (p: INTEGER) is do
       atrib2:= p;
       atrib2:= atrib1.at1;
       at1:= p;
       atrib1.at1:= p;
   end;
end
```

Exportación de atributos en modo **consulta** (=función)

-- 👁 **No error**

Modificación:

```
atrib1.setAt1(p);
```

Ejemplo de definición de clase en C++ (interfaz)

// **Cuenta.h**, definición del TAD Cuenta

```
class Cuenta {
  public:
        Cuenta (Persona *quien)   {saldo=0;
                                   titular=quien;
                                   codigo = nuevoCodigo();
                                   ultOper = new lista<int>;}

        void        reintegro(int suma);
        void        ingreso(int suma);
        int         verSaldo();
        void        verUltOper(int n);
        static int  nuevoCodigo();      //devuelve el ultimoCodigo y lo incrementa

  private:
        Persona  * titular;
        int         saldo;
        int         codigo;
        static int  ultimoCodigo;       //variable de clase
        lista<int> * ultOper;

        bool        puedoSacar(int suma)  {return (saldo >=suma);}
};
```

Ejemplo de definición de clase C++ (implementación)

// **cuenta.cpp**, Definición de las funciones de la clase

```
#include "cuenta.h"

// inicializa la variable de clase
int Cuenta :: ultimoCodigo = 0;

void    Cuenta :: reintegro (int suma) {
            if puedoSacar(suma) saldo=saldo-suma;
}

void    Cuenta :: ingreso (int suma) {
            saldo=saldo+suma;
}

int     Cuenta :: verSaldo () {
            return saldo;
}

void    Cuenta :: verUltOper(int n) {
            ...
}

static int Cuenta :: nuevoCodigo() {
            return (ultimoCodigo++);
}
```

Clases en C++

- **Abstracción de tipos**
  - atributos.
    - No pueden ser exportados en modo consulta
    - Tipos primitivos y punteros
  - todas las rutinas tienen un valor de retorno.
  - Atributos y métodos de clase (`static`)
- **Ocultación de información**
  - Especificación de acceso para un grupo de miembros:
    - **`public`**: un cliente puede consultarlo y ¡¡modificarlo!!
    - **`private`**: sólo accesible dentro de la clase
  - Clases ***amigas***: Se le concede acceso TOTAL a la clase amiga

Clase "amiga"

```
class B {
        friend class A;
        private:
                int  i;
        public: ...
}
```

En la clase *A*,

```
B *ob
ob -> i = 89
```

```
class NodoArbol {
        friend class Arbol;
        private:
                int dato;
                 NodoArbol decha;
                NodoArbol izda;
                …
};
class Arbol{
        private:
                NodoArbol *raiz;
                ...
        … raiz ->dato=50; ...
};
```

La amistad no es hereditaria ni transitiva

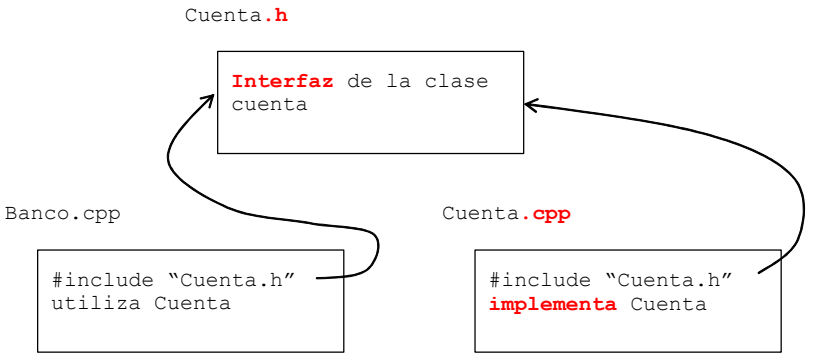
Clases en C++

- **Modularidad**
  - Definición de nuevos tipos: clases (**`class`**) y estructuras (**`struct`**)
  - Una estructura equivale a una clase con todos los miembros públicos por defecto (se puede usar `private`)
  - **`namespace`**: mecanismo para agrupar datos, funciones, etc. relacionadas dentro de un espacio de nombres separado

```
namespace Parser {
  double term(bool obten){/*multiplica y divide */}
  double expr(bool obte) {/*suma y resta */}
  …
}
```

Para usar una función: `Parser::expr(…);`

Ficheros para la especificación de la clase Cuenta



- **`#include`** para importar ficheros cabecera
  - toda la información se use o no
  - pueden ocurrir dependencias circulares que debe solucionar el programador en lugar del compilador

Ejercicio

Intenta escribir en C++ la siguiente clase Eiffel,

