Departamento de Informática | Universidad de Valladolid

# Programación III.I.T.I. de Sistemas

# Introducción

Félix Prieto

Curso 2007/08



## Contenidos

- Contenidosss
  - Clases
  - Objetos
  - Clases genéricas
  - El estilo de programación OO
- Repaso rápido de conceptos estudiados en Programación II
- Detalles técnicos referidos a Eiffel
- Consultar documentación adicional y/o bibliografía si es necesario





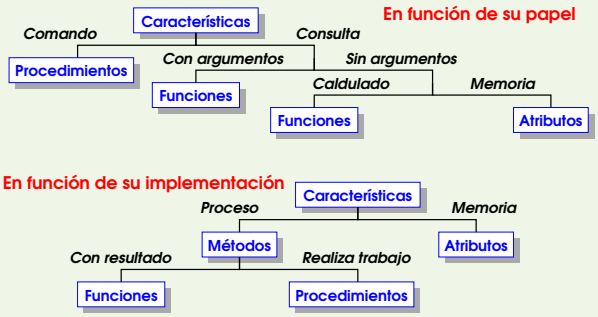
## Definición de clase

- Una *clase* es un TAD dotado de una implementación (posiblemente parcial)
- Las clases son *diferidas* si están parcialmente implementadas y *efectivas* en otro caso
- Sobre las clases diferidas volveremos más adelante
- Las clases están formadas por *características*
- La instancia de una clase es un objeto del sistema
- En Eiffel todo objeto del sistema es instancia de una clase, de modo que cada clase nos proporciona «un» tipo. Podemos utilizar «tipos» como `INTEGER`, `REAL`, `DOUBLE`, `BOOLEAN`, `CHARACTER`, pero *todos ellos son clases en Eiffel.*





## Clasificando las características








## Otros conceptos básicos

- Llamamos *entidades* a los *atributos*, las *entidades locales* y los *argumentos formales* de los métodos
- Todas las entidades tienen un *tipo*, asociado a una clase
- Una clase `A` es cliente de `B` si dispone de alguna entidad que es instancia de `B`
- Podemos enviar mensajes a las instancias de las clases de que somos clientes, o a nosotros mismos (mensajes no cualificados o enviados a `Current`)
- No hay que utilizar `Current` cuando no es necesario





## Definición de objeto

- Llamaremos *objeto* a la instancia de una clase en tiempo de ejecución
- Una clase se puede instanciar en varios objetos distintos
- Una *referencia* es un valor en tiempo de ejecución que puede estar *vacío* (`Void`) o *conectado*. Cuando una referencia está conectada apunta un único objeto, y decimos que está conectada al objeto
- Cada objeto tiene su propia identidad, pero sólo podemos acceder a los objetos mediante referencias
- En Eiffel no existe la aritmética de punteros





## Dos formas de ser cliente






## Clientes expandidos y no expandidos

- Una clase es *expandida* si su definición comienza por `expanded class`
- Una relación de cliente es expandida si la clase proveedor es expandida
- Una relación de cliente es expandida si en la definición de la entidad el nombre de la clase va precedido de `expanded`
- Los clientes expandidos no pueden compartir a sus proveedores
- Los clientes expandidos no pueden ver modificados a sus proveedores por otros
- `INTEGER` o `CHARACTER` son clases expandidas

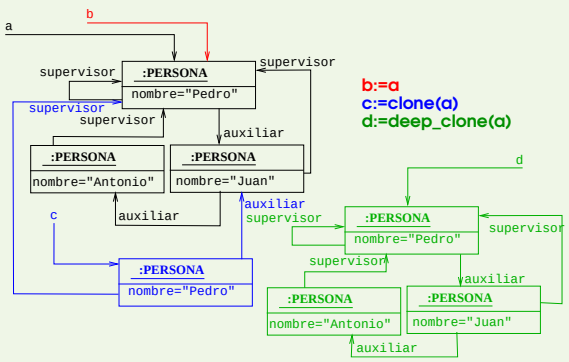
# Creación de los objetos

- La creación de objetos expandidos es *automática* mientras la de los no expandidos en *manual* utilizando `create` o `!!`
- El código de la clase puede imponer la utilización de un *método de creación* para inicializar sus atributos
- Las clases expandidas no tienen método de creación, o tienen *sólo uno* sin parámetros
- La destrucción de los objetos es automática cuando dejan de tener clientes
- La gestión de memoria es automática y utiliza un recolector de basura
- Los atributos y entidades locales se inicializan a valores por defecto. No necesariamente hay que aplicar `create` para utilizarlos





# Operaciones sobre referencias y objetos








# Operaciones sobre referencias y objetos (II)

Fuente $y$

Objetivo $x$

| | *Referencia* | *Expandido* |
|---|---|---|
| ***Referencia*** | Conexión por referencia | Como `x:=clone(y)` |
| ***Expandido*** | Como `x.copy(y)` ( falla si `y=Void` ) | Como `x.copy(y)` |

