TEMA 2

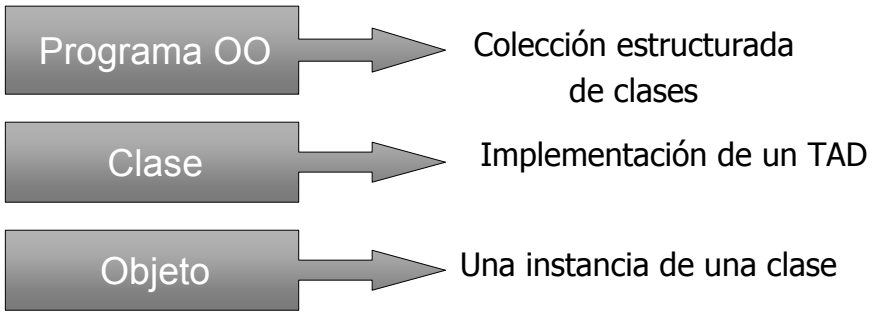
Clases y Objetos

Facultad de Informática

Universidad de Murcia

Índice

1.- Introducción



Los objetos se comunican mediante **mensajes**

2.- Clases

- **DEFINICIÓN:** Implementación total o parcial de un TAD
- Entidad sintáctica que describen objetos que van a tener la misma estructura y el mismo comportamiento.
- **Doble naturaleza**: Módulo + Tipo de Datos
  - **Módulo** (*concepto sintáctico*)
    - Mecanismo para organizar el software
    - Encapsula componentes software
  - **Tipo** (*concepto semántico*)
    - Mecanismo de definición de nuevos tipos de datos: describe una estructura de datos (objetos) para representar valores de un dominio y las operaciones aplicables.

Ejemplo Modula2: Módulo ≠ Tipo

```
DEFINITION MODULE Pila;
   EXPORT QUALIFIED PILA, vacia, pop, push, tope;

   TYPE PILA;

   PROCEDURE vacia(pila:PILA): BOOLEAN;
   PROCEDURE nuevaPila: PILA;
   PROCEDURE pop (VAR pila:PILA):INTEGER;
   PROCEDURE push (VAR pila:PILA; valor:INTEGER);
   PROCEDURE tope (VAR pila:PILA):INTEGER;

END Pila;
```


Especificación separada de la implementación

```
IMPLEMENTATION MODULE Pila;
   TYPE PILA = POINTER TO Node;
        Node = RECORD
           valor:INTEGER;
           siguiente:PILA;
        END;
   PROCEDURE pop (VAR pila:PILA):INTEGER;
   VAR rslt:INTEGER; tmp:PILA;
   BEGIN
        rslt:=0;
        IF (pila <>NIL)
        BEGIN
           rslt:=pila^.valor;
           tmp:=pila;
           pila:=pila^.siguiente;
           delete(tmp);
        END;
        RETURN rslt;
   END pop;
   ...
END Pila;
```

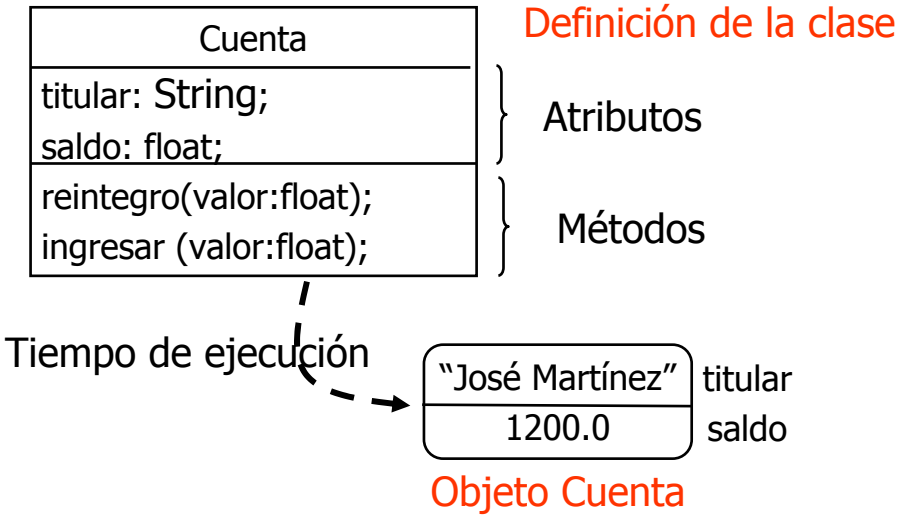
Componentes de un clase

- **Atributos**
  - Determinan una estructura de almacenamiento para cada objeto de la clase
- **Rutinas (Métodos)**
  - Operaciones aplicables a los objetos
  - Único modo de acceder a los atributos

Ejemplo: Al modelar un banco, encontramos objetos "*cuenta*".
Todos los objetos "*cuenta*" tienen propiedades comunes:
- atributos: *saldo*, *titular*, ...
- operaciones: *reintegro*, *ingreso*, …

Definimos una clase CUENTA.

Ejemplo: Clase Cuenta



Relaciones entre clases

- **Clientela**

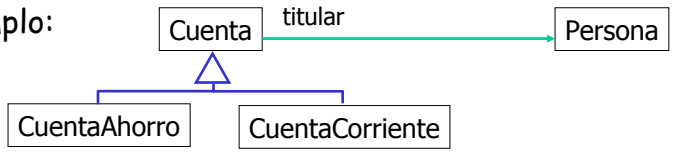
```
class Cuenta{
        Persona titular;
…
}
```

```
class CuentaAhorro extends Cuenta{
…
}
```

- **Herencia**

  Una clase es una versión especializada de otra existente

Ejemplo:



Cuenta es cliente de **Persona**

CuentaAhorro es una especialización de **Cuenta**

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

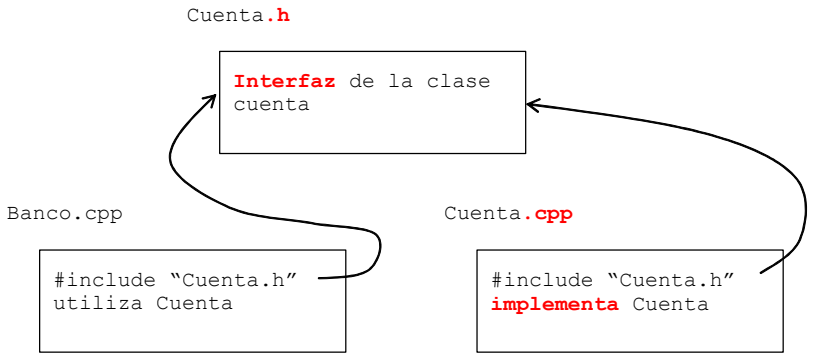
Clases en C++

- **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 Java

// **Cuenta.java**, declaración de la clase Cuenta (→ **.class** por cada clase al compilar)

```
package gestiondecuentas;
import java.util.*;
public class Cuenta {

   private Persona  titular;
   private int      saldo;
   private int      codigo;
   private static int  ultimoCodigo;      //variable de clase
   private int [] ultOper;

   public Cuenta (Persona quien)  {saldo=0;
                                   titular=quien;
                                   codigo = nuevoCodigo();
                                   ultOper = new int[20];}
   public  void     reintegro(int suma)
   { ... }
   public  void     ingreso(int suma)
   { ... }
   public  int      verSaldo() { ... }
   public  void     verUltOper(int n) { ... }
   public static int  nuevoCodigo() {return ultimoCodigo++}

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

Clases en Java

- **Abstracción de tipos**
  - no existen los punteros
  - tipos primitivos y referencias
  - Variables y métodos de clase (`static`)
- **Ocultación de Información**
  - Se especifica para cada característica
  - **`public, private`** (El mismo significado que en C++)
  - ***visibilidad a nivel de paquete***: accesible desde todas las clases del paquete, inaccesible para los clientes del paquete.

Clases en Java

- **Modularidad**
  - Definición de clases (**`class`**) e interfaces (**`interface`**)
  - Una interfaz sólo define el comportamiento abstracto del tipo, no contiene implementación.
  - Categorías de módulos relacionados: **paquetes (`package`)**.
  - Si un cliente quiere utilizar la clase `Cuenta` dentro del paquete `gestionCuentas` puede:
    - Importar el paquete: **`import`** `gestionCuentas.*;`
    - Notación punto: `gestionCuentas.Cuenta`
      (no existe el operador de alcance :: de C++)

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++
  - Clases anidadas — Java y C++
  - Exportación selectiva — Eiffel
- Tanto los paquetes como las clases anidadas añaden complejidad al lenguaje.
- Con las soluciones de Java y C++ se corre el riesgo de perjudicar la reutilización.

3.- Objetos

Definición

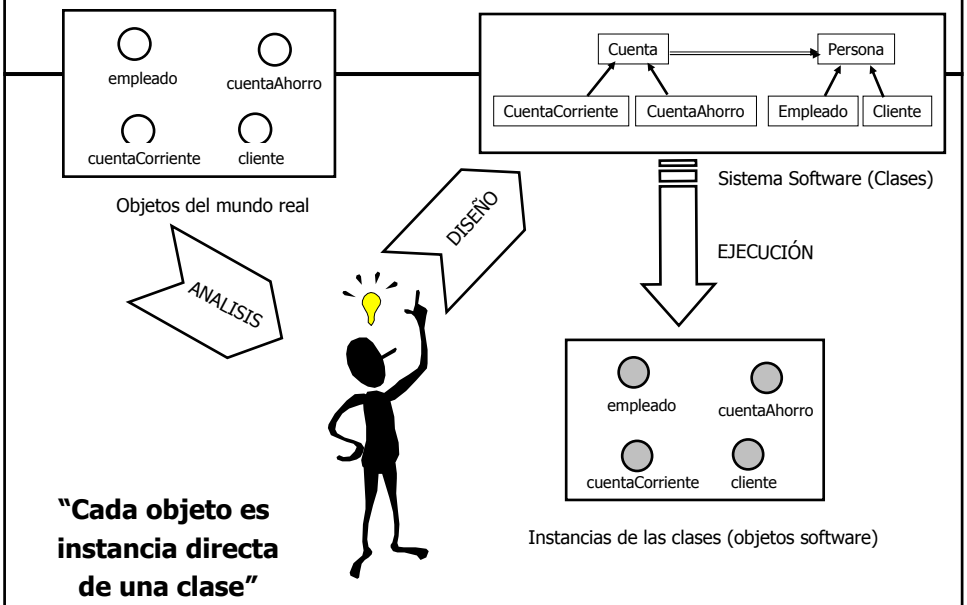
"Es una instancia de una clase, creada en tiempo de ejecución"

