TEMA 2

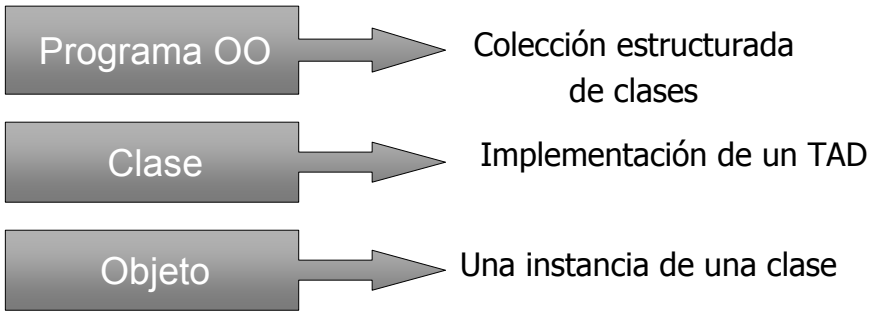
Clases y Objetos

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;
```

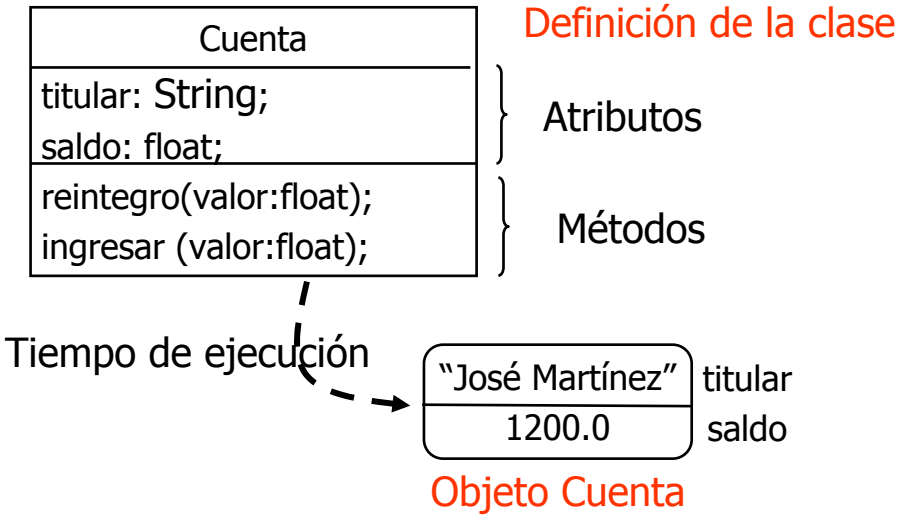
Componentes de un clase

- **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**

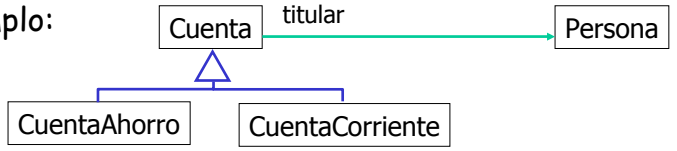
```
class Cuenta{
        Persona titular;
…
}
```

```
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 de definición de clase en Java

ESTRUCTURA

```
package gestionDeCuentas;
import java.util.*;

public class Cuenta {
     //Variable de clase
     private static int  ultimoCodigo;
     //Constantes
     public static final double SALDO_MINIMO=60;

     //Variables de instancia
     private Persona titular;
     private double saldo;
     private int codigo;
     private double [] ultimasOperaciones;

     public Cuenta (Persona quien)   {
            saldo=SALDO_MINIMO;
            titular=quien;
            codigo = nuevoCodigo();
            ultimasOperaciones = new double[20];
     }
```

Ejemplo de definición de clase en Java

```
//Método de clase
private static int nuevoCodigo() {
     return ultimoCodigo++;
}

//Métodos de acceso a los atributos (set/get)
public double getSaldo() {
     return saldo;
}

//Operaciones para modificar el estado del objeto
public void reintegro(double cantidad) {
   if (puedoSacar(cantidad))
            saldo = saldo - cantidad;
}
public void ingreso(double cantidad) {
                  saldo = saldo + cantidad;
}

//Métodos auxiliares
private boolean puedoSacar (double cantidad){
     return (cantidad <= saldo);
}
}//fin clase Cuenta
```