```
class CELOSO
    feature {NONE}
        esposa: MUJER;
    feature {MECANICO}
        coche: AUTOMOVIL
    …
end
```


Ejemplo de definición de clase en C#

```
using System;
namespace Banco.GestionCuentas{
  public class Cuenta {
    private Persona titular;
    private float saldo;
    private int codigo;
    private static int ultimoCodigo;
    private Operacion[] ultOper;
     Cuenta (Persona quien){
          saldo=0;
          titular=quien;
          codigo=nuevoCodigo();
          ultOper=new Operacion[20];
     }
     static Cuenta(){          //constructor de clase
          ultimoCodigo=1;
     }
   public float Saldo{
        get{
            return saldo;
        }
   }
    public virtual void Reintegro(float suma){
          if puedoSacar(suma) saldo-=suma;
    }
    public virtual void Ingreso(float suma){
          saldo+=suma;
    }
    public void VerUltOper(int n) { ... }
    public static int NuevoCodigo(){
          return ultimoCodigo++;
    }
    private boolean puedoSacar(float suma){
          return (saldo >=suma);
    }
  }
}
```

Clases en C#

- **Abstracción de tipos**
  - especificación de atributos y métodos igual que Java
  - Atributos y métodos de clase (`static`)
- **Ocultación de Información**
  - `public` y `private` igual que Java y C++
  - `internal`: accesible desde el código del **ensamblado** (librería o ejecutable)
  - Proteger el acceso a los atributos mediante la definición de **propiedades** (principio de Acceso Uniforme)
  - Se escribe el código que se ejecutará en el acceso para lectura (**`get`**) y modificación (**`set`**) de un atributo privado de igual nombre que la propiedad.

Definición de propiedades en C#

```
<tipoPropiedad> <nombrePropiedad>
{
        set
        {
                <códigoEscritura>
        }
        get
        {
                <códigoLectura>
        }
}
```

- Puede ser una propiedad de sólo lectura (sólo se define el `get`) o de sólo escritura (sólo se define el `set`).

Ejemplo de propiedad C#

```
class Cuenta{
  …
  public double Saldo
   {
        get
        {
         double total = 0;
         for …
                total = total + ultOper[i];
          return total;
        }
        set
        {
          ultOper[indice]= value;
        }
   }
}
```

- Acceso a una propiedad como si estuviéramos accediendo a uno de los campos del objeto

```
Cuenta cta = new Cuenta();
cta.Saldo = 300;
```


Clases en C#

- Modularidad
  - Definición de nuevos tipos: clases, estructuras e interfaces
  - Agrupación de tipos de datos en espacios de nombres (equivalente a los paquetes de Java)

```
namespace nombreEspacio{
 …//tipos pertenecientes al espacio de nombres
}
```

  - Para utilizar un tipo definido en un espacio de nombres:
    - **using**: para importar los tipos definidos en un espacio de nombres
    - Calificar el tipo utilizando la notación punto.

Exportación de características entre clases relacionadas

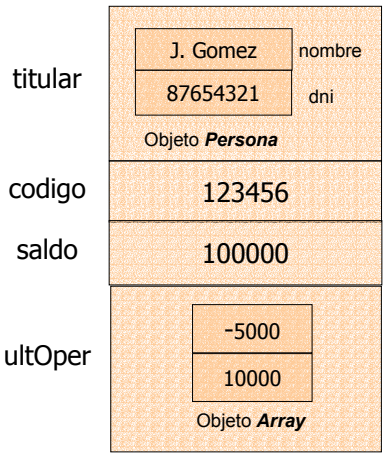
- ¿Cómo exportar características a un conjunto de clases sin violar la regla de ocultación de información?
- Soluciones:
  - Paquetes Java
  - Clases amigas C++
  - Exportación selectiva Eiffel
- Los paquetes añaden complejidad al lenguaje.
- Con las soluciones de Java y C++ se corre el riesgo de perjudicar la reutilización.

Ejercicio

[Examen 2/12/2005] Dada la implementación parcial de la clase Bola en Eiffel que representa una bola en un juego de billar, especificar una clase Java equivalente, explicando TODAS las decisiones y los problemas encontrados.

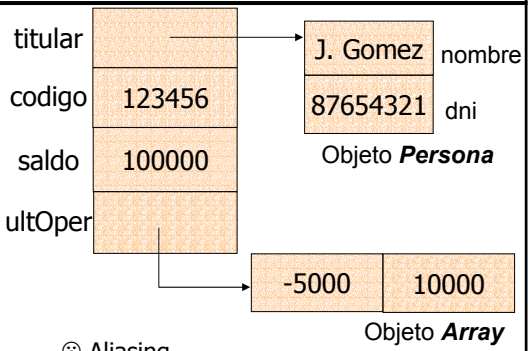
```
class Bola
feature
    direccion: Real --en radianes
    energia: Real
feature {NONE}
     calculaCentro: Punto is do …end
feature {MesaBillar}
     region:Rectangulo
…
end
```

2.- Objetos
Subobjetos vs. Referencias



☹ Desperdicia memoria

☹ No permite compartir



☹ Aliasing

☺ Compartición (integridad referencial)

☺ Estructuras recursivas (Autoreferencias)

☺ Más eficiente manejo obj. complejos

☺ Obj. se crean cuando se necesitan

☺ Soporte para el polimorfismo

Tipos expandidos en Eiffel

- Los posibles valores de una entidad son los objetos mismos en lugar de referencias
- No necesitan instrucciones de creación

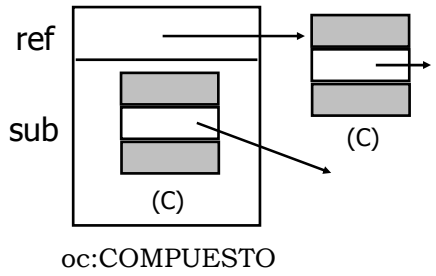
```
expanded class PERSONA
…
end;