- Es una estructura de datos formada por tantos **campos** como **atributos** tiene la clase.
- El **estado** de un objeto viene dado por el valor de los campos.
- Las **rutinas** permiten consultar y modificar el estado del objeto.
- Durante la ejecución de un programa OO se crearán un conjunto de objetos.

Objetos dominio vs. Objetos aplicación

Ejemplo: Aplicación Correo electrónico

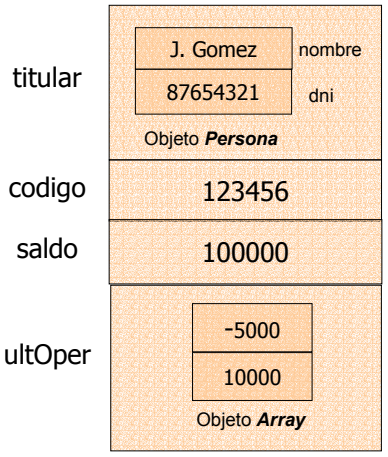
- **Objetos externos:**
  - Procedentes del dominio de la aplicación
    "carpeta", "buzón", "mensaje"
- **Objetos software:**
  - Procedentes del ANALISIS: todos los externos
  - Procedentes del DISEÑO/IMPLEMENTACION:
    "árbol binario", "cola", "lista enlazada", "ventana",...



Tipos de campos

- **Simples**
  - Corresponden a atributos de tipos (clases) básicos
  - En Eiffel:
    INTEGER, REAL, DOUBLE, BOOLEAN,
    CHARACTER, STRING
- **Compuestos**
  - Sus valores son objetos de tipos no básicos.
  - **Subobjetos** vs. **Referencias**.

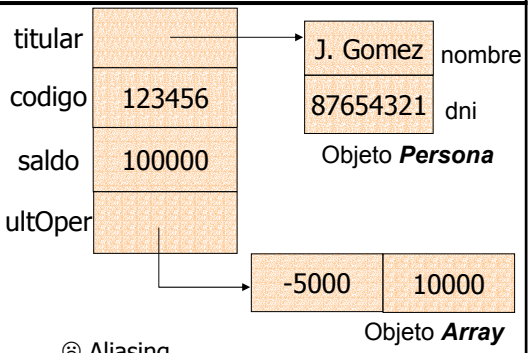
Subobjetos



☹ Desperdicia memoria

☹ No permite compartir

Referencias



☹ Aliasing

☺ Compartición (integridad referencial)

☺ Estructuras recursivas (Autoreferencias)

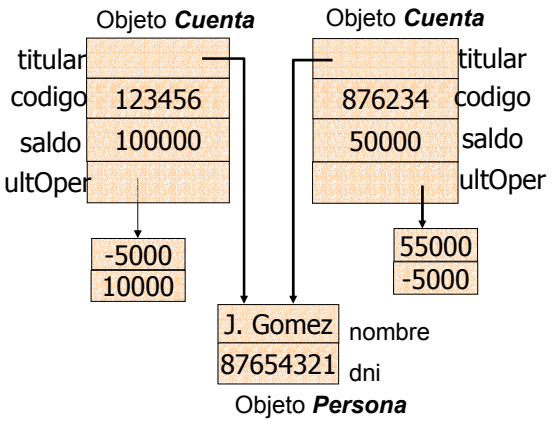
☺ Más eficiente manejo obj. complejos

☺ Obj. se crean cuando se necesitan

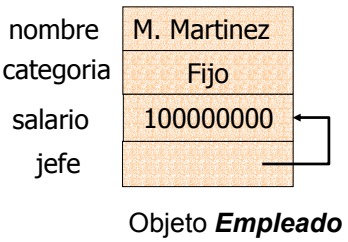
☺ Soporte para el polimorfismo

Ejemplos referencias:

a) Compartición



b) Autorreferencias



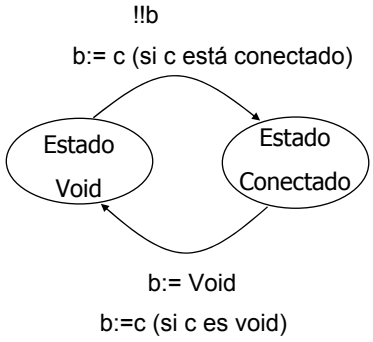
Referencias e identidad de objetos

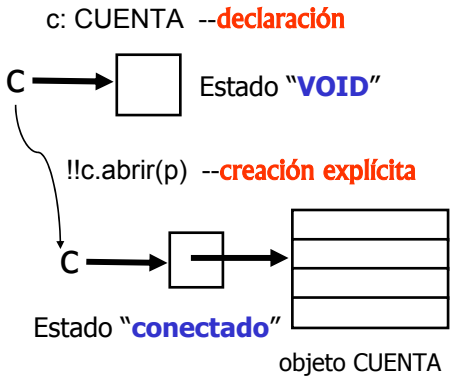
Definición: referencia

Una referencia es un valor en tiempo de ejecución que está o **vacío** (`void`/`null`) o **conectado**. Si está conectado, una referencia identifica a un único objeto *(nombre abstracto* para el objeto).

- Puede implementarse de distintas formas.
- Mientras exista, cada objeto posee una identidad única, independiente de sus valores (**identificador de objeto**, oid):
  - *Dos objetos con diferentes oids pueden tener los mismos valores en sus campos.*
  - *Los valores de los campos de un objeto pueden cambiar, pero su oid es inmutable.*

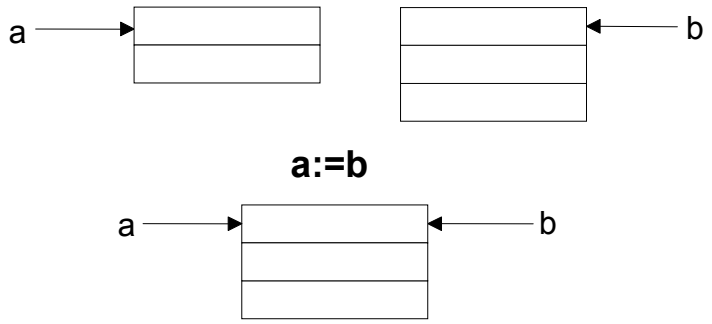
Estados de una referencia





En Eiffel, los posibles valores de una entidad (atributo, parámetro, …) son referencias a objetos potenciales que pueden ser creados en tiempo de ejecución a través, siempre, de instrucciones de creación explícitas.

☹ ¡Cuidado con el Aliasing!



La asignación no implica copia de valores sino de referencias

Consecuencia del aliasing