Comportamiento

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:`** accesible desde todas las clases
    - **`private:`** accesible sólo dentro de la clase donde se define
    - ***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 gestionDeCuentas.*;`
    - Notación punto: `gestionDeCuentas.Cuenta;`

3.- Objetos

Definición

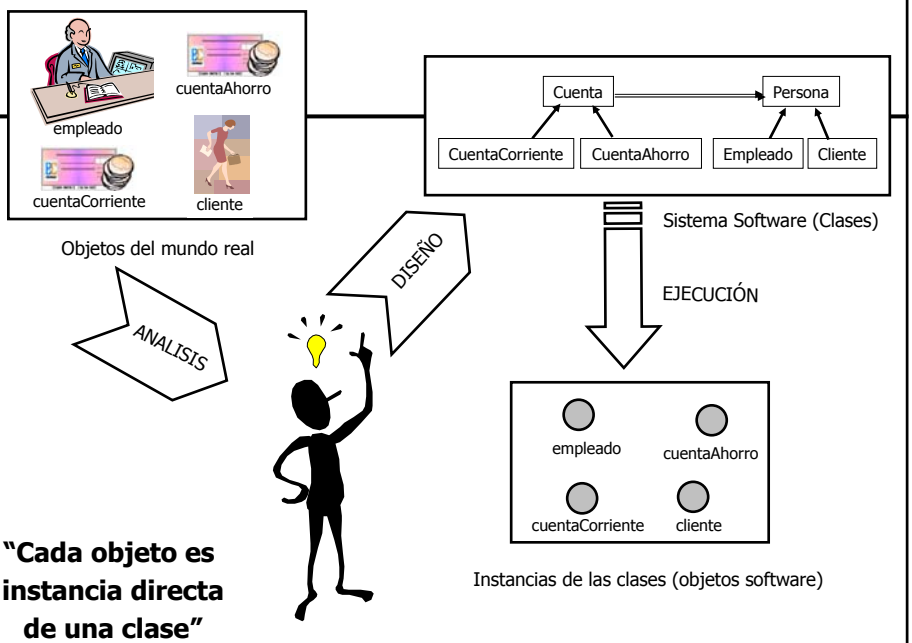
"Es una instancia de una clase, creada en tiempo de ejecución"

- 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",...



"Cada objeto es instancia directa de una clase"

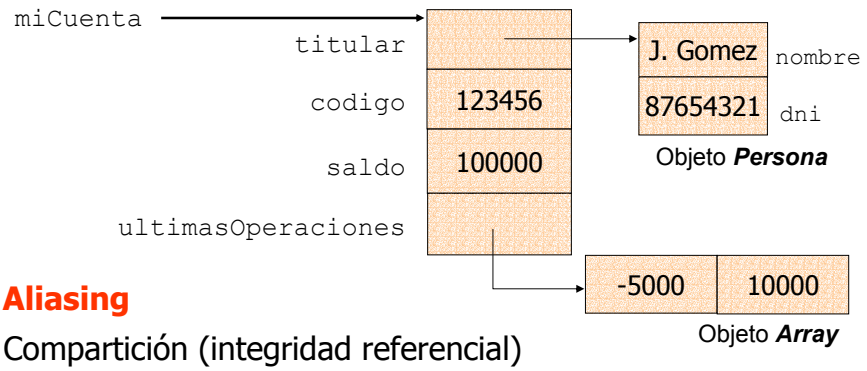
Tipos de campos

- **Tipos de datos primitivos**
  - **byte** (entero de 8 bits)
  - **short** (entero de 16 bits)
  - **int** (entero de 32 bits)
  - **long** (entero de 64 bits)
  - **float** (decimal de 32 bits)
  - **double** (decimal de 64 bits)
  - **char** (Unicode de 16 bits)
  - **boolean** (**true**, **false**)
- **Enumerados: cjto finito de valores**

```
public enum Estado {
     OPERATIVA, INMOVILIZADA, NUMEROS_ROJOS;
}
```

- **Referencias**
  - Sus valores son objetos de tipos no básicos, otras clases.