p: expanded PERSONA
```

- Este mecanismo añade la noción de **objeto compuesto**.
- **¿Para qué necesitamos tipos expandidos?**
  - Modelar con realismo objetos del mundo real
  - Ganar en eficiencia (tiempo y espacio)
  - Para los valores de los tipos básicos

Objetos Compuestos

Un objeto compuesto, *oc*, es aquel que tiene uno o más campos que son objetos (sub objetos).

```
class COMPUESTO feature
        ref: C;
        sub: expanded C
                …
end
```



oc:COMPUESTO

Semántica referencia vs. Subobjetos

- En **C++**, semántica referencia asociada al tipo "puntero" (en otro caso semántica almacenamiento)

```
Persona *titular;
Persona titular;
```

- En **Eiffel**, tipos **referencia** y tipos **expandidos**

```
titular: PERSONA
titular: expanded PERSONA
```

- En **Java**, semántica referencia para cualquier entidad asociada a una clase, no hay objetos embebidos.

```
Persona titular;
```

- En **C#**, semántica referencia para cualquier entidad asociada a una clase. Las instancias de las estructuras (`struct`) no son referencias (equivale a los tipos expandidos de Eiffel).

3.- Métodos y mensajes

- Definición de métodos:
  - Instrucciones en Eiffel
  - Paso de parámetros
  - Instancia actual
  - Valor de retorno
- Invocación de métodos: Mensaje

Instrucciones Eiffel:

i) **Asignación**

```
ox := oy
```

Semántica:

- COPIA: cuando ox y oy tienen semántica almacenamiento
- COMPARTICIÓN: cuando ox y oy tienen semántica referencia

ii) **Iteración**

```
from "inicialización" until "condición terminación"
loop
  "Cuerpo"
end
```

Instrucciones Eiffel

iii) **Condicional**

```
if c1 then s1
elseif c2 then s2
...
elseif cn-1 then sn-1
else sn
end;
```

"switch"

```
inspect var
when v1 then s1
...
when vn-1 then sn-1
[else sn]
end;
```


Paso de parámetros

- **C++**
  - Se utilizan los punteros para simular argumentos por referencia con argumentos por valor
  - El programador tiene distinguir entre *p y &p para referenciar y deferrenciar.
- **Java**
  - siempre paso por valor tanto tipos simples como referencias
  - los objetos se pasan por referencia automáticamente (sin el lío de añadir *p o &p)
  - Se pueden declarar como `final` (el valor del parámetro no cambiará mientras el método se ejecuta).
- **Eiffel**
  - paso por valor de las referencias
  - El cuerpo de la rutina no puede cambiar la referencia, de esta forma se puede utilizar cualquier expresión como argumento real.
- **C#**
  - Cuatro tipos diferentes de parámetros (entrada, salida, por referencia y de número indefinido)

Instancia actual

- Cuando un mensaje no especifica al objeto receptor la operación se aplica sobre la instancia actual.
- Es posible referenciar a la instancia actual
  - Eiffel: `Current`
  - C++, Java y C#: `this`
- La utilización de la palabra reservada que hace referencia a la instancia actual es opcional.

Valor de retorno de una función

- Técnicas utilizadas más comunes:

1) **Instrucción explícita: `return expr`**
   (C)
   - Código poco estructurado
   - Necesidad de variables auxiliares
   - ¿Qué sucede si no se devuelve nada?

2) **nombre de la función es un identificador de variable**
   (Pascal)
   - Ambigüedad (mismo nombre para una función y para una variable).

Result vs. return

- **Result**
  - Variable predefinida para denotar el resultado de la función en Eiffel
  - Se trata como una entidad local y se inicializa con el valor por defecto apropiado
  - Este valor siempre está definido aunque no aparezca en el cuerpo de la función.
  - Evita los problemas anteriores
- **return**
  - C++, Java y C#
  - Se tiene que devolver una expresión del mismo tipo que se indica en la función.
  - En C++ es posible no poner el `return` (en Java y C# daría un error en tiempo de compilación)

Ejemplo: Clase Punto (x,y)