```
-- P(b) es cierto
        a:=b
        rutina(a)        -- rutina no afecta a b
-- P(b) puede ser falso
```

Ejemplo 1:

```
class C  feature
    atrib: BOOLEAN;
    ...
    set_atrib_false is do
        atrib:= false
    end;
end
```

```
x,y:C
-- y /= void and y.atrib=true
        x:=y;
        x.set_atrib_false;
-- y.atrib=false
```

Consecuencia del aliasing

Ejemplo 2:

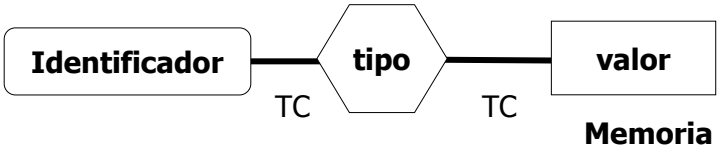
```
x,y: INTEGER;

-- y>0
        x:= y;
        x:= -1
-- y>0
```

- Con **semántica de almacenamiento** la propiedad se sigue cumpliendo
- Una asignación a **x** no puede afectar a **y**

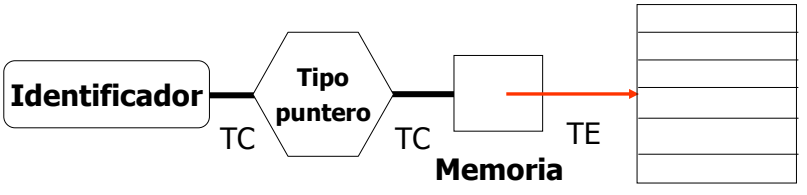
"Aliasing" es peligroso pero inevitable.

4.- Semántica almacenamiento vs. semántica referencia

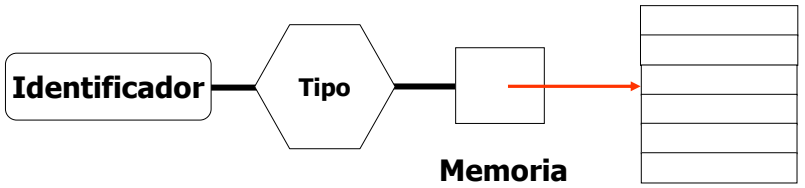
- Variables en los lenguajes tradicionales



- Variables tipo "puntero" en lenguaje tradicionales



Semántica referencia



"referencias" y "punteros" son conceptos muy **PRÓXIMOS PERO:**

- "**referencias**" se asocian a **objetos.** Toda referencia tiene un tipo.
  (**`void`** / **`null`** = **estado** no ligado)
- "**punteros**" se asocian a **direcciones** de memoria.
  (**`nil/null`** (Pascal/C) = **valores** de tipo puntero)

"Una variable denota una referencia a un objeto"

Tipos expandidos en Eiffel

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

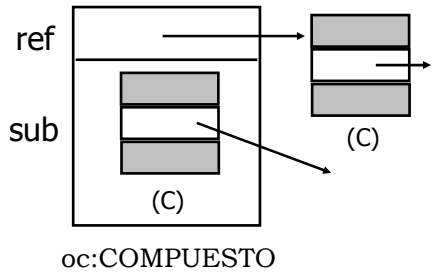
```
expanded class PERSONA
…
end;

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).

5.- Métodos y mensajes

Está compuesta por:

- **CABECERA**: Identificador y Parámetros
- **CUERPO**: Secuencia de instrucciones

Ejemplo (Eiffel):

```
reintegro (suma: REAL) is do
        if puedo_sacar(suma) then saldo:= saldo - suma
end
```


Definición de Métodos

- ¿Qué **instrucciones** podemos incluir en el cuerpo de un método?
  - Asignación
  - Condicional
  - Iteración
  - Invocación a otro método = **Mensajes**
  - Creación de objetos (apartado 7)
- Un método se ejecutará como respuesta a un mensaje.

Instrucciones Eiffel:

i) **Asignación**

```
ox := oy
```

ox: una entidad (atributo, variable local, Result)
oy: una expresión (constante, mensaje, entidad, Current) de tipo compatible

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
```


Rutinas. 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;
```

Mensajes

- Mecanismo básico de la computación OO.
- Invocación de la aplicación de un método sobre un objeto.
- La modificación o consulta del estado de un objeto se realiza mediante mensajes.
- Formado por tres partes
  - Objeto **receptor**
  - **Selector** o identificador del método a aplicar
  - **Argumentos**

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

Semántica mensajes

- Sea el mensaje **`x.f`**, su significado es:

> "Aplicar el método *f* sobre el receptor *x*, efectuando el paso de parámetros"

👁 ¡NO CONFUNDIR CON LA INVOCACIÓN DE UN PROCEDIMIENTO!

Mensajes vs. Procedimientos

- Un mensaje parece una llamada a procedimiento en la que sólo cambia el formato:

```
unaCuenta.ingreso (100000)
ingreso (unaCuenta,100000)
```

- En una invocación a procedimiento todos los argumentos se tratan del mismo modo.
- En un mensaje un argumento tiene una naturaleza especial: **"objeto receptor"**

Argumentos de las rutinas

- Los argumentos son entradas a las rutinas y no deberían cambiarse
- Paso de parámetros:
  - paso de valores simples (por valor)
  - referencias a entidades (por referencia)
    - Permite cambiar el valor de una entidad externa a la rutina (debe hacerse a través de la interfaz del objeto referenciado)
    - Aliasing (cambias el estado de los objetos)
- En programación OO se utilizan los argumentos por referencia para pasar el objeto original (no la copia)

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
- **C#**
  - Cuatro tipos diferentes de parámetros (entrada, salida, por referencia y de número indefinido)

Paso de parámetros en Eiffel (Cap13. Meyer)

Sea la rutina

$r\ (p_1: T_1, \ldots, p_n: T_n)$

Argumento formal

la invocación

$r\ (a_1, \ldots, a_n)$

Argumento real

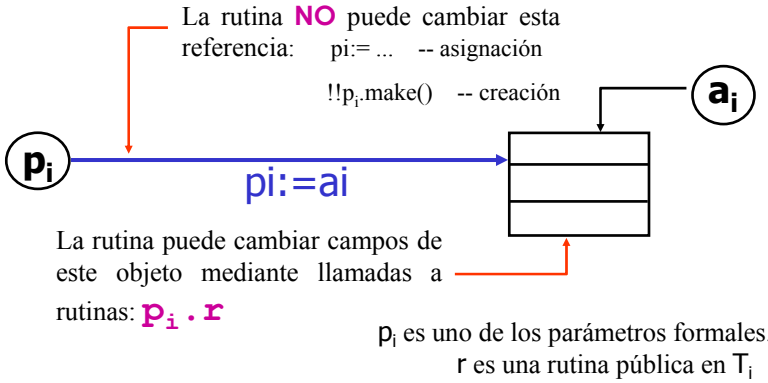
Las preguntas relevantes son:

- ¿Cuál es la correspondencia entre parámetros reales y formales?

$$\left.\begin{array}{l} p_1 := a_1; \\ \ldots \\ p_n := a_n; \end{array}\right\} \text{ ASIGNACIÓN}$$

- ¿Qué operaciones se permiten sobre los parámetros formales?
- ¿Qué efecto tendrán éstas sobre los parámetros reales correspondientes?