Referencias



☹ **Aliasing**

☺ Compartición (integridad referencial)

☺ Estructuras recursivas (Autoreferencias)

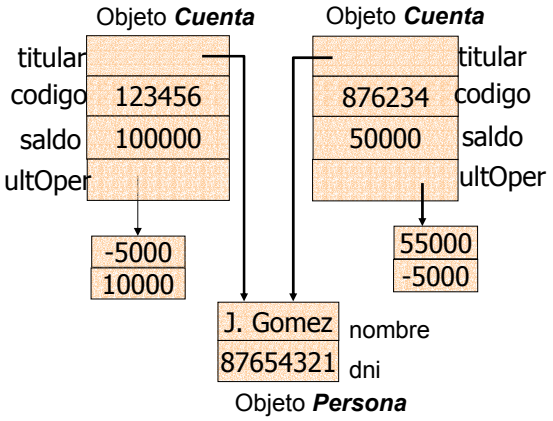
☺ Más eficiente manejo obj. complejos

☺ Objetos se crean cuando se necesitan

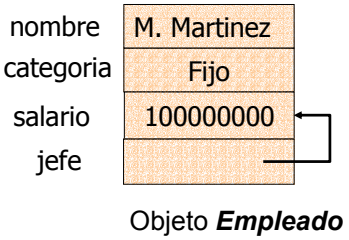
☺ Soporte para el polimorfismo

Ejemplos referencias:

a) Compartición



b) Autorreferencias



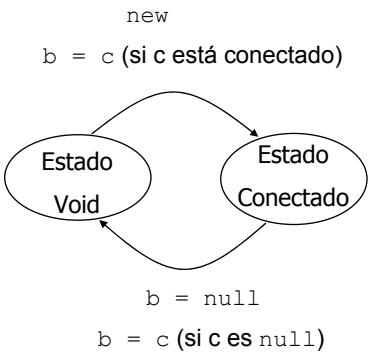
Referencias e identidad de objetos

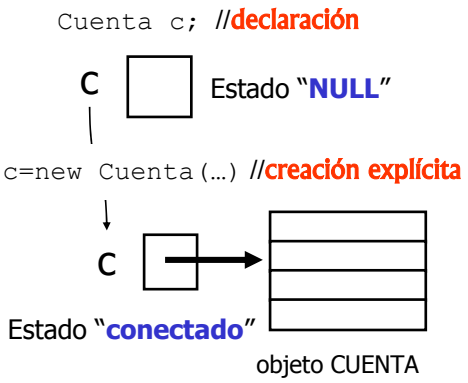
Definición: referencia

Una referencia es un valor en tiempo de ejecución que está o **vacío** (`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.*

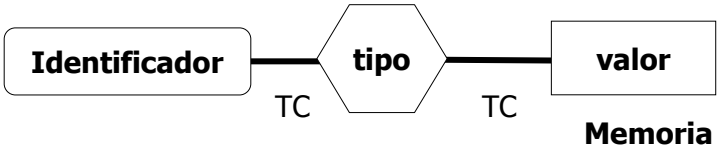
Estados de una referencia



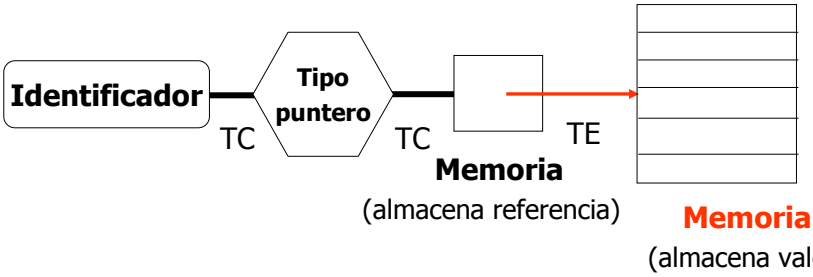


4.- Semántica almacenamiento vs. semántica referencia

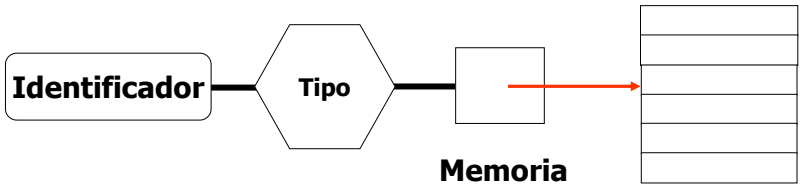
- Variables en los lenguajes tradicionales



- 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**.
  (`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"

5.- Métodos y mensajes

- Un **método** está compuesto por:
  - **Cabecera**: Identificador y Parámetros
  - **Cuerpo**: Secuencia de instrucciones
- **Mensaje**:
  - 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**

Ejemplo método vs. mensaje

- Método `reintegro` en la clase `Cuenta`:

```
public double reintegro (double cantidad) {
      if (puedo_sacar(cantidad))
            saldo = saldo - cantidad;
}
```