```
class PUNTO feature
    x,y: REAL;                          -- Coordenadas cartesianas

    rho: REAL is do                     -- Coordenada polar
            Result:= sqrt(x^2 + y^2)
    end;
    theta: REAL is do                   -- Coordenada polar
            Result:= atan2(y,x)
    end;

    distancia (p: PUNTO): REAL is do
            if p /= Current then Result:= sqrt((x-p.x)^2 + (y - p.y)^2)
    end;
    trasladar (a, b: REAL) is do
            x:= x + a;
            y:= y + b
    end;
    escalar (factor: REAL) is do … end;
    rotar (p: PUNTO; angulo: REAL) is do … end;
end
```

else la distancia de un pto a él mismo es cero.

Result (Eiffel) vs. **return** (Java)

Sintaxis de los mensajes

- **C++**
  - '`->`' y '`.`' en función de que sea o no un puntero, respectivamente.
  - Viola el Principio de ocultamiento de la información
  - hay que conocer los detalles de implementación para acceder a los miembros
- **Java, Eiffel y C#**
  - Notación punto
  - Principio de Acceso Uniforme en Eiffel y C#

Ejemplos. Sintaxis mensajes

- C++

```
Cuenta c; Cuenta *ptroCta;
c.reintegro(1000);
ptroCta->reintegro(1000);
```

- Java

```
Cuenta c;
c.reintegro(1000);
```

- Eiffel

```
c:Cuenta; c2:expanded Cuenta;
   c.reintegro(1000);
s:=c2.saldo;
```

Acceso Uniforme

Características de operador

• **Eiffel** ofrece la posibilidad de declarar **operadores:**

```
class REAL feature
    infix "+" (other: REAL): REAL is do … end;  --suma
    infix "-" (other: REAL): REAL is do … end;  --resta
    prefix "- " : REAL is do ... end;           --negación
    …
end
```

mecanismo para reconciliar ***consistencia*** (un único mecanismo=mensaje) y ***compatibilidad*** con las notaciones tradicionales.

• **Java no** ofrece la posibilidad de usar los operadores (+,- ,*, /, ...) como nombres de funciones:

```
total.setValue(shipChg.mas(unitPrice.por(quantity)));
```

en lugar de:

```
total = unitPrice * quantity + shipChg;
```


4.- Creación de Objetos

- Declaración ≠ Creación
- Mecanismo explícito de creación de objetos

  (A) Eiffel: instrucción de creación, ! !

  (B) C++, Java y **C#**: **new**

- Constructores: deja el objeto en un estado válido
  - Diferentes formas para inicialización de objetos según el lenguaje
    - **C++, Java y C#** ***constructores*** con el nombre de la clase que no se pueden invocar una vez que el objeto es creado.
    - **Eiffel** permite tener ***rutinas de creación*** que se pueden utilizar como rutinas "normales" (reinicializar un objeto).

(A) Creación de Objetos en Eiffel

```
!tr!e.rc(..)
```

donde

tr: tipo referencia (opcional)

e: identificador de una entidad

rc: rutina de creación (opcional)

Ejemplos:

!!oc

!!oc.abrir (p)

!Cuenta_Ahorro!oc

!Cuenta_Ahorro!oc.abrir(p)

Creación de Objetos en Eiffel

Supuesta la declaración `e:T`

a) T no tiene rutina de creación . `!!e`

1) Crea una nueva instancia de ***T.***
2) Inicializa los campos de la instancia con los valores por defecto.
3) Conecta ***e*** a la instancia creada.

b) T tiene rutina de creación. `!!e.rc1(...)`

1) Crea una nueva instancia de ***T***
2) Inicializa los campos de la instancia con los valores por defecto.
3) Se aplica sobre la instancia la rutina de creación ***rc1,*** **de modo que quede en un estado consistente**
3) Conecta ***e*** a la instancia creada

Inicialización de objetos por defecto (Eiffel)

TIPO	VALOR
Referencia	Void
BOOLEAN	False
INTEGER	0
REAL, DOUBLE	0
CHARACTER	carácter nulo

Creación de Objetos en Eiffel

Ejemplo:

```
class CUENTA
    creation abrir,nothing
    feature
      abrir (quien: PERSONA) is do
            ...
      end;
      nothing is do end;
end;
```

¿si me sirve la inicialización por defecto?

- Una una rutina de creación puede ser **privada**, de manera que sólo se puede utilizar en las llamadas de creación.

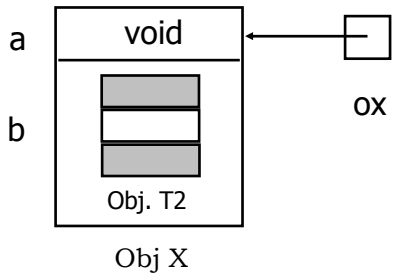
Tipos expandidos y Creación

Para entidades de tipo expandido no es necesario crear objetos, el espacio se asigna en tiempo de compilación.

Sea la clase:

```
class X feature
        a: T1
        b: expanded T2
        …
-- T1 y T2 tipos referencia
end
```

Si ox:X ¿!!ox?



Obj X

Tipos expandidos y creación

¿Son válidas las siguientes declaraciones de clases?

```
class X feature
   a1: REAL;
   a2: Y;
   a3: Z;
   …
end
```

```
expanded class Y feature
        a3: expanded X;
        a4: STRING;
        …
end
```

(B) Creación de objetos en C++

- Inicialización implícita mediante **CONSTRUCTORES** que realizan la inicialización después de que los objetos con creados.
- Un **constructor:**
  - procedimiento especial con el mismo nombre que la clase
  - Se invoca siempre que se crea un objeto de la clase:
    i) cuando se declara una variable
    ii) con objetos creados dinámicamente con **new**
  - No tiene valores de retorno
  - Permite sobrecarga

Creación de Objetos en C++. Ejemplo

```
class Complejo {
      public:
        float   real;
        float   imag;
        //Constructor con valores por defecto
        Complejo (float=0, float=0);
...
}
```

NO es posible en Java
constructor por defecto es un constructor sin argumentos

Sea la declaración Complejo *c1, c2, c3; entonces

c1 = new Complejo ()	c1=(0,0)
c2 = new Complejo (3.14159)	c2=(3.14159,0)
c3 = new Complejo (3.14159,2.4)	c3=(3.14159,2.4)

...en C++

- Posibilidad de definir **DESTRUCTORES** que se ejecutan automáticamente cada vez que se libera memoria.
  - Al acabar un procedimiento (variables automáticas)
  - Al aplicar el operador **delete () (variables dinámicas)**

```
class Complejo {
        public:
                ~Complejo () {...};
                 Complejo(float pr = 0.0; float pi = 0.0);
                …
end
```

  - No tiene argumentos y tampoco regresa un valor
  - No destruye el objeto, ejecuta "trabajos de terminación"

(D) Constructores en C#

- Igual que en Java:
  - Igual nombre de la clase
  - Sin valor de retorno
  - Sobrecarga
  - Constructor por defecto
- Constructor de clase:
  - Inicializa las variables de clase
  - Llama automáticamente la primera vez que se accede al tipo

```
static Cuenta(){
   nextNumero = 1;
}
```

`this` en los contructores (Java y C#)

- Invocación explícita a otro constructor de la clase
  - **Java**

```
class A {
    int total;
    public A(int valor){
      this(valor, 2)
    }
    public A(int valor, int peso) {
           total = valor*peso;
    }
  }
```

  - **C#**

```
class A {
    int total;
    A(int valor): this(valor, 2){}
    A(int valor, int peso) {
           total = valor*peso;
    }
  }
```

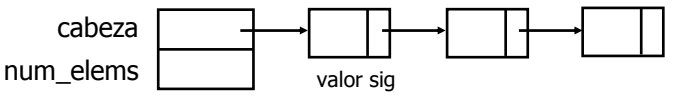

Ejercicio: Traducir a Eiffel el siguiente código Pascal

```
TYPE   tipoLista= ^nodo;
       nodo=  RECORD
                valor: INTEGER;
                sig: tipoLista
              END

p,q: tipoLista; n: INTEGER;

readln(n);
p:=nil;
WHILE n>0 DO BEGIN
       new(q); q^.sig:=p; p:=q;
       q^.valor:=n; n:=n-1;
END
```

Clase NODO_ENTERO en Eiffel

