



# Tema 2: Clase y objetos en Java

---

Programación Orientada a Objetos

Curso 2009/2010

Begoña Moros Valle



**DIS**

Departamento de  
Informática y Sistemas



# Índice

---

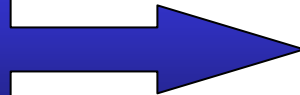
- Introducción
- Clases
- Objetos
- Semántica referencia
- Métodos y mensajes
- Creación de objetos
- Modelo de ejecución OO
- Genericidad
- Principios de diseño de clases



# Introducción

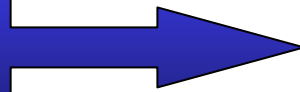
---

Programa OO



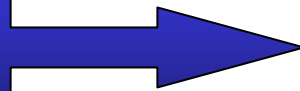
Colección estructurada  
de clases

Clase



Implementación de un  
Tipo Abstracto de Datos (TAD)

Objeto



Una instancia de una clase

Los objetos se comunican mediante **mensajes**



# 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
  - **Tipo** (concepto semántico)
    - Mecanismo de definición de nuevos tipos de datos: describe una estructura de datos (objetos) y las operaciones aplicables.



# Módulo $\neq$ Tipo. Ejemplo Modula2

```
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;  
  ...  
  END pop;  
  ...  
END Pila;
```

# Componentes de un clase

- **Atributos:**

- Determinan una estructura de almacenamiento para cada objeto de la clase

- **Métodos:**

- Operaciones aplicables a los objetos
- Único modo de acceder a los atributos

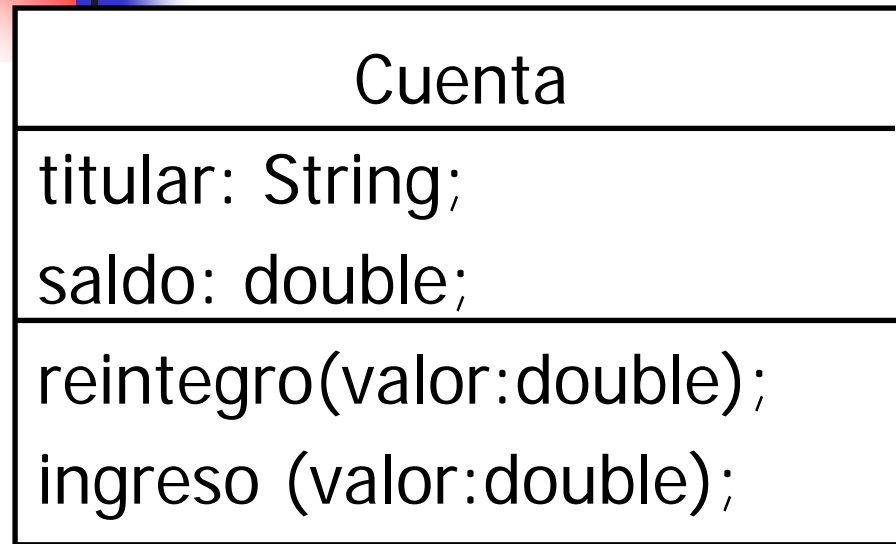
- **Ejemplo:** En una aplicación bancaria, encontramos objetos "cuenta". Todos los objetos "cuenta" tienen propiedades comunes:

- atributos: saldo, titular, ...
- operaciones: reintegro, ingreso, ...



Definimos una **clase CUENTA**

# Ejemplo: Clase Cuenta

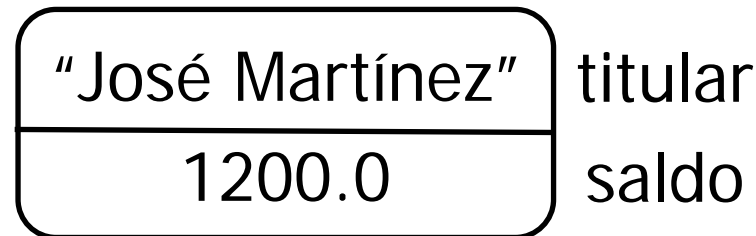


Definición de la clase

Atributos

Métodos

Tiempo de ejecución



Objeto Cuenta





# Clase Cuenta en Java

```
class Cuenta{
```

```
String titular;  
double saldo;
```

**ATRIBUTOS**

```
void ingreso (double cantidad){  
    saldo = saldo + cantidad;  
}  
void reintegro (double cantidad){  
    if (cantidad <= saldo)  
        saldo = saldo - cantidad;  
}
```

**MÉTODOS**

```
}
```



# Ocultación de Información

---

- A las características de una clase (atributos y métodos) se les puede asignar un modificador de visibilidad:
  - **public**:
    - Característica pública, accesible desde todas las clases
  - **private**:
    - Característica privada, accesible sólo dentro de la clase donde se define
- Principio de diseño:
  - Todos los atributos de una clase son privados
  - Los métodos pueden tener distintos niveles de visibilidad



# Clase Cuenta en Java

```
class Cuenta{
    private String titular;
    private double saldo;

    public void ingreso (double cantidad){
        saldo = saldo + cantidad;
    }
    public void reintegro (double cantidad){
        if (puedoSacar(cantidad))
            saldo = saldo - cantidad;
    }
    private boolean puedoSacar(double cantidad){
        return cantidad <= saldo;
    }
}
```



# Acceso y modificación de atributos

---

- La única forma de acceder a los atributos es a través de métodos públicos
  - Métodos **get** → para consultar el valor del atributo
  - Métodos **set** → para establecer el valor del atributo
- La implementación de los métodos set/get depende del nivel de acceso de los atributos.
- Se aísla al cliente de los cambios en la estructura de datos.



# Clase Cuenta en Java

---

```
class Cuenta{
    private String titular;
    private double saldo;

    public String getTitular(){
        return titular;
    }
    //No setTitular porque es inmutable

    public double getSaldo(){
        return saldo;
    }
    /* No setSaldo porque se modifica con ingresos
       y reintegros */
}
```



# Paquetes

---

- Unidad modular del lenguaje que permite agrupar clases que están relacionadas
- Además de `public` y `private`, se puede establecer que una característica de la clase tiene **visibilidad a nivel de paquete**
  - Visible a todas las clases del mismo paquete
  - Visibilidad por defecto
- A las clases también se les puede especificar un nivel de visibilidad:
  - `public`: la clase es visible desde cualquier paquete
  - A nivel de paquete: la clase sólo es visible en el paquete al que pertenece.



# Paquetes


---

- La pertenencia de una clase a un paquete se debe especificar antes de la declaración (**package**)
- Para utilizar una clase definida en un paquete distinto:
  - Se utiliza el **nombre cualificado** de la clase
    - Ejemplo: `java.util.LinkedList unaLista;`
  - Se importa la clase o el paquete al comienzo de la declaración
    - Se puede utilizar el nombre de la clase sin cualificar
    - **import** `java.util.LinkedList;` o bien
    - `import java.util.*;`



# Clase Cuenta en Java

---

```
package banco.cuentas;  Anidamiento de paquetes!!  
import java.util.LinkedList;  
  
public class Cuenta{  
    String titular;  
    double saldo;  
    LinkedList ultimasOperaciones;  
  
    void ingreso (double cantidad){...  
    }  
    void reintegro (double cantidad){...  
    }  
}
```





# Objetos

---

- Un objeto 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.
- Los métodos permiten consultar y modificar el estado del objeto.