- Mensaje, invoca la ejecución del método:

```
cuenta.reintegro(600.0);
```

Objeto receptor | Notación punto | Nombre del método | Argumentos

Definición de Métodos

- ¿Qué **instrucciones** podemos incluir en el cuerpo de un método?
  - Asignación
  - Condicional
  - Iteración
  - Invocación a otro método = **Mensajes**
  - Creación de objetos

Sentencias de control de flujo

- Control de saltos:

```
if( expresión-booleana )
{
        sentencias;
}
 [else {
        sentencias;
}]
```

```
switch(expresión) {
    case valor1:
        sentencias;
        break;
    case valor2:
        sentencias;
        break;
    [default:
        sentencias;]
}
```


Ejemplos

```
1) int saldo;
   …
   if (saldo<0)
      estadoCuenta = Estado.NUMEROS_ROJOS;

2) int dia;
   …
   switch (dia){
      case 1: System.out.println("Lunes"); break;
      case 2: System.out.println("Martes"); break;
      …
      case 7: System.out.println("Domingo"); break;
   }
```

Ejemplo

- Supuesto que en la clase Cuenta añadimos el atributo `Estado estadoCuenta;`

  Podríamos redefinir el método reintegro como sigue:

```
public boolean reintegro (double cantidad){
  switch (estadoCuenta) {
   case INMOVILIZADA:
   case NUMEROS_ROJOS: return false;
   case OPERATIVA: if (puedoSacar(cantidad))
                        saldo = saldo-cantidad;
                   return true;
  }
  return false;
}
```


Bucles

```
for( inicialización; exp-booleana; iteración ) {
      sentencias;
}
```

```
for(Tipo valor : nombreColeccion){
      //hacer algo con "valor"
   }
```

```
[inicialización;]
do {
      sentencias;
      [iteración;]
}while(expresión-booleana );
```

```
[inicialización;]
while( expresión-booleana ) {
      sentencias;
      [iteración;]
}
```

Ejemplos

```
2.  public double getSaldo(){
        double saldo = 0;
        for (int index=0; i<ultimasOperaciones.length; index++)
                saldo = saldo + ultimasOperaciones[index];
        return saldo;
    }
```

Bucle foreach

```
2.  public double getSaldo(){
        double saldo = 0;
        for (double operacion : ultimasOperaciones)
                saldo=saldo+operacion;
        return saldo;
    }
```

```
3.  int index=0;
    while (i< ultimasOperaciones.length){
          saldo = saldo + ultimasOperaciones[index];
          ++i;
    };
```


Sobrecarga de métodos

- Java soporta **sobrecarga de métodos**
  - el mismo nombre pero con **DIFERENTE** lista de argumentos
  - **SIEMPRE** devuelven el mismo tipo

```
public class Cuenta{
        …
        //Pago de una compra en una vez
        public boolean cobrar(Compra ticket){
            return reintegro(ticket.getTotal());
        }
        //Pago a plazos
        public boolean cobrar(Compra ticket,
                              boolean aplazado){
          …
        }
}
```

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"**

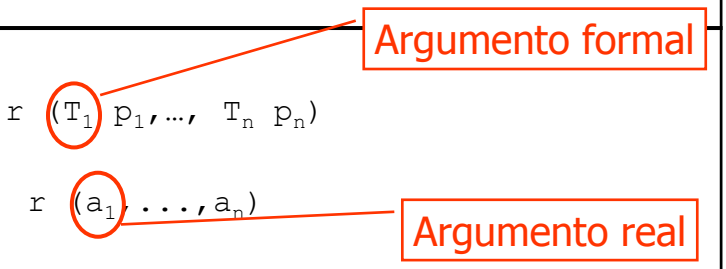
Paso de parámetros

Sea la rutina

r (T_1 p_1,…, T_n p_n)

la invocación

r (a_1,...,a_n)



Las preguntas relevantes son:

- ¿Cuál es la correspondencia entre parámetros reales y formales?
- ¿Qué operaciones se permiten sobre los parámetros formales?
- ¿Qué efecto tendrán éstas sobre los parámetros reales correspondientes?