```
class NODO_ENTERO
feature {LISTA_ENTEROS}
    valor: INTEGER;
    sig: NODO_ENTERO;

    cambiar_valor (v: INTEGER) is do
        valor:= v;
    end;

    cambiar_sig (s:NODO_ENTERO) is do
        sig:= s
    end;
end
```


Clase LISTA_ENTEROS en Eiffel

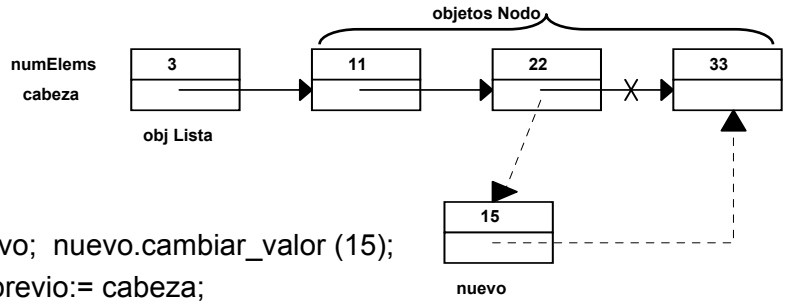



```
class LISTA_ENTEROS feature          -- clase Eiffel
    cabeza: NODO_ENTERO;
    num_elems: INTEGER;

    valor (i: INTEGER): INTEGER is do … end;
    cambiar_valor (i: INTEGER, v: INTEGER) is do … end;
    insertar (i: INTEGER, v: INTEGER) is do … end;
    eliminar (i: INTEGER) is do … end;
    buscar (v: INTEGER): INTEGER is do … end;
    ….
end
```

Rutina "insertar nodo en lista lineal"

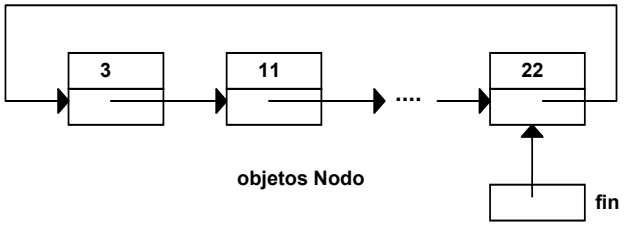
```
insertar (i: INTEGER, v: INTEGER) is
        local   nuevo, previo: NODO_ENTERO;
                j: INTEGER;
        do
                !!nuevo;
                nuevo.cambiar_valor(v);
                if i = 1 then
                        nuevo.cambiar_sig (cabeza);
                        cabeza:=nuevo;
                else
                        from j:=1; previo:=cabeza;
                        until j=i-1
                        loop
                                j:= j+1;
                                previo:=previo.sig
                        end
                        nuevo.cambiar_sig (previo.sig);
                        previo.cambiar_sig (nuevo);
                end
                num_elems:= num_elems + 1;
        end;
end;
```


Ejemplo: Llamada insertar(3,15)



!!nuevo; nuevo.cambiar_valor (15);
j=1; previo:= cabeza;
j=2; previo:= previo.sig;
nuevo.cambiar_sig (previo.sig)
previo.cambiar_sig (nuevo)
num_elems:= num_elems + 1

Lista Circular de enteros en C++




```
class Nodo {
        friend class Lista_circular;
                Nodo (int i)              {valor=i; sig=this;}
                Nodo (int i, Nodo *n)     {valor=i; sig=n;}
        int     valor;
        nodo  * sig;
};
```


Lista Circular de enteros en C++

```
//Lista_circular.h

class Lista_circular {
        Nodo  * fin;

public:
                Lista_circular ()       {fin= new nodo (0);}

        int     vacio ()                {return fin == fin -> sig;}
        void    inserta (int);
        void    entrada (int);
        int     extrae ();
};
```

...Lista circular de enteros en C++

```
// Lista_circular.cpp
// inserta un elemento al frente
void Lista_circular :: inserta (int x) {
        fin -> sig = new Nodo (x, fin -> sig)}

// inserta un elemento por la cola
void Lista_circular :: entrada (int x) {
        fin -> valor = x;  fin = fin -> sig = new Nodo (0,fin -> sig)}