Tipo de $y$

Tipo de $x$

| | *Referencia* | *Expandido* |
|---|---|---|
| ***Referencia*** | Compara referencias | Como `equal` `False` si `x=Void` |
| ***Expandido*** | Como `equal` `False` si `y=Void` | Como `equal` |





# Clases genéricas

```
class PILA[G]
feature
  contador: INTEGER
    -- Número de elementos
  vacia: BOOLEAN is
    -- ¿Está vacía?
  do ... end
  llena: BOOLEAN is
    -- ¿Está llena?
  do ... end
  cima: G is
    -- Elemento de la cima de la pila.
  do ... end
  mete( x:G) is
    -- Introduce x por la cima
  do ... end
  saca is
    -- Quita el elemento de la cima
  do ... end
end -- Clase PILA
```

- Utilizamos un *parámetro genérico*, `G` en este caso
- Para instanciar la clase debemos proporcionar como *parámetro actual* un tipo, por ejemplo `a:PILA[INTEGER]`
- De una clase genérica se derivan tantos tipos como parámetros actuales podamos proporcionar
- El parámetro puede aparecer en los métodos, definiendo el tipo de las entidades utilizadas
- Debemos reconsiderar las reglas que controlan la sustitución de *argumentos formales* por *argumentos actuales* en el paso de mensajes





# Genericidad y mensajes

- Sea `C` una clase genérica con parámetro formal `G` y `h(a:G):G is ...` la definición de un método que aparece en ella. La llamada `y.h(e)` que aparece en un método de la clase `B` es correcta si existe un tipo `V`, parámetro actual de `C` tal que :
  - `y` es de tipo `C[V]` (o compatible por herencia).
  - `h` está exportada a `B` (explícita o implícitamente).
  - `e` es de tipo `V` (o compatible por herencia).
- En este caso el resultado de `h` es de tipo `V`
- Al compilar la clase genérica no conocemos el tipo que se utilizará como parámetro formal, luego...
- En el código de la clase genérica, las entidades de tipo `G` sólo pueden utilizarse como si fuesen de tipo `ANY`





# El estilo Orientado a Objetos

- La programación Orientada a Objetos hace uso intensivo de la modularidad
- Debemos prestar especial atención a...
  - Ocultación de información
  - Diferenciación entre consultas y comandos
  - Elección de los argumentos
  - Elección de los identificadores
  - Normas de documentación
- Que «funcione» es necesario, pero no suficiente
- Que funcione «deprisa» es deseable, pero en general es preferible que su mantenimiento sea rápido





# Ocultación de información

- Al diseñar cada clase debemos decidir qué características serán visibles para los clientes (*interfaz*) y cuales no
- Ocultar información sobre nuestra clase protege a los programadores de otras clases de las consecuencias de nuestras tareas de mantenimiento
- En la forma corta de una clase no podemos distinguir las funciones sin argumentos de los atributos
- No podemos modificar los atributos de un objeto distinto de `Current`
- Ante la duda, las características deben ser privadas (arquitectura «iceberg»)





# Consultas frente a comandos

- Las *consultas* devuelven un resultado pero no cambian el estado del receptor
  - Sin embargo pueden cambiar sus atributos privados
  - Estos cambios no deben afectar al comportamiento del receptor
- Los *comandos* cambian el estado del receptor
- No debemos mezclar comandos y consultas en la misma orden
- Este criterio explica la forma de leer enteros implementada en la clase `STD_INPUT`
  - `read_integer` es un comando
  - `last_integer` es una consulta

## Elección de los argumentos

- Hay que mantener pequeño el número de argumentos que acompañan a los mensajes (mantener bajo el *acoplamiento*)
- Cuando sean muchos...
  - ¿Seguro que los argumentos no pueden encapsularse en un objeto no artificial?
  - ¿Seguro que no es mejor cambiar antes el estado del receptor?
  - ¿Seguro que el receptor no puede obtener la información por si mismo?





## Elección de los identificadores

- El código de una clase debe ser comprensible en si mismo
- Como norma general los identificadores deben ser claros, pero no tediosos
- Los nombres de las características públicas deben ser suficientemente explícitos
  - No utilizar diminutivos
  - No utilizar letras, salvo que tengan significado en el contexto (c en física significa algo)
- Los nombres de los argumentos formales pueden ser letras o diminutivos, pero deben estar documentados en la forma corta
- Los nombres de las entidades locales a los métodos pueden ser más o menos descriptivos en función de su importancia para la comprensión del código





## Normas de documentación

- Las clases deben ser comprensibles en si mismas
- Las clases deben contener toda su documentación (código y comentarios para mantenerlas, forma corta para usarlas)
- Sólo las clases de la interfaz interactúan con el usuario
- La documentación debe escribirse pensando en el lector
- Los identificadores, el sangrado, el espaciado, o incluso el algoritmo elegido forman parte de la documentación
- Los comentarios no deben dar información redundante para un programador en el lenguaje utilizado