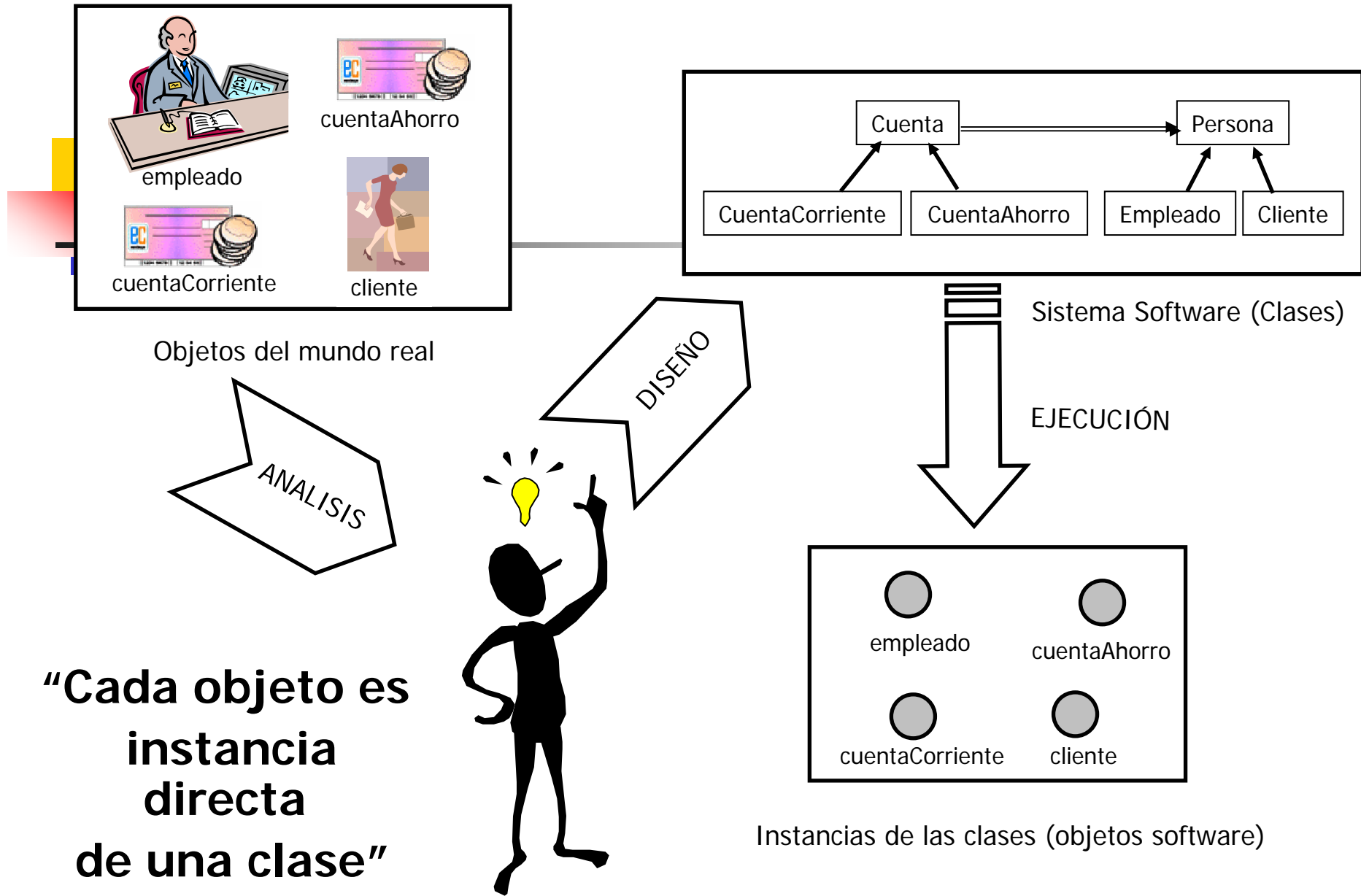


# Objetos dominio vs. Objetos aplicación

---

## 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 los atributos

---

- Tipos de datos primitivos:
  - Enteros: `byte`, `short`, `int`, `long`
  - Reales: `float`, `double`
  - Carácter: `char`
  - Booleano: `boolean`
- **Referencias:**
  - Sus valores son objetos de tipos no básicos, otras clases.
  - **Enumerados:** son clases que representan un conjunto finito de valores



# Enumerados

---

```
enum EstadoCuenta{  
    OPERATIVA, INMOVILIZADA, NUM_ROJOS;  
}
```

```
public class Cuenta{  
    private String titular;  
    private double saldo;  
    private EstadoCuenta estado;  
    ...  
}
```