// elimina el elemento del frente de la lista y devuelve su valor
int Lista_circular :: extrae () {
        if (vacio () ) return 0;
        Nodo *frente = fin -> sig;
        fin -> sig = frente -> sig;
        int x = frente -> valor;
        delete frente;
        return x;
}
```


Críticas a esta representación (Cap23. Meyer)

- **Redundancia de código entre los métodos:**
  Bucles casi idénticos
  Ejemplo: buscar (v:INTEGER):INTEGER is do ... end
  sustituir (i:INTEGER; v:INTEGER) is do ... end

- **Ineficiencia:**
  Para cualquier operación hay que volver a recorrer la lista
  Ejemplo: 1º l.buscar(valor)- >devuelve la posición (pos)
  2º l.sustituir(nuevo_valor, pos)

La clase Lista está mal diseñada -> Clase pasiva

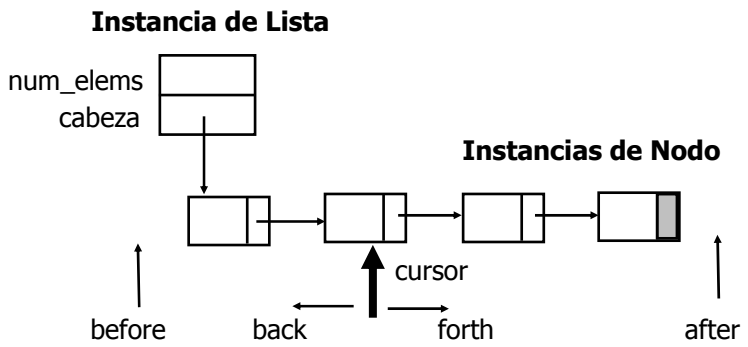
Soluciones para la representación de listas

- **buscar** podría devolver la referencia al objeto en lugar de la posición
  - violación ocultamiento de la información
- Proporcionar rutinas que abarquen combinaciones comunes de operaciones: búsqueda y sustitución, búsqueda e inserción, ...

  El numero de variantes es enorme
  - Cada nueva operación supone un cjto nuevo de variantes
  - Rutinas muy parecidas
- Cursor: Listas Activas. Recordando donde se hizo la última operación

Listas activas

- Además del estado incluimos la noción de ***posición activa*** o ***cursor***
- La interfaz permitirá que los clientes trasladen el cursor de manera explícita.



Listas activas

- Ordenes básicas para manipular el cursor:
  - **`start`** y **`finish`**, para trasladar el cursor a la primera y última posición
  - **`forth`** y **`back`**, trasladar a la posición siguiente y anterior
  - **`go(i)`**, trasladar a la posición i
- Consultas relativas a la posición del cursor:
  - **`before`** = posición a la izquierda del primero
  - **`after`** = posición a la derecha del último
  - **`index`** = devuelve la posición actual
  - **`is_first`**
  - **`is_last`**

Listas activas

- La manipulación de la lista se vuelve mas simple porque no se preocupan por la posición
- Se limitan a actuar sobre la posición actual

¡Desaparecen todos los bucles innecesarios!

Ejemplo:

antes	**ahora**
`eliminar(i)`	`l.go(i)` `l.remove`

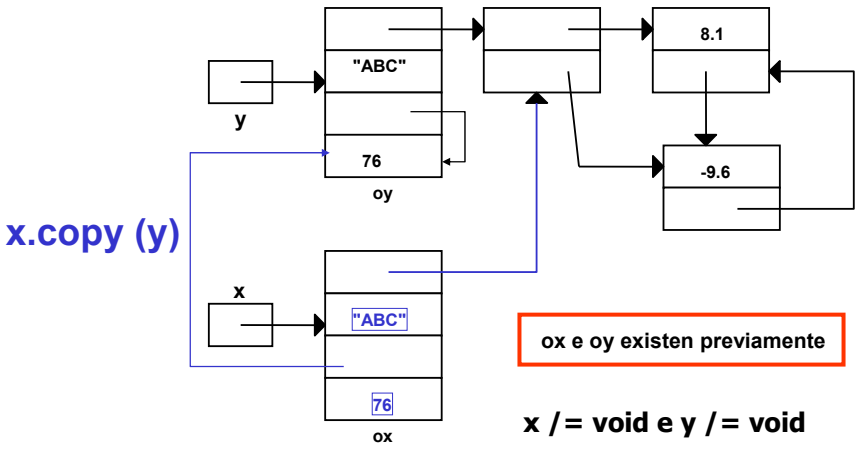
- Es necesario establecer de **manera precisa** lo que sucede con el cursor después de cada operación.

Ejemplo de uso de las Listas activas

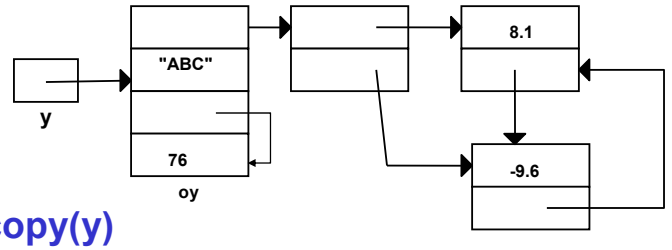
```
l: Lista_Enteros;
m, n: INTEGER;
...
l.start; l.search(m);
if not l.after then l.put_right(n) end;
...
l.start;
l.search(m); l.search(m); l.search(m);
if not l.after then l.remove end;
...
l.go(i);l.put_left(m);
```


6.- Operaciones sobre referencias:

(A.1) Copia superficial de un objeto en Eiffel

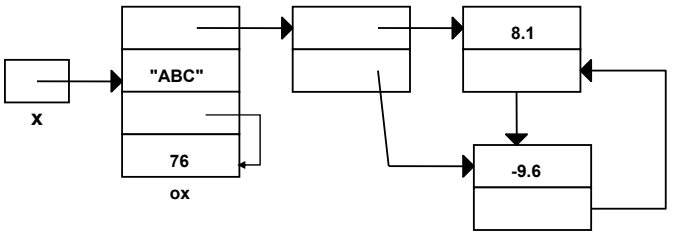


(A.2) Copia profunda de un objeto en Eiffel



x.deep_copy(y)

ox e oy existen previamente



(B) Operación `clone´ en Eiffel

x:= **clone** (y)
x:= **deep_clone** (y)

- **Crea** un nuevo objeto que es una copia idéntica de uno ya existentes.
- Combinamos con la **asignación** para conectar el objeto *ox*.
- Equivale a:

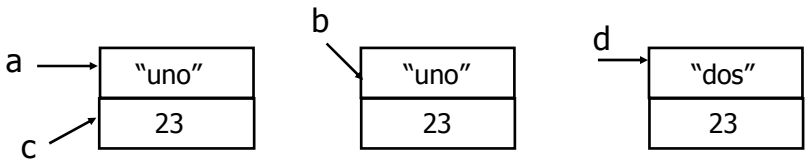
```
!!x
x.copy(y)
```

¿Ofrece alguna ventaja en relación a *copy*?

Conexión de entidades x:=y

Tipo de x \ Tipo de y	**REFERENCIA**	**EXPANDIDO**
REFERENCIA	Conexión de referencia	x:= clone (y) (duplicado)
EXPANDIDO	x.copy(y) (copia, falla si y es vacio)	x.copy(y)

(C) Igualdad de objetos



- **Igualdad entre referencias (Identidad)**
  a=c {true} a=b {false}
- **Igualdad entre objetos**
  equal(a,b) {true} equal(a,d) {false}
- De lo que se deduce que:

a = b ⇒ equal(a,b)
equal(a,b) ⇏ a=b

Igualdad superficial y profunda de objetos

x:=clone(y) ⇒ equal (x,y)

x.copy(y) ⇒ equal (x,y)

x:= deep_clone(y) ⇒ deep_equal(x,y)

¿Una igualdad profunda implica una igualdad superficial?

Igualdad de entidades x=y

Tipo de x \ Tipo de y	**REFERENCIA**	**EXPANDIDO**
REFERENCIA	Compara referencias	equal (x,y)
EXPANDIDO	equal (x,y)	equal (x,y)

7.- Genericidad

- La genericidad sólo tiene sentido en un lenguaje con comprobación estática de tipos.
- Tanto Eiffel como C++ como C# son lenguajes tipados e incluyen la genericidad como elemento del lenguaje
- Varía la sintaxis:
  - Eiffel: `class Pila[T]`
  - C++: `template<class T> Pila`
  - C#: `class List<T>`

Genericidad en Eiffel