Operaciones permitidas con argumentos de tipo referencia



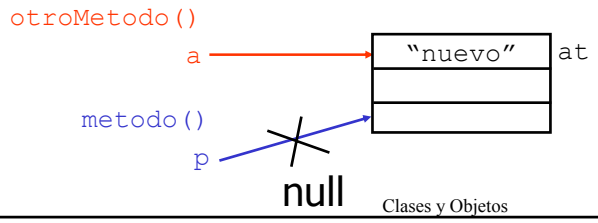
Al excluir las rutinas que modifican los argumentos se puede utilizar **cualquier expresión** como argumento real.

Paso de parámetros en Java

- Es posible modificar el parámetro (si no es `final`)
- El argumento no se cambia porque el parámetro era una copia que se pasa por valor.

```
public void metodo (T p){
      p.setAt("nuevo");
      p=null;
}
```

```
public void otroMetodo(){
      T a=new T();
      obj.metodo(a);
}
```



Instancia actual:

"Cada operación de una computación OO es relativa a cierto objeto, la **instancia actual**, en el momento de la ejecución de la operación".

¿A qué objeto **Cuenta** se refiere el texto de la rutina **reintegro**?

El cuerpo de una rutina se refiere a la instancia actual sobre la que se aplica

Ejemplo:

```
oc1, oc2: CUENTA;
os: INTEGER;
!!oc1; !!oc2;
        …
oc1. reintegro(1000);
os := oc1.saldo;
        …
oc2. reintegro(2000)
```

inst.actual=receptor llamada actual

Mensajes y "objeto actual"

```
class ICE1  feature          -- clase Eiffel
   at1: INTEGER;
   at2: ICE2;
   at3: ICE3;
   rut1(p: INTEGER) is do
       at1:= p;
       at2.una_rutina;
       rut2;
   end;
   rut2 is do
       at3. otra_rutina(Current)
   end;
end
```

Llamada calificada
Objeto Receptor Explícito

Llamada no calificada
No se especifica el obj. receptor

-- ¿Quién es el objeto receptor?

<=> Current.rut2

???

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`

  } opcional
- Pasar referencia al objeto actual como parámetro a otro método:

```
servicio.añadir(this);
```


Combinación módulo-tipo

- Como cada módulo es un tipo, cada operación del módulo es relativa a cierta instancia del tipo (instancia actual)

Cómo funciona la fusión módulo-tipo

"Los servicios proporcionados por una clase, vista como un módulo, son precisamente las operaciones disponibles sobre las instancias de la clase, vista como un tipo".

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)

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;
```

3.5.- 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

Sea la declaración Eiffel:

`oc:Cuenta` (tipo referencia)

¿por qué no se le asocia el objeto en tiempo de compilación en lugar de inicializarlo como `void`?

¿por qué es necesaria una creación explícita y no es suficiente con la declaración?

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.

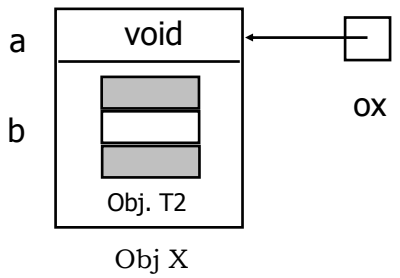
Tipos expandidos y Creación

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?



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"

(C) Creación en Java

Igual que en C++:

- Adopta los **constructores** para garantizar la inicialización de los objetos (permite sobrecarga)
- El compilador proporciona un constructor por defecto siempre y cuando no hayas definido ninguno

A diferencia de C++:

- Garantiza que cada atributo de una clase tenga un **valor inicial** antes de la llamada al constructor

```
class Contador {
    int valor;
    Contador () {i=7;}  // i = 0 -> i=7
    ...}
```

- Puede inicializar en el cuerpo de la clase

```
class Contador{
  int valor=7; …}
```

79

(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;
    }
}
```


//:InitialValues.java

```
//Muestra los valore iniciales por defecto
class Measurement{
        boolean t;
        char c;
        byte b;
        short s;
        int i;
        long l;
        float f;
        double d;
        void print(){
                System.out.println (
                        "Data type      Initial value\n" +
                        "boolean        " +t + "\n" +
                        "char           " +c +"\n" +
                        "byte           " +b +"\n" +
                        "short          " +s +"\n" +
                        "int            " +i  +"\n" +
                        "long           " +l  +"\n" +
                        "float          " +f  +"\n" +
                        "double         "+d);
                }
        }
```

//:InitialValues.java (continuación)

```
public class InitialValues{
        public static void main (String[] args) {
                Measurement d = new Measurement();
                d.print();
        }
} //FIN
```

La salida del programa sería:

Data type	Initial value	
boolean	false	
char		(null no se imprime)
byte	0	
short	0	
int	0	
long	0	
float	0.0	
double	0.0	

NOTA: las referencias se inicializan a null

Recolección de basura en Java

- **NO** hay destructor en Java
- Recolección automática de **memoria**
- Existe un método **`finalize()`** para casos "especiales" en los que se asigna memoria por un procedimiento distinto al normal (`new`).
- Este método se invocará justo antes de la recolección de basura
- En C++ todos los objetos se destruyen (en un programa sin errores), mientras que en Java no siempre se "recolectan".