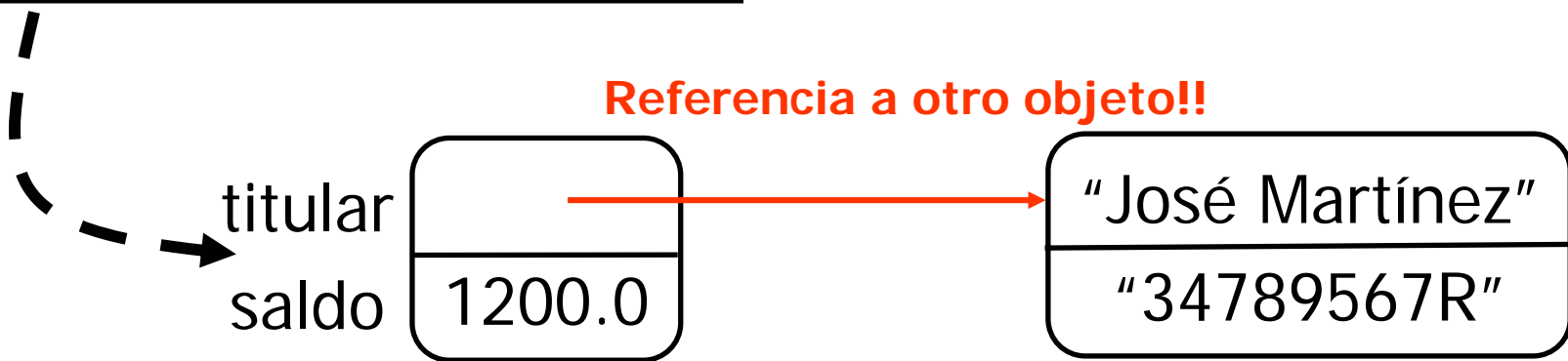
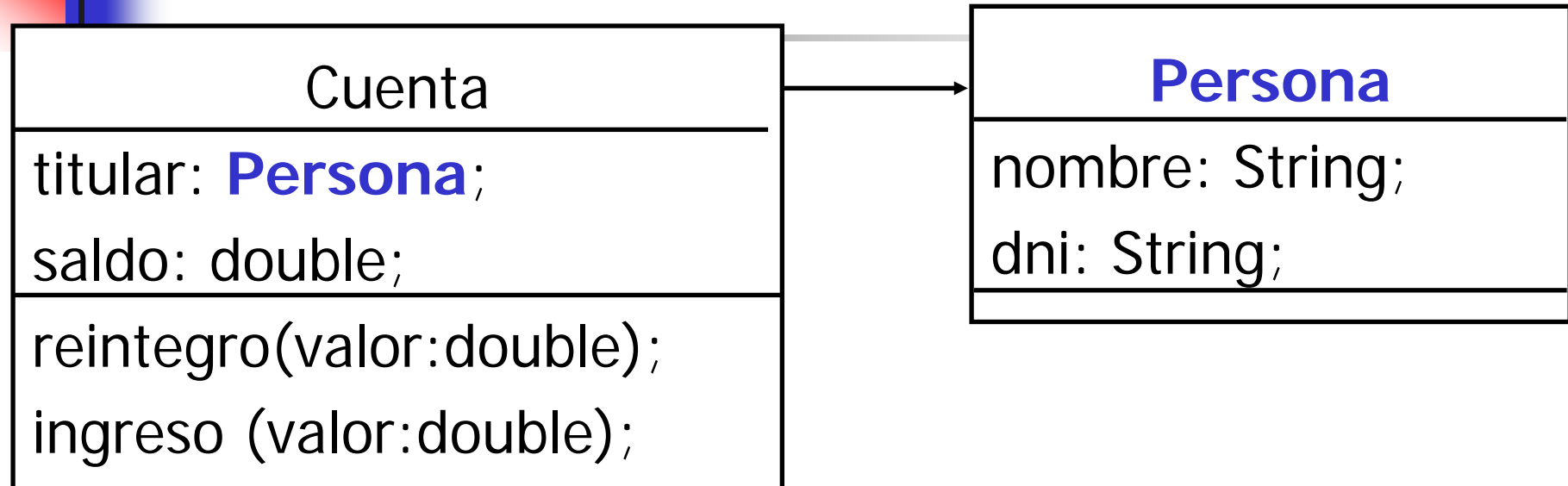


# Relación de clientela

---

- Cuando en una clase A establecemos que el tipo de un atributo es otra clase B, decimos que **A es cliente de B**.
  
- Por ejemplo:
  - Definimos la clase `Persona`
  - Declaramos el tipo del atributo `titular` (en la clase `Cuenta`) como `Persona`.
  - La clase `Cuenta` es cliente de la clase `Persona`

# Relación de clientela



Objeto Cuenta

Objeto Persona



# Semántica referencia

---

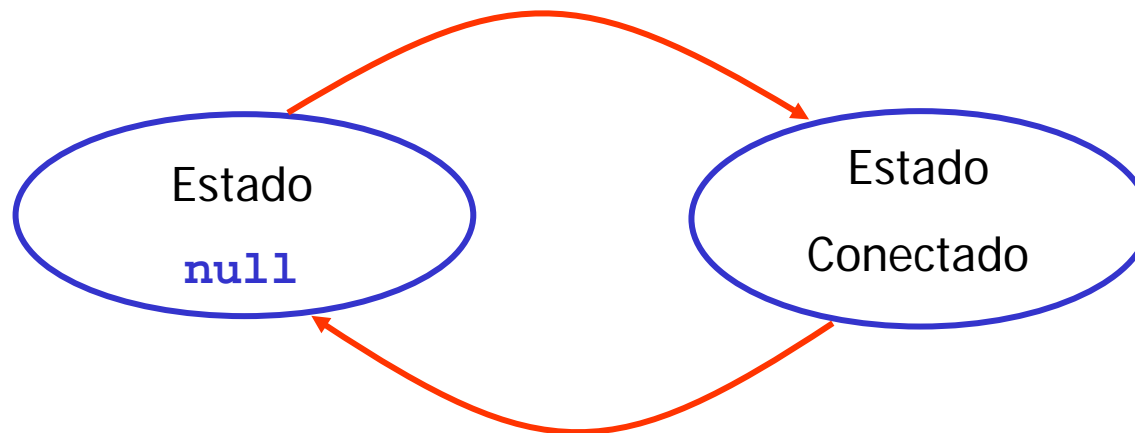
- Una referencia es un valor que en tiempo de ejecución está o vacío (**null**) o conectado.
- Si está conectado, una referencia identifica a un único objeto.
- Mientras exista, cada objeto posee una identidad única, independiente de su estado → **identificador de objeto** (oid):
  - Dos objetos con 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.
- Cuando se asigna un objeto a una variable no se asigna la estructura de datos del objeto sino el oid.



# Estados de una referencia

**Cuando se crea el objeto!!!**

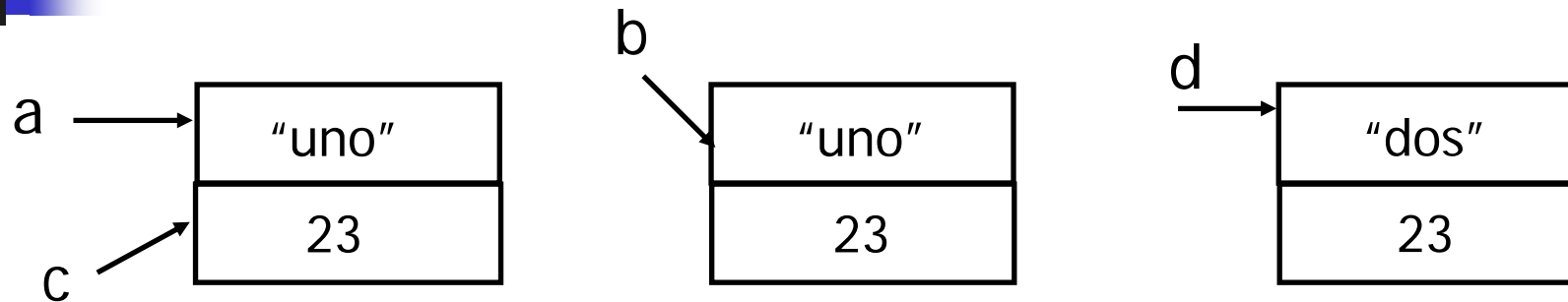
`b = c` (si `c` está conectado)



`b = null`

`b = c` (si `c` es `null`)