Paso de parámetros

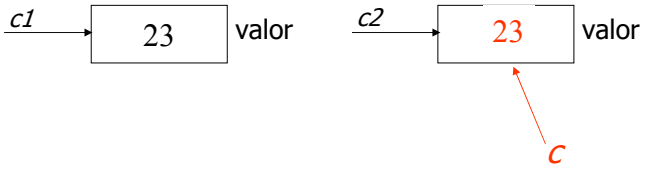
- El efecto del paso de parámetros es: $p_i = a_i$
  - En el caso de las referencias, el argumento formal referencia al mismo objeto referenciado por el argumento real
  - En el caso de los tipos primitivos p_i es una copia de a_i
- **Paso de parámetros** siempre **por valor**
  - tipos primitivos no cambian
  - los objetos pueden cambiar su estado → **se pasa por valor la referencia**
- No existe ninguna restricción sobre las operaciones aplicables sobre los parámetros formales
- Cuando trabajamos con referencias el efecto de una operación sobre el argumento formal implica que se modifique el estado del argumento real.

Paso de parámetros

```
public class Contador {
   int valor;
   …
   public void sincronizar(Contador c){
        c.setValor(valor);
   }
}
```

```
c1.sincroniza(c2);
```

c=c2



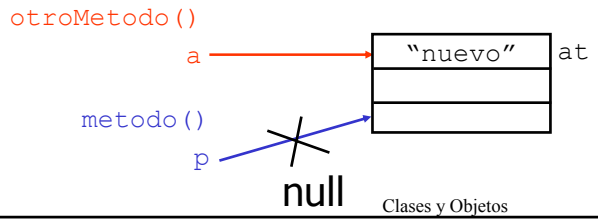
Se puede modificar el estado del objeto pero no la referencia

Paso de parámetros en Java

- Es posible modificar el argumento formal (si no es `final`)
- El argumento real no se cambia porque el argumento formal 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);
}
```

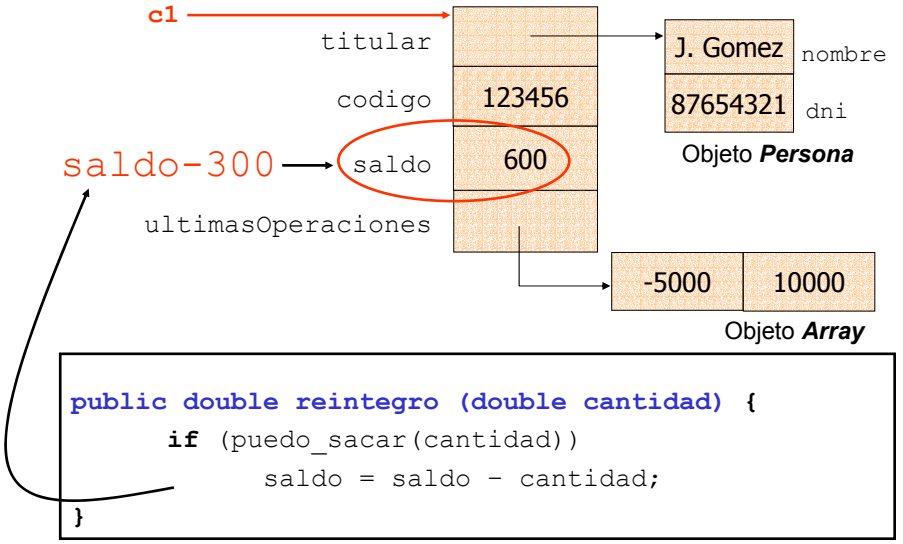


Instancia actual

"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
- La instancia actual es el receptor de la llamada actual, el objeto receptor del mensaje

Instancia actual: `c1.reintegro(300);`



Mensajes y "objeto actual"

```
public class Ejemplo {
   private int at1;
   private ICE2 at2;
   private ICE3 at3;

   public void metodo1(p: int){
       at1 = p;
       at2.unMetodo();
       metodo2();              // 👁¿Quién es el objeto receptor?
   }

   public void metodo2(){
       at3.unMetodo(this);
   }
}
```

Llamada calificada
Objeto Receptor Explícito

Llamada no calificada
No se especifica el obj. receptor

<=> this.metodo2();

???

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 utilizando **`this`**
- ¿Cuándo utilizamos **`this`**?:
  - Pasar el objeto actual como parámetro de otro método

```
servicio.añadir(this);
```

  - En cualquier método de la clase para **mejorar la legibilidad del código**