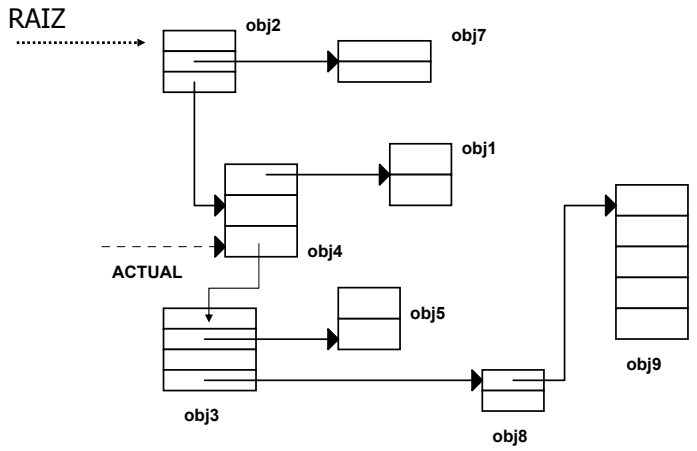
8.- Modelo de ejecución OO

- Para obtener un código ejecutable se deben ensamblar las clases para formar **sistemas** (cerrado).
- Un sistema viene dado por:
  - Un conjunto de clases
  - La **clase raíz**
  - El procedimiento de creación de la clase raíz.
- La **ejecución** de un programa **OO** consiste en:
  - Creación dinámica de objetos
  - Envío de mensajes entre los objetos creados, siguiendo un patrón impredecible en tiempo de compilación
- **Ausencia de programa principal**

Modelo de ejecución OO

- ¿**Cómo empieza** la ejecución de un programa OO?
  - –Creación de un "objeto raíz"
  - –Aplicar mensaje sobre "objeto raíz"
- **En tiempo de ejecución**, el flujo de ejecución siempre se encuentra **aplicando una operación sobre un objeto (instancia actual)** o ejecutando una operación que no es un mensaje (asignación, creación).
- En un instante dado bien se aplica un mensaje sobre la instancia actual o sobre un objeto accesible desde él.
- **¿Cómo se ejecuta un mensaje?**
  - Ej: `c.reintegro(cantidad)`
  - Formará parte del cuerpo de una rutina de una clase

Estructura de objetos en tiempo de ejecución



El método `main`

- Debemos proporcionar el nombre de la clase que conduzca la aplicación
- Cuando ejecutamos un programa, el sistema localizará esta clase y ejecutará el `main` que contenga
- El método `main` debe ser:

```
public class Eco{
  public static void main(String[] args){
       for(int i=0;i<args.length;i++)
             System.out.println(args[i]+" ");
  }
}
```

- Los parámetros en el array de cadenas de texto son los *parámetros del programa*

```
java Eco estamos aquí --> SALIDA: estamos aqui
```

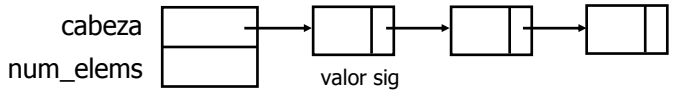
Ejercicio: Traducir a Eiffel el siguiente código Pascal

```
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

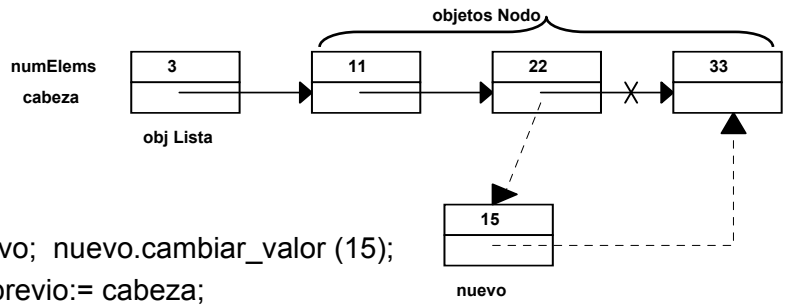



```
class LISTA_ENTEROS feature          -- clase Eiffel
    cabeza: NODO_ENTERO;
    num_elems: INTEGER;

    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"