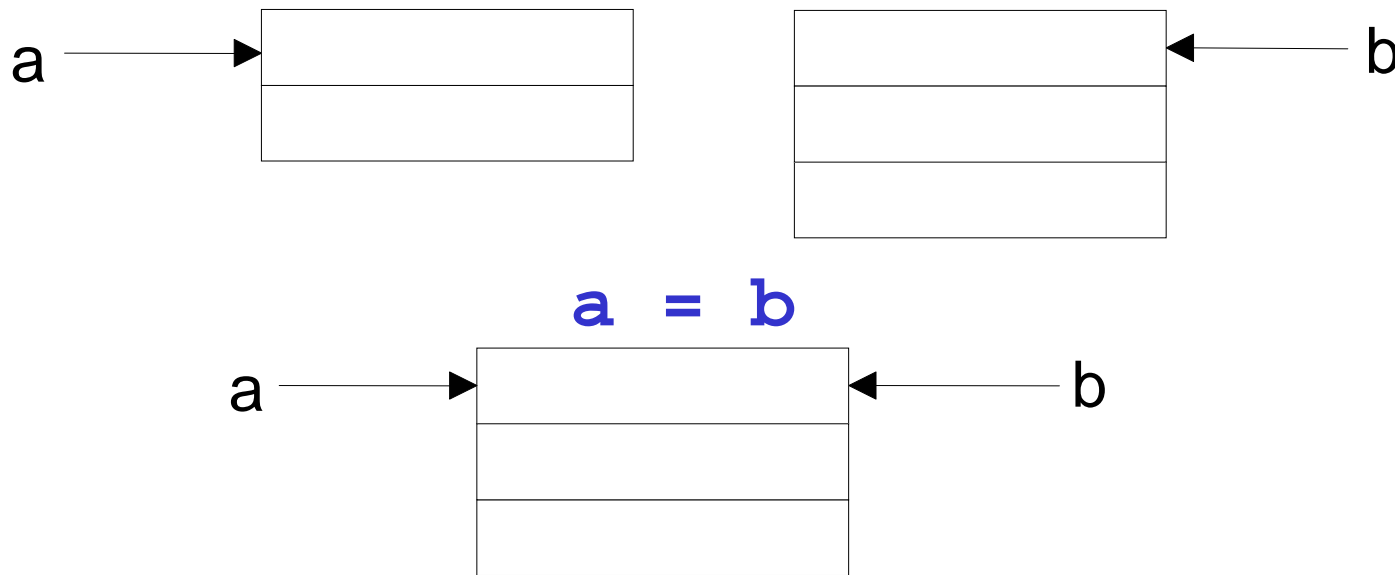
# Igualdad vs. identidad



- Igualdad entre referencias → Identidad
  - `a == c` {true}
  - `a == b` {false}
- Igualdad entre objetos
  - Podemos utilizar el método `equals`

# Asignación de referencias

- La asignación de referencias no implica copia de valores sino de oids
- Problema: "**aliasing**"





# Aliasing

---

```
Cuenta cuenta1;  
Cuenta cuenta2;  
...  
  
double saldo = cuenta1.getSaldo();  
cuenta2 = cuenta1;  
cuenta2.reintegro(1000.0);  
  
// cuenta1.getSaldo() != saldo !!
```



# Copia de objetos

---

- Si la asignación no implica copia de objetos ¿cómo se pueden copiar?
  - `copia = obj.clone();`
  - **Constructor de copia:**
    - Se pasa como parámetro un objeto de la misma clase
    - Se inicializan los campos del nuevo objeto con los mismos valores de los campos del objeto que se pasa como parámetro.