```
this.otroMetodo();

public void setAutorizado(Persona autorizado){
      this.autorizado = autorizado;
}
```

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".

6.- Creación de Objetos

- Declaración ≠ Creación
- Mecanismo explícito de creación de objetos: `new`
- **¿Por qué?**
  - Evitar cadena de creaciones antes de empezar a hacer nada útil
  - Estructuras recursivas
  - Los objetos se crean cuando se necesitan (referencias vacías, compartir objeto)
- Constructores: dejan el objeto en un estado válido
  - Métodos con el mismo nombre de la clase
  - Se invoca automáticamente cada vez que se crea un objeto de la clase
  - No se pueden invocar una vez que el objeto se ha creado
  - No pueden especificar tipos ni valores de retorno

Constructores

- Permite **sobrecarga** para especificar formas distintas de inicializar los objetos.
- Si no se define, el compilador crea uno **por defecto** sin argumentos que inicializa los atributos a los valores por defecto (ver tabla).
- El programador también puede definir un constructor sin argumentos.

Valores por defecto

Tipo	Valor Inicial
boolean	false
char	'\u000'
byte,short,int,long	0
float	+0.0f
double	+0.0d
Referencia a objeto	null

Constructores para la clase Cuenta

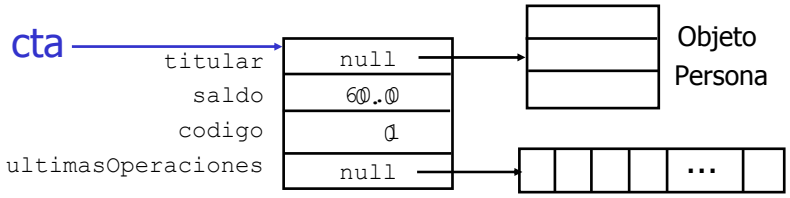
```
public class Cuenta{
   …
   public Cuenta(){      //Constructor por defecto
        codigo = nuevoCodigo();
        ultimasOperaciones = new double[20];
        // El resto de atributos toman los valores por defecto
   }
   public Cuenta (Persona quien){
        this();
        saldo=SALDO_MINIMO;
        titular=quien;
   }

   public Cuenta(Persona quien, double saldoInicial){
          this(quien);
          saldo = saldoInicial;
   }
   …
}
```

this se utiliza para invocar a otro constructor

Creación de objetos

```
Cuenta cta = new Cuenta(p);
```

1) Crea una nueva instancia de la clase Cuenta.
2) Inicializa los campos de la instancia con los valores por defecto
   - Garantiza que cada atributo de una clase tenga un **valor inicial** antes de la llamada al constructor
3) Se aplica sobre la instancia el constructor que se invoca.
4) Conecta `cta` a la instancia creada.



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".

Principios de diseño de clases

- **Principios de diseño modular:**
  - Alta Cohesión, Bajo Acoplamiento, Ocultamiento de la Información, Abierto-Cerrado, Elección Única.
- **Experto en Información:**
  - Asignar una responsabilidad al experto en información, la clase que tiene la información necesaria para llevar a cabo la responsabilidad.
- **Ley de Demeter**: "*Habla sólo con tus amigos*"
  - para un método **m** de una clase **C** sólo deberían invocarse los métodos:
    - de la clase **C**
    - de los parámetros que recibe el método **m**
    - de cualquier objeto creado en el método **m**
    - de cualquier atributo (variable de instancia) de la clase **C**

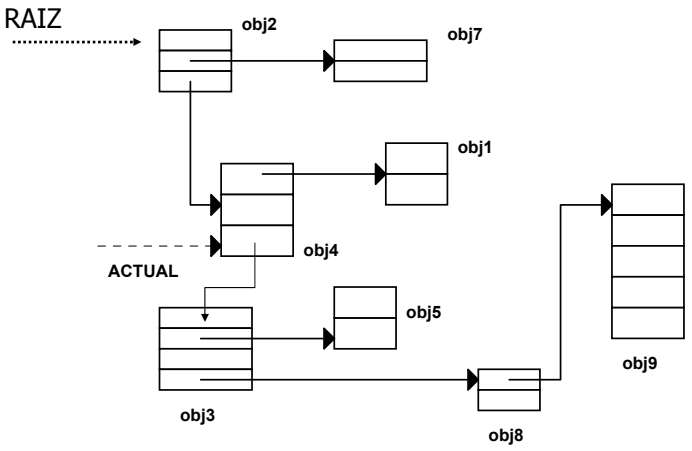
7.- Modelo de ejecución OO

- 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]+" ");
  }
}
```