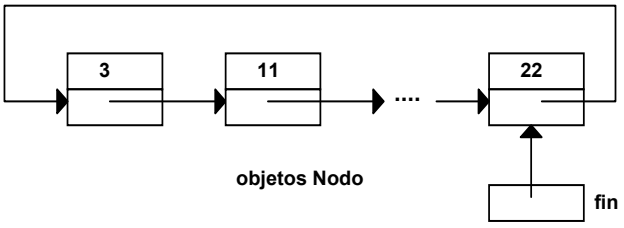
```
insertar (i: INTEGER, v: INTEGER) is
        local       nuevo, previo: NODO_ENTERO;
                    j: INTEGER;
        do
                !!nuevo;
                nuevo.cambiar_valor(v);
                if i = 1 then
                        nuevo.cambiar_sig (cabeza);
                        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)

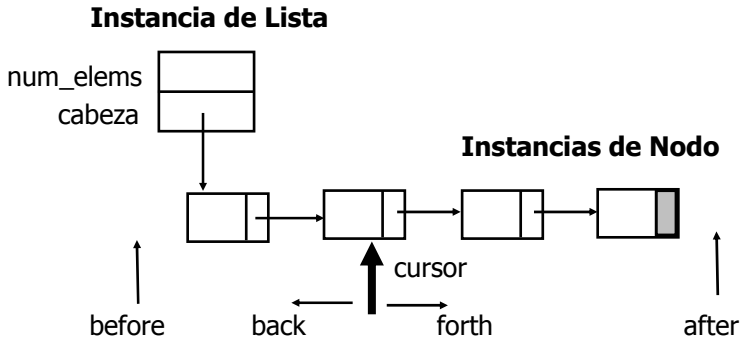
La clase Lista está mal diseñada -> Clase pasiva

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`

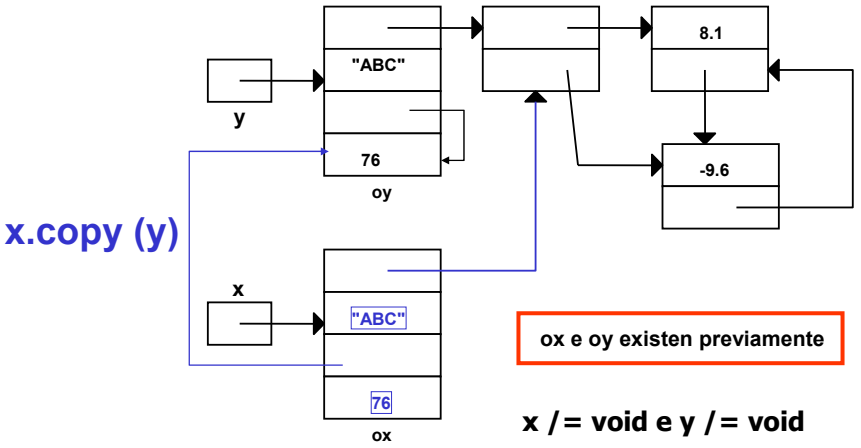
- Es necesario establecer de **manera precisa** lo que sucede con el cursor después de cada operación.

Ejemplo de uso de las Listas activas

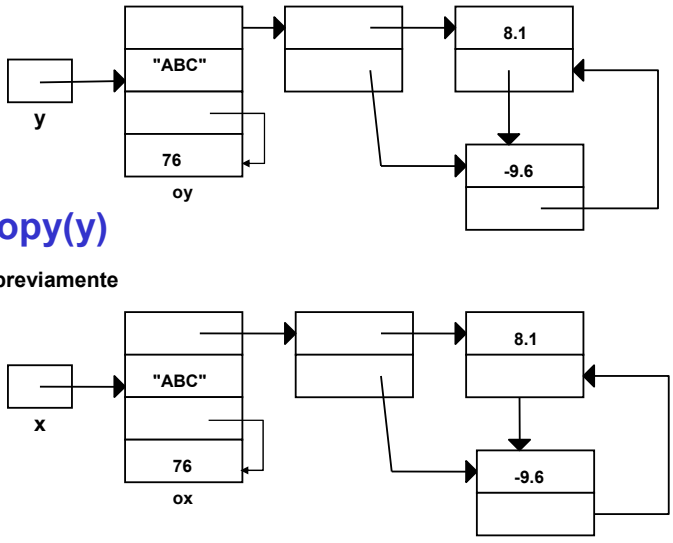
```
l: Lista_Enteros;
m, n: INTEGER;
...
l.start; l.search(m);
if not after then l.put_right(n) end;
...
l.start;
l.search(m); l.search(m); l.search(m);
if not after then l.remove end;
...
l.go(i);l.put_left(m);
```

9.- Operaciones sobre referencias:

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



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



(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:

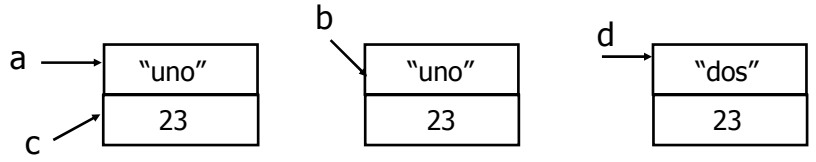
```
!!x
x.copy(y)
```

¿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

(C.2) Igualdad profunda

- **<u>Dos referencias</u>** x e y son iguales en profundidad si:

  1) x=y=void

  2) Están conectados a "objetos iguales en profundidad"

- **Dos objetos,** *ox* y *oy*, **son iguales en profundidad**, si satisfacen las siguientes cuatro condiciones:

  1) Tienen el mismo tipo

  2) Los objetos obtenidos al hacer *void* todos los campos-referencia de *ox* e oy *son* iguales.

  3) Para cada campo-referencia de *ox* con valor *void*, el correspondiente campo de *oy* es void

  4) Para cada campo-referencia de *ox* conectado a un objeto *px*, el correspondiente campo de *oy* está conectado a un objeto *py*, y es posible demostrar recursivamente que *px* y *py* son iguales en profundidad, asumiendo que *ox* e *oy* lo son.

Igualdad de objetos

x:=clone(y) $\Rightarrow$ equal (x,y)

x.copy(y) $\Rightarrow$ equal (x,y)

x:= deep_clone(y) $\Rightarrow$ 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)

Copia y clonación de objetos en Java

- Puede ser útil para hacer una copia local de un objeto
- Constructor de copia:
  - Construye un nuevo objeto como una copia del que se le pasa

```
Cuenta(Cuenta otra){
        codigo = otra.getCodigo();
        saldo = otra.getSaldo();
        titular = otra.getTitular();
}
```

  - No se usa mucho dentro de las bibliotecas de clases de Java.
  - Existe en la clase String y las colecciones.
  - La forma preferida de obtener la copia de un objeto es utilizar el método **`clone`**

Clonación de objetos: `Object.clone`

- Devuelve un nuevo objeto cuyo estado inicial es una copia del estado actual del objeto sobre el que se invoca a `clone`
- Factores a tener en cuenta:
  - La clase que proporciona el método **`clone`** debe implementar el interfaz **`Cloneable`**
  - Definir el método `clone` como `public` (en la clase `Object` es `protected`, por lo que no se puede hacer el clone de un `Object`)
  - Puede ser necesario cambiar la implementación por defecto del método para hacer un clone en profundidad
  - Se puede utilizar la excepción **`CloneNotSupportedException`** para inidicar que no se debería haber llamado al método clone

Igualdad en Java

- Igualdad de referencias *(identidad)*:

```
objPila1 == objPila2 --> false
objPila1 != objPila2 --> true
```

- Método `equals`
  - Disponible para todo objeto
  - **`public boolean equals(Object obj)`**
  - Comportamiento por defecto: `this==obj`
  - Utilizado para implementar la igualdad de objetos.

10.- Genericidad

- ¿Cómo escribir una clase que represente una estructura de datos y que sea posible almacenar objetos de cualquier tipo?

~~**Pila-Enteros**~~
~~**Pila_Libros**~~
~~**Pila_Figuras**~~
~~…~~

$\Rightarrow$ **Pila de?**

- Necesidad de reconciliar reutilización con el uso de un lenguaje tipado. → Legibilidad → Fiabilidad

Genericidad

- Posibilidad de parametrizar las clases; los parámetros son tipos de datos.

- Facilidad útil para describir estructuras contenedoras generales que se implementan de la misma manera independientemente de los datos que contiene: TIPO BASE ES UN PARÁMETRO.

  class ARRAY [T], class PILA [T], class LISTA [T], ...

Declaración de una clase genérica

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

Uso de una clase genérica

- El **parámetro genérico actual** puede ser:

1) Un **tipo expandido**

```
pe: PILA [INTEGER]
```

2) Un **tipo referencia**

```
pp: PILA  [PUNTO] ;      ppp: PILA [PILA [PUNTO]] ;
aac: ARRAY [ARRAY [CUENTA]]
```

3) Un **parámetro genérico formal de la clase cliente**

```
class C[G]  feature
        at: PILA [G];
        ....
end
```


Genericidad y Control de tipos

- Mediante el uso de la genericidad el compilador garantiza que una estructura de datos contenga sólo elementos de un único tipo.