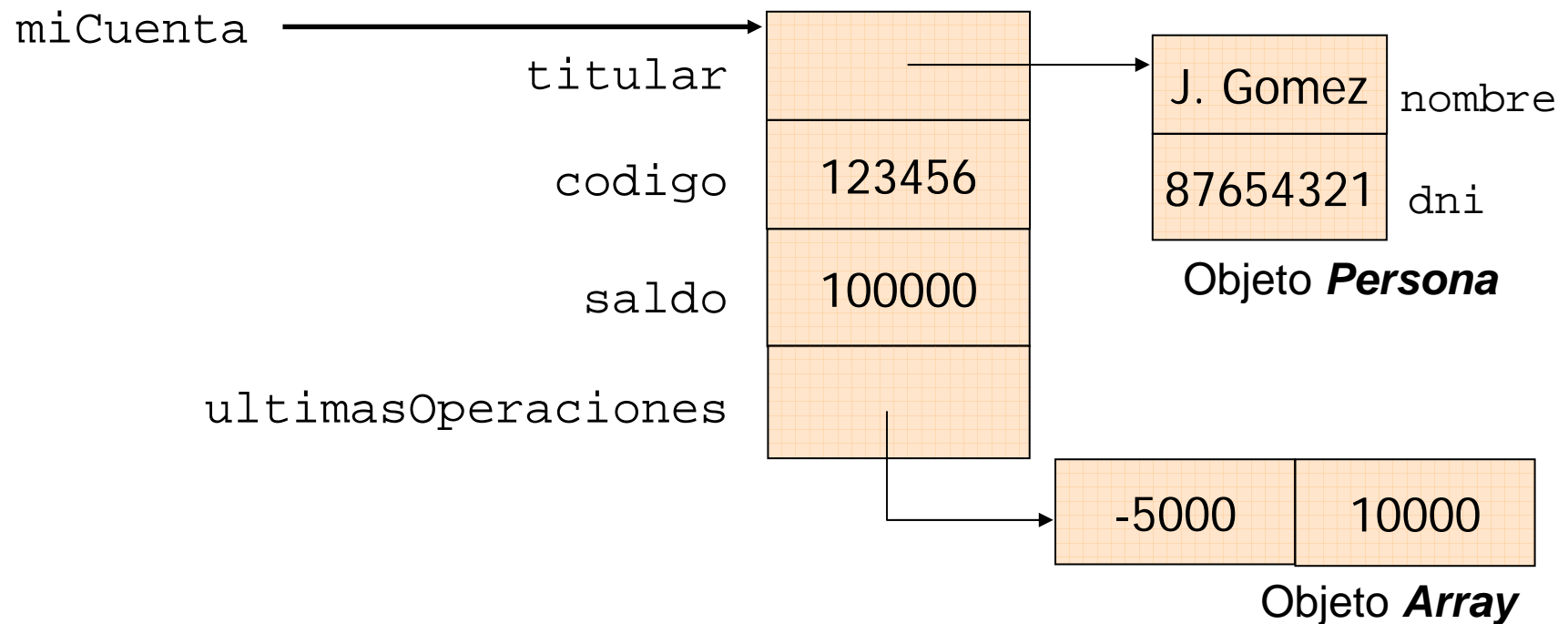


# Semántica referencia

---

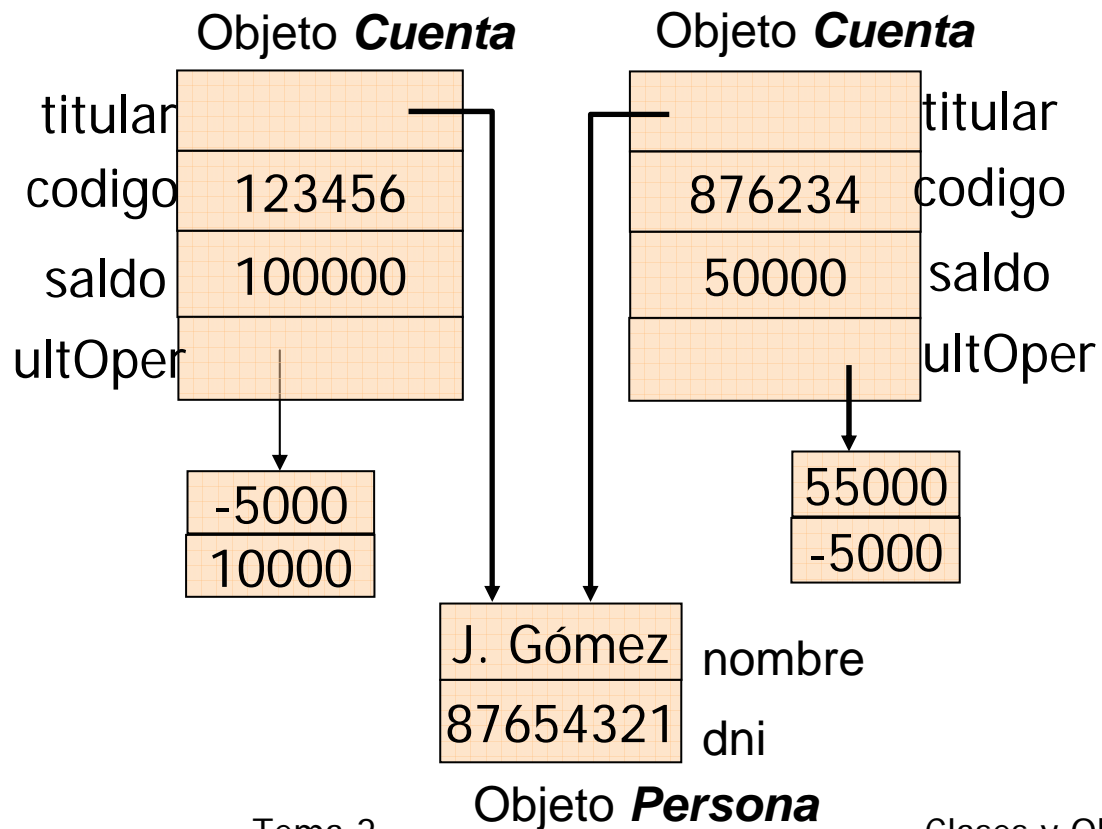
- Ventajas de las referencias:
  - Compartición de objetos → integridad referencial
  - Permite definir estructuras recursivas (auto-referencias)
  - Más eficiente manejo objetos complejos
  - Los objetos se crean cuando se necesitan
  - Soporte para el polimorfismo (Tema 3)
- Inconvenientes:
  - **Aliasing**

# Semántica referencia

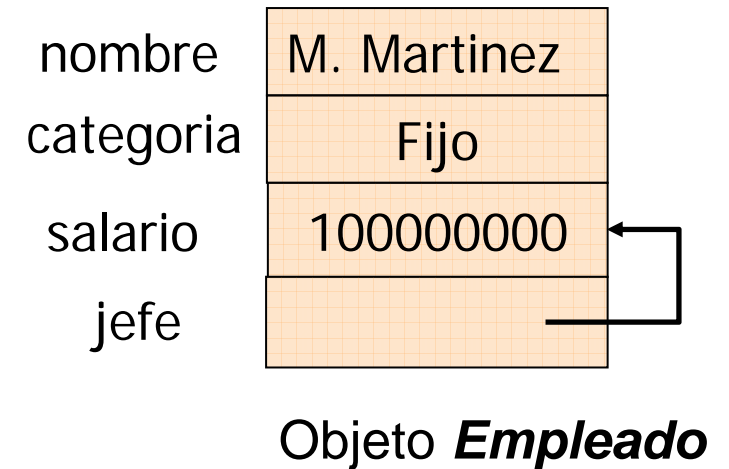


# Semántica referencia

## Compartición



## Autorreferencias







# Referencia vs. punteros

---

- Referencias y punteros son conceptos muy próximos pero diferentes
- Referencias se asocian a **objetos**.
  - Toda referencia tiene un tipo
  - `null` representa el estado no conectado
- Punteros se asocian a **direcciones de memoria**.
  - `null` en C es un valor de tipo puntero
- Una variable denota una referencia a un objeto



# 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.
- Un mensaje está 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 (puedoSacar(cantidad))  
        saldo = saldo - cantidad;  
}
```

- Mensaje, aplica el método reintegro sobre un objeto cuenta:

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



# Mensajes vs. Procedimientos

---

- **¡No confundir con la invocación de un procedimiento en un lenguaje imperativo!**
  - Un mensaje parece una llamada a procedimiento en la que sólo cambia el formato:
    - Mensaje → `unaCuenta.ingreso (100000)`
    - Procedimiento → `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**”



# Definición de Métodos

---

- Todo método tiene un valor de retorno
  - Si no devuelve nada se indica con `void`
- Para cada método se establece el nivel de visibilidad
- ¿Qué instrucciones podemos incluir en el cuerpo de un método?
  - Asignación
  - Estructuras condicionales
  - Iteración
  - Invocación a otro método = Mensajes
  - Creación de objetos



# Sentencias de control de flujo

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

Tema 2

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

Clases y Objetos

38



# Ejemplos

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

```
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

---

- Método reintegro teniendo en cuenta el estado de la cuenta:

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





# Bucles (1/2)

---

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

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



# Ejemplos

---

```
public class Cuenta{
    ...
    private double[] ultimasOperaciones;

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

**LOS ARRAYS SON OBJETOS!!!**



# Bucles (2/2)

---

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

## Bucle ForEach

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



# Ejemplos

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

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



# Sobrecarga de métodos

- Java soporta **sobrecarga de métodos**
  - el mismo nombre pero con **DIFERENTE** lista de tipos de argumentos.
  - No pueden diferenciarse únicamente en el tipo de retorno.

```
//Pago de una compra en una vez
public boolean cobrar(Compra ticket){
    return reintegro(ticket.getTotal());
}

//Pago a plazos → devuelve el número de plazos
public int cobrar(Compra ticket, boolean aplazado){
    ...
}
```



# Paso de parámetros

---

- Sea el método
  - `met (T1 p1, ..., Tn pn)`
  - Donde p<sub>1</sub> ... p<sub>n</sub> son los **parámetros formales**
- y la invocación (mensaje)
  - `obj.met (a1, ..., an)`
  - Donde a<sub>1</sub> ... a<sub>n</sub> son los **parámetros reales**
- Debemos responder a las siguientes preguntas:
  - ¿Cuál es la correspondencia entre parámetros reales y formales?
  - ¿Qué operaciones se permiten sobre los parámetros formales?
  - ¿Qué efecto tendrán las operaciones aplicadas sobre los parámetros formales sobre los parámetros reales correspondientes?



# Paso de parámetros

---

- El efecto del paso de parámetros es una asignación →  
 $p_i = a_i$ 
  - 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**
  - Al parámetro real no le afectan los cambios en el parámetro formal
- Cuando trabajamos con referencias el efecto de una operación sobre el parámetro formal implica que se **modifique el estado** del parámetro real.
  - Paso por valor de la referencia!!!



# Paso de parámetros

---

- No existe ninguna restricción sobre las operaciones aplicables sobre los parámetros formales
- Es posible modificar el parámetro formal
- El parámetro real no se cambia porque el parámetro formal era una copia de la referencia o del tipo primitivo.



# Paso de parámetros

```
public void transferencia(Cuenta emisor, Cuenta receptor,  
                        double cantidad) {
```

```
    emisor.reintegro(cantidad);  
    receptor.ingreso(cantidad);  
    emisor = null;
```

```
}
```

**Cambia el estado  
de los dos objetos**

**No afectaría a la  
variable cuenta1**

...