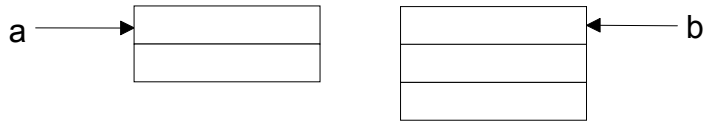
- Los parámetros en el array de cadenas de texto son los *parámetros del programa*

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

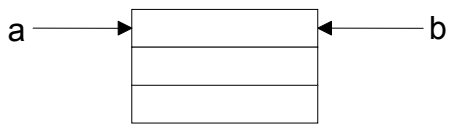

8.- Operaciones sobre referencias:

A) Asignación

- La asignación no implica copia de valores sino de referencias
- Cuidado con el "**aliasing**"



a = b



Consecuencia del aliasing

```
//P(b) es cierto
        a = b;
        a.rutina();        // rutina no afecta a b
//P(b) puede ser falso
```

Ejemplo 1:

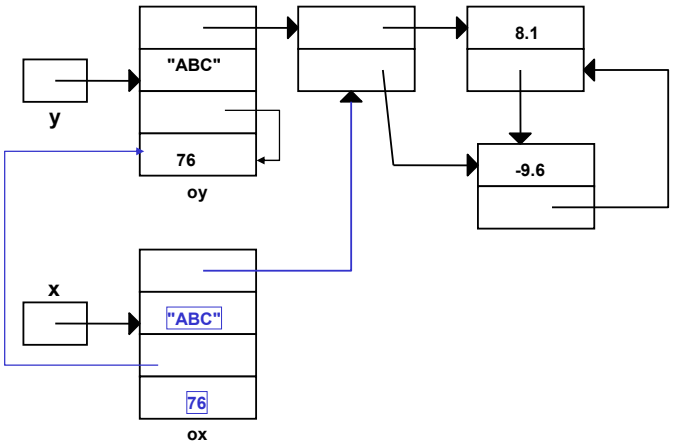
```
public class C  {
    boolean atrib;
    ...
    public void setAtrib(boolean valor){
        atrib = valor;
    }
}
```

```
C x,y;
//y != null && y.atrib=true
        x = y;
        x.setAtrib(false);
// y.atrib = false
```

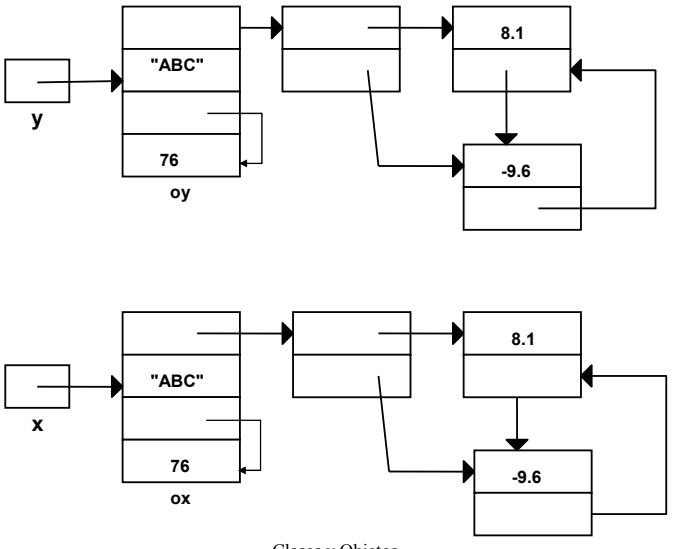

Aliasing y el Principio de Ocultamiento de la Información

- ¡¡Ojo!! Cuando devolvemos objetos que forman parte de la implementación de una clase.
- Por ejemplo, `getUltimasOperaciones`
  - debe **devolver una copia** de la colección, no la colección.
  - Si devuelve la colección el cliente de la clase `Cuenta` podría acceder a la implementación para modificarla.
  - Las modificaciones de la colección se deben hacer SIEMPRE desde métodos de la clase cuenta, nunca desde los clientes de la clase.