```
pp: PILA [PUNTO];
pc: PILA [CUENTA];
p: PUNTO;
c: CUENTA;
```

MENSAJES VALIDOS	**MENSAJES NO VALIDOS**
pp.put (p)	pp.put (c)
pc.put (c)	pc.put (p)
p:= pp.item	p:= pc.item

- **La genericidad SÓLO tiene sentido en un LENGUAJE TIPADO**

Operaciones sobre entidades de tipos genéricos

Sea la clase:

```
class C [G,H,..] feature
        x:G
        rut (p:H) is do .. end;
                ...
end
```

¿Qué operaciones podemos aplicar sobre las entidades cuyo tipo es un parámetro genérico?

En una clase cliente, **G, H,…** pueden ser instanciados a cualquier tipo

Operaciones sobre entidades de tipos genéricos

- Cualquier operación sobre el atributo *x* debe ser aplicable a cualquier tipo.
- Cinco posibles operaciones:

  1) **x:= y** (y es una expresión de tipo G)
  2) **y:= x** (y es una entidad de tipo G)
  3) **x=y ó x/=y** (y es de tipo G)
  4) **a.f(...,x,...)** (x actúa como argumento en un mensaje, el correspondiente parámetro es de tipo G o ANY.
  5) Receptor de un mensaje que invoca a una rutina de ANY (ej. copy, clone, equal)

- **No se permite !!x.**

Ejemplo: Arrays en Eiffel

```
class ARRAY[G] creation make
  feature
        make(minindex,maxindex: INTEGER) is do .. end;
            -- Asignar espacio a un array de límites minindex, maxindex
        lower, upper, count: INTEGER;
                -- Indices mínimo y máximo permitido y tamaño del array
        put (v:G; i: INTEGER) is do .. end;
                -- Asignar v a la entrada de índice i
        infix "@", item (i:INTEGER): G is do .. end;
                -- Elemento cuyo índice es i
end.
```


Cuestiones sobre genericidad

- "Sin genericidad es imposible lograr una comprobación estática de tipos en un lenguaje OO realista"

  [B. Meyer]
- ¿Cómo definimos sin genericidad las estructuras de datos sin repetir código?
- ¿Cómo podemos definir una estructura de datos que almacene objetos que sean tipos de figuras?
- ¿Es posible exigir que los parámetros genéricos actuales sean tipos que incluyan ciertas operaciones?

Genericidad en C++

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

```
template <class T> class Pila {
  private:
    T* top;
    int count;
  public:
    Pila (int t);
    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 Java (hasta JDK 1.4)

- No tiene templates ni otra forma de implementar tipos parametrizados.
- Proporciona un conjunto de colecciones (ej. `ArrayList`) que contienen referencias a `Object`.
- Problemas => Se pierde la información del tipo:
  - La colección puede contener cualquier tipo.
  - Hay que efectuar un **cast** antes de utilizar el objeto que se recupera de la colección.
  - Se detecta un objeto del tipo no deseado en tiempo de ejecución.

Genericidad en Java

```
public class Pila {
  private ArrayList contenido; //contenedor de Object
  public void push (Object obj){...}
  public Object pop () {...}
}
```

```
Pila p; //quiero que sea de Movimientos de Cuenta
Movimiento m; Animal a;
p.push (m);
p.push(a);                  //NO daría error el compilador
m=p.pop();                  //error asignamos un Object
m=(Movimiento)p.pop()       //OK
```

Perdemos la ventaja de la comprobación estática de tipos, las comprobaciones las hace el programador

Genericidad en Java JDK 1.5

- Introduce la genericidad en el lenguaje.
- Se comprueba la corrección de tipos en tiempo de compilación.
- No hay que hacer conversiones explícitas al extraer los elementos del contenedor.
- No produce múltiples copias del código. Una declaración de un tipo genérico se compila sólo una vez y produce un único fichero .class.

Genericidad en JDK 1.5

```
public class Pila<E> {
  private ArrayList<E> contenido;

 public Pila(){
   contenido = new ArrayList<E>();
 }
  public void push (E x){ …}
  public E pop(){ … }
}

Pila<Libro> miPila = new Pila<Libro>();
miPila.push(new Libro(…));
Libro libroReciente = miPila.pop();
```


Genericidad en Java 1.5

- No se puede instanciar el parámetro con un tipo primitivo
- No hay genericidad en la máquina virtual, sólo clases y métodos ordinarios.
- Todos los parámetros son sustituidos (internamente) por un tipo
  - Por ejemplo, `en Pila<T>` se sustituye `T` por `Object`
- El compilador inserta automáticamente el *cast* cuando lo considera necesario para preservar la seguridad de los tipos.

Genericidad en C#

- Igual que en Java, no formaba parte de la primera versión.
- Se va a incorporar como elemento del lenguaje permitiendo definir una clase como:

```
class List<T>{ … }
```

- El parámetro genérico actual podrá ser cualquier tipo definido: `List<int>`, `List<Cuenta>`
- No hay que hacer conversión de tipos y nos beneficiamos de la comprobación estática de tipos.

Ejemplo: "Correo electrónico"

```
class BUZON_CORREO feature
    conj_carpetas: LIST [CARPETA];
    buscar (s: STRING): LIST [MENSAJE] is do
        "Para cada carpeta c en conj_carpetas"
        c.buscar(s)
    end; ...
end

class CARPETA feature
    conj_mensajes: LIST [MENSAJE];
    buscar (s: STRING): LIST [MENSAJE] is do
        "Para cada mensaje m en conj_mensajes"
        m.buscar(s)
    end; ...
end

class MENSAJE feature
    contenido: STRING
    emisor: PERSONA;
    fecha: FECHA;
    buscar (s: STRING): BOOLEAN is do
        Result:= (contenido = s)
    end; ...
end
```

Secuencia de mensajes:

 .buscar("tema")

Para cada 

 .buscar("tema")

Para cada

.buscar("tema")