```
banco.transferencia(cuenta1, cuenta2, 3000.0);
```

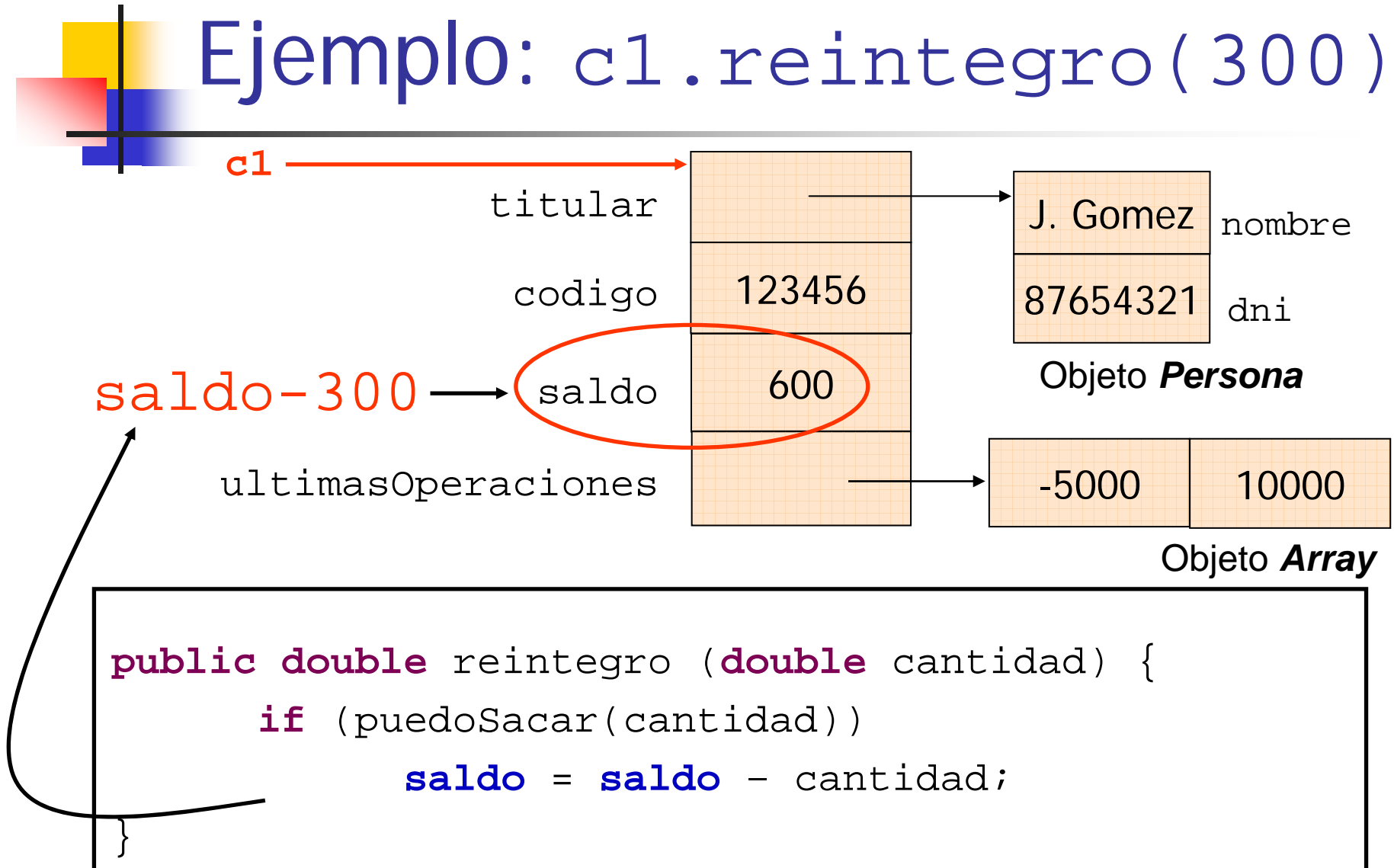


# Instancia actual

---

- 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 sobre la que se aplica
- La instancia actual es el receptor de la llamada actual, **el objeto receptor del mensaje**

# Ejemplo: `c1.reintegro(300)`





# Instancia actual

---

- Si se aplica un método y no se especifica el objeto receptor, se asume que es la instancia actual.

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

- El objeto receptor de `puedoSacar` será el objeto receptor del `reintegro`



# Referencia `this`

---

- El lenguaje Java proporciona la palabra clave `this` que **referencia a la instancia actual**.
- **Utilidad:**
  - Distinguir los atributos de los parámetros y variables locales dentro de la implementación de un método.
  - Aplicar un mensaje a otro objeto estableciendo como parámetro la referencia al objeto actual.

# Referencia this

```
public void trasladar(Sucursal sucursal){  
    this.sucursal.eliminar(this);  
    sucursal.añadir(this);  
}
```

Se refiere al atributo de la clase

Se refiere al parámetro

Se refiere a la instancia actual, la cuenta que se va a trasladar



# 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.



# Creación de Objetos

---

- La **declaración** de una variable cuyo tipo sea una clase **no implica la creación del objeto**.
- Se necesita un 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)

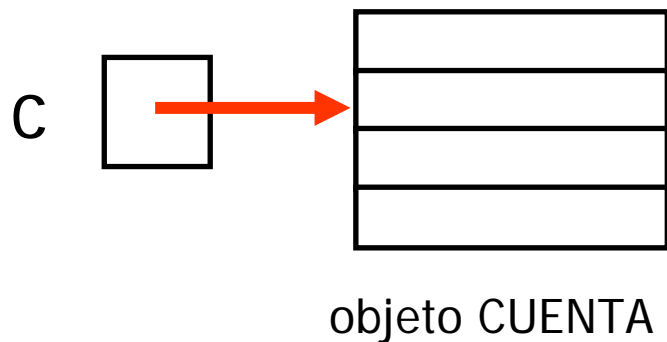


# Declaración vs. Creación

```
Cuenta c; // declaración
```

C  Estado **null**

```
c = new Cuenta(...) // creación explícita
```



- Estado "**conectado**"
- `c` contiene la referencia al objeto
- `c` almacena el oid asignado al objeto al crearse



# Constructores

---

- Método encargado de inicializar correctamente los objetos
- Métodos con el mismo nombre de la clase pero sin valor de retorno
- No se pueden invocar una vez que el objeto se ha creado
- Permite **sobrecarga** para especificar formas distintas de inicializar los objetos
- Toda clase tiene que tener al menos un constructor
- Si no se define ningún constructor, el compilador crea uno **por defecto** sin argumentos, vacío, que inicializa los atributos a los valores por defecto.



# Inicialización por defecto

Tipo	Valor Inicial
boolean	false
char	Carácter 0 ('\u0000')
byte, short, int, long	0
float	+0.0f
double	+0.0d
Referencia a objeto	null



# Constructores para la clase Cuenta

---