(B.1) Copia superficial de un objeto



(B.2) Copia profunda de un objeto



Copia vs. Clone

Supongamos las variables `ox, oy` de tipo `C`:

- En una copia ambos objetos existen previamente y con valores distintos de `null`
  - **`ox.copy(oy);`**
  - En Java no existe el método `copy` disponible para todo objeto.
- Una alternativa es clonar un objeto y el resultado de la clonación asignárselo a una variable:
  - **`ox = oy.clone();`**
  - Sería equivalente a:

```
ox = new C();
ox.copy(oy);
```

  - En Java el método clone está disponible para todo objeto.
    - La implementación por defecto es un clone superficial

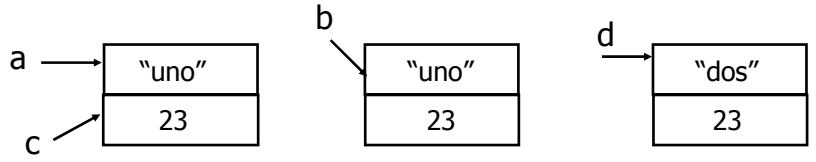
Copia y clonación de objetos en Java

- 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`**

(C) Igualdad de objetos vs. identidad



- **Igualdad entre referencias (Identidad)**

```
a == c          {true}
a == b            {false}
```

- **Igualdad entre objetos**

```
a.equals(b)     {true}
a.equals(d)     {false}
```

- De lo que se deduce que:

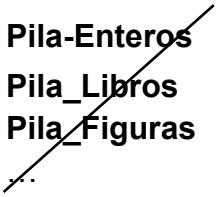
`a == b` ⇒ **`a.equals(b)`**
`a.equals(b)` ⇏ **`a==b`**

(C.2) Igualdad profunda

- **Dos referencias** x e y son iguales en profundidad si:
  1) x=y=null
  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 `null` todos los campos-referencia de *ox* e oy *son* iguales.
  3) Para cada campo-referencia de *ox* con valor `null`, el correspondiente campo de *oy* es `null`
  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.

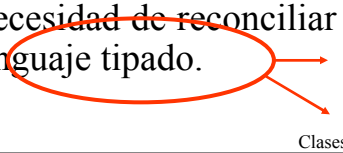
9.- Genericidad

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



⇒ **Pila de?**

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



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

```
public class Pila<T>{
    //T es el parámetro genérico formal
  private ArrayList<T> contenido;
  …
  public boolean isEmpty(){…}
  public void push (T item){ … }
  public T pop() {…}
  //Devuelve el tope sin desapilarlo
  public T tope(){…}
}
```


Uso de una clase genérica

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

1) Una de las clase que encapsulan a los **tipos primitivos**

```
Pila<Integer> pilaEnteros;
```

2) Un **tipo referencia**

```
Pila<Punto> pilaPuntos;
Pila<Pila<Punto>> pilaDePilasPuntos ;
```

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

```
class Pila <T>  {
      ArrayList<T> contenido;
      ....
}
```

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.

```
Pila <Punto> pp;
Pila <Cuenta> pc;
Punto p;
Cuenta c;
```

MENSAJES VALIDOS

```
pp.push(p);
pc.push(c);
p = pp.pop();
```

MENSAJES NO VALIDOS

```
pp.push(c);
pc.push(p);
p = pc.pop();
```

- La genericidad SÓLO tiene sentido en un LENGUAJE TIPADO

Operaciones sobre entidades de tipos genéricos

Sea la clase:

```
public class C <T,G,...> {
      private T x;
      public void metodo (G p){ … }
            ...
}
```

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

En una clase cliente, **T, G, …** 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 T)
  2) **`y = x`** (y es una entidad de tipo T)
  3) **`x==y o x!=y`** (y es de tipo T)
  4) **`a.f(...,x,...)`** (x actúa como argumento en un mensaje, el correspondiente parámetro es de tipo T u `Object`)
  5) `x.clone()` o `x.equals(y)` (rutinas de `Object`)

- **¡¡No se permite la creación!!** **`T at = new T();`**

Ejemplo: "Correo electrónico"

```
public class BuzonCorreo {
   private List <Carpeta> carpetas;
   public List<Mensaje> buscar (String s){
        for (Carpeta c : carpetas){
          mensajes = c.buscar(s);
         if (!mensajes.isEmpty()) resultado.addAll(mensajes);
        }
       return resultado;
   }    ...
}

public class Carpeta {
   List <Mensaje> mensajes;
   List <Mensaje> buscar (String s){
              for (Mensaje m : mensajes)
                  if (m.buscar(s))  resultado.add(m);
              return resultado;
   }    ...
}

public class Mensaje{
   String contenido;
   …
   boolean buscar (String s){
        return (contenido.equals(s));
   }    ...
}
```

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?