```
class PILA[T]              -- G es el parámetro genérico formal
   feature {all}
        count: INTEGER;
        empty: BOOLEAN is do .. end;
        full:  BOOLEAN is do .. end;
        put (x:T) is do .. end;
        remove is do .. end;
        item: T is do .. end;
end.
```

Genericidad en C++

- **Definición de una clase genérica = Templates**

```
template <class T> class Pila {
  private:
    T* top;
    int count;
  public:
    Pila (int capacity);
    void push(T a);
    T pop ();
    int size();
};
```

```
Pila<char>  pc(100);
        // pila de caracteres
Pila<Punto> pp(20);
        // pila de puntos
```

- **Crítica:** por cada tipo que se pasa el compilador replica el código haciendo una simple sustitución de texto. Esto afecta: al tiempo de compilación, tamaño del código generado, tiempo y espacio de ejecución.

Genericidad en C#

```
public class Pila<T> {
  private T[] elementos;
  private int cantidad;

  public Pila(int capacidad){
      this.cantidad = 0;
      this.elementos = new T[capacidad];
  }
  public void Añadir(T valor){
      elementos[cantidad++] = valor;
  }
  public T Extraer(){
      return elementos[--cantidad];
  }
  public bool EstaVacia{
      return cantidad == 0;
  }
}
```