```
public Cuenta(Persona quien) {  
    //Utilizamos this para invocar al otro constructor  
    // → reutilización de código  
    → this(quien, 100);  
}
```

```
public Cuenta(Persona quien, double saldoInicial) {  
    titular = quien;  
    saldo = saldoInicial;  
    ultimasOperaciones = new double[20];  
}
```

→ El array hay que crearlo!!!



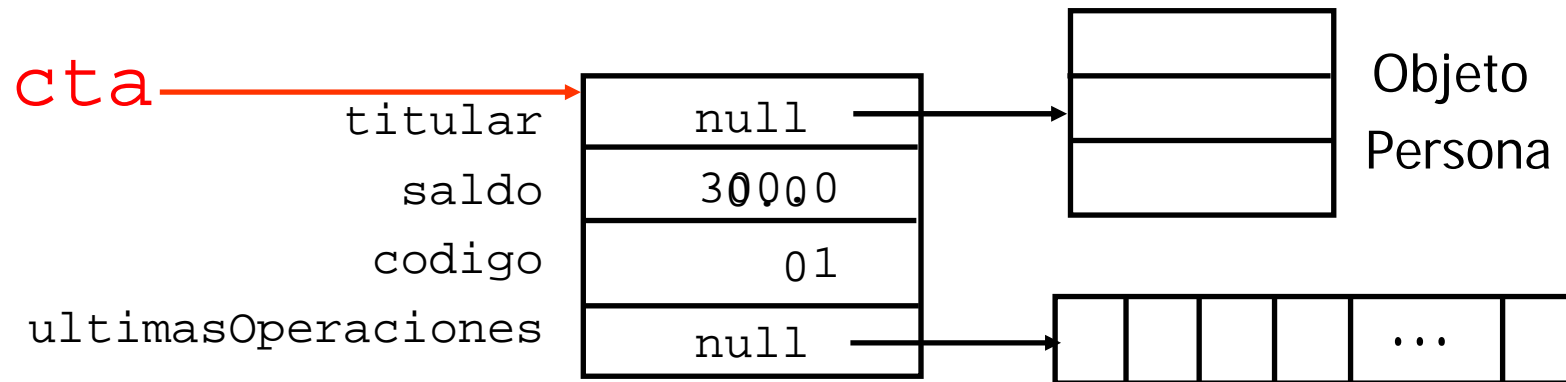
# Creación de objetos

---

- La construcción de un objeto consta de tres etapas:
  - Se reserva espacio en memoria para la estructura de datos que define la clase.
  - 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
  - Se aplica sobre la instancia el constructor que se invoca

# Creación de objetos

```
Cuenta cta = new Cuenta (persona, 300.0);
```





# Atributos finales

---

- Java permite especificar que el valor de un atributo no podrá variar una vez construido el objeto
- Un atributo se declara de sólo consulta anteponiendo el modificador **final** a su declaración
- Los atributos finales sólo pueden ser inicializados en la declaración o en el constructor

# Atributo final

```
public class Cuenta {  
  
    //Los atributos se pueden inicializar  
    //en el momento de la declaración  
    private double saldo = 100; ←  
  
    private final Persona titular;  
  
    public Cuenta(Persona persona) {  
        titular = persona;  
    }  
  
    ...  
    public void setTitular(Persona persona){  
        titular = persona;  
    }  
}
```

Error en tiempo de compilación





# Atributos de clase

---

- Representa una propiedad cuyo valor es compartido por todos los objetos de una misma clase
- Ejemplo:
  - Añadimos a las cuentas un atributo para el código de cuenta.
  - Es necesario una variable que almacene el último código de cuenta asignado.
- En un lenguaje imperativo se declararían una variable global.
- Java es un lenguaje OO puro que no permite declaraciones fuera del ámbito de una clase.

# Atributos de clase

```
public class Cuenta {
    private static int ultimoCodigo = 0;

    private int codigo;
    private double saldo;
    private final Persona titular;
    private double [] ultimasOperaciones;

    public Cuenta(Persona nombre, double saldoInicial) {
        codigo = ++ultimoCodigo;
        titular = nombre;
        saldo = saldoInicial;
        ultimasOperaciones = new double[20];
    }
}
```

A los atributos de clase se tiene acceso desde cualquier método de la clase




# Constantes

---

- En Java **no hay una declaración específica para las constantes.**
- Se consigue el mismo resultado definiendo un **atributo de clase y final.**
- Las constantes no pueden ser modificadas.
  - ➔ No tiene sentido definir métodos de acceso y modificación.
- El nivel de acceso es controlado por su visibilidad.

# Constantes

```
public class Cuenta {  
    private static final int MAX_OPERACIONES = 20;  
    private static final double SALDO_MINIMO = 100;  
  
    private static int ultimoCodigo = 0;  
  
    private int codigo;  
    private double saldo;  
    private final Persona titular;  
    private double[] ultimasOperaciones;  
  
    public Cuenta(Persona persona) {  
        codigo = ++ultimoCodigo;  
        titular = persona;  
        saldo = SALDO_MINIMO;  
        ultimasOperaciones = new double[MAX_OPERACIONES];  
    }  
    }  
}
```





# Métodos de clase

---

- ¿Cómo definimos operaciones que manejan atributos de clase?
- Un método se define de clase anteponiendo el identificador **static** a su declaración
- En el cuerpo del método de clase **sólo se puede acceder a los atributos de clase**
- Para la **aplicación** de un método de clase no se hace uso de ningún objeto receptor, sino del **nombre de la clase**



# Métodos de clase

```
public class Cuenta {  
    private static int ultimoCodigo = 0;  
    ...  
    public static int getNumeroCuentas() {  
        return ultimoCodigo;  
    }  
}
```

```
Cuenta.getNumeroCuentas();
```



# Destrucción de objetos

---

- En Java los objetos no se destruyen explícitamente
- Recolección automática de memoria de los objetos no referenciados (**Garbage Collector**)
- Existe un método **finalize()**
  - Este método se invocará justo antes de la recolección de basura
  - Interesa para liberar recursos (ej. conexión bases de datos).
- En C++ todos los objetos se destruyen (en un programa sin errores), mientras que en Java no siempre se "recolectan".



# 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



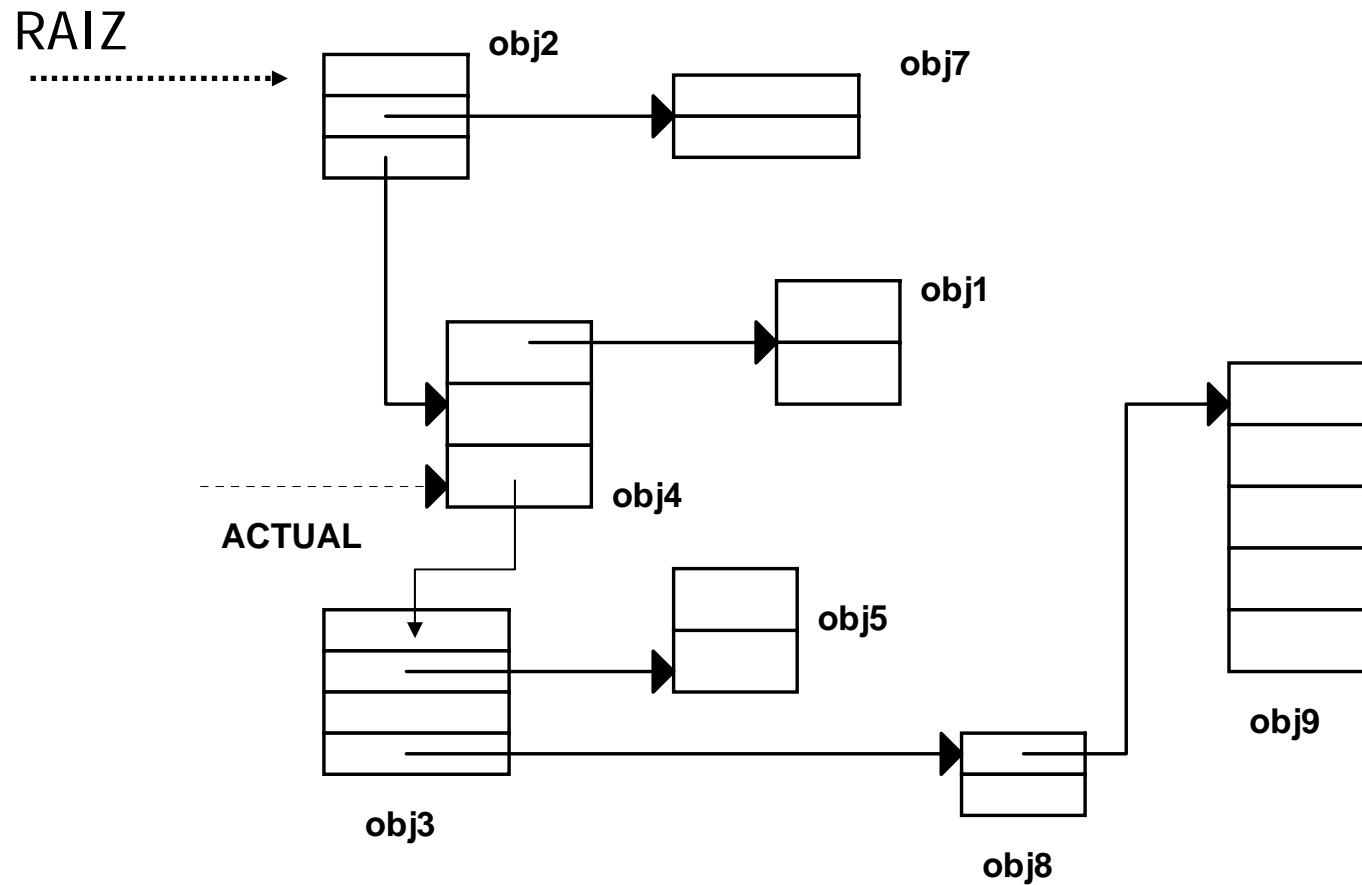


# Modelo de ejecución OO

---

- ¿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.
- Un mensaje siempre formará parte del cuerpo de una rutina de una clase

# Modelo de ejecución OO





# 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` es un método de clase que recibe como parámetro un array de cadenas de texto que son los *parámetros del programa*



# El método `main`

---

- Definición del método `main`

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

- Parámetros del programa:

```
c:\ java Eco estamos aquí --> SALIDA: estamos aquí
```



# Genericidad

---

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

~~PilaEnteros~~

~~PilaLibros~~ ⇒

**Pila de T?**

~~PilaFiguras~~

...

- 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 independiente de los datos que contiene: TIPO BASE ES UN PARÁMETRO.
  - `class ARRAY <T>`
  - `class PILA <T>`
  - `class LISTA <T>, ...`



# Clase genérica

---

```
import java.util.ArrayList;

public class Pila<T>{
    private ArrayList<T> contenido;
    ...
    public boolean isEmpty(){...}
    public void push (T item){ ... }
    public T pop() {...}
    public T tope(){...}
}
```



# Instanciación de tipo genérico

---

- Se tiene que instanciar el parámetro tanto en la declaración como en la creación:
  - `Pila<Cuenta> pilaCuentas = new Pila<Cuenta>();`
- El parámetro genérico 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;
    ....
}
```





# Autoboxing

---

- No se puede instanciar una clase genérica con un tipo primitivo.
- Existe una **clase envoltorio** para cada tipo primitivo
  - Integer, Float, Double, Character, Boolean, etc.
- El compilador transforma automáticamente tipos primitivos en objetos de las clases envoltorio y viceversa (autoboxing)

```
Pila<Integer> pilaEnteros = new Pila<Integer>;  
pilaEnteros.push(7);  
int tope = pilaEnteros.tope();
```



# 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 una entidad genérica debe ser aplicable a cualquier tipo.
- Posibles operaciones:
  - Asignación entre entidades genéricas ( $x=y$ )
  - Identidad ( $x==y$  o  $x!=y$ )
  - $a.f(\dots, x, \dots)$  (el parámetro es de tipo T)
  - Operaciones aplicables sobre cualquier objeto:
    - `x.clone()` o `x.equals(y)`
- **¡¡No se permite la creación!!**
  - `T at = new T();` No compila!!
- ¿Es posible ampliar el conjunto de operaciones?
  - Si → Genericidad restringida (Tema 3)



# Principios de diseño de clases

---

- Favorecer la **legibilidad** del código:
  - Asignar nombres significativos para los identificadores de atributos, variables y métodos
  - Inicializar los atributos
- **Experto en Información:**
  - Asignar una responsabilidad al experto en información, la clase que tiene la información necesaria para llevar a cabo la responsabilidad.
- Favorecer la **extensibilidad**:
  - Utilizar constantes simbólicas significativas
  - Seguir los **principios de diseño modular** →



# Principios de diseño de clases

## ■ Ocultación de la Información

- Los **atributos** de una clase deben ser privados y ofrecer los métodos de consulta necesarios dependiendo de su nivel de acceso (set y/o get)
- Diferenciar entre **métodos** públicos (interfaz de la clase) y métodos privados (métodos auxiliares)
- Para acceder a una **constante**, ésta debe ser pública en lugar de ofrecer un método de consulta.

## ■ Alta Cohesión

- Fragmentar clases que tengan demasiadas responsabilidades

## ■ Bajo Acoplamiento

- **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**



# Aliasing y el Principio de Ocultación de la Información

---

- Hay que prestar atención a los métodos de acceso, ya que si un atributo es una referencia, **al devolver la referencia se compromete la integridad del objeto.**
- 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.
- Se debe valorar el contexto de la clase para decidir si devolver la referencia o una copia.