

3.2 Clases y objetos en Java

Clases y tipos de datos, objetos y variables

Crear objetos: memoria dinámica y *garbage collection*

Componentes de la clase

- atributos de instancia: visibilidad y acceso

- métodos de instancia: invocación sobre objeto

- variable especial **this**

- atributos de clase

- métodos de clase

- paso de parámetros y sobrecarga

Constructores

TAD clásicos: métodos *getters* & *setters*

Clases y tipos de datos genéricos (paramétricos)

Tipos de datos enumerados

Clase: estructura de datos + operaciones

Tipo de datos

```
class CuentaBancaria {  
    long numero;  
    String titular;  
    long saldo = 0;  
  
    void ingresar(long cantidad) {  
        saldo += cantidad;  
    }  
    void retirar(long cantidad) {  
        if (cantidad > saldo)  
            System.out.println("Saldo insuficiente");  
        else saldo = saldo - cantidad;  
    }  
} // fin declaración de clase CuentaBancaria
```

Atributos
(Variables)
de
instancia

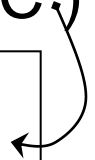
Métodos

Atributos de instancia

- Son los componentes de la estructura de datos (similar a C.)

```
class CuentaBancaria {  
    long numero;  
    String titular;  
    long saldo;  
}
```

```
typedef struct {  
    long numero;  
    char *titular;  
    long saldo;  
} CUENTA_BANCARIA;
```



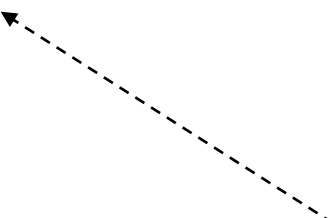
- Pueden almacenar objetos de la misma u otras clases (tb. en C)

```
class CuentaBancaria {  
    long numero = -1;  
    Cliente titular;  
    long saldo;  
}
```

```
class Cliente {  
    String nombre;  
    long dni;  
}
```

- Por defecto, inicializadas a 0 ó **null**

Las variables se pueden inicializar directamente en la clase (con una expresión que no genere excepciones)



Creación de objetos

- Una clase define un tipo de dato utilizable para declarar variables

```
CuentaBancaria cuenta1, cuenta2;
```

y que incluye las operaciones para manipular esas variables.

- `cuenta1` y `cuenta2` son **objetos** de clase `CuentaBancaria`

- Mejor: son variables de tipo **referencia a objetos** de la clase

- Los objetos **se crean** con el operador **new**

```
CuentaBancaria cuenta1 = new CuentaBancaria();
```

```
CuentaBancaria cuenta2;           // no está creado
```

- Crear un objeto reserva espacio en memoria para sus variables de instancia (y más) y devuelve una referencia al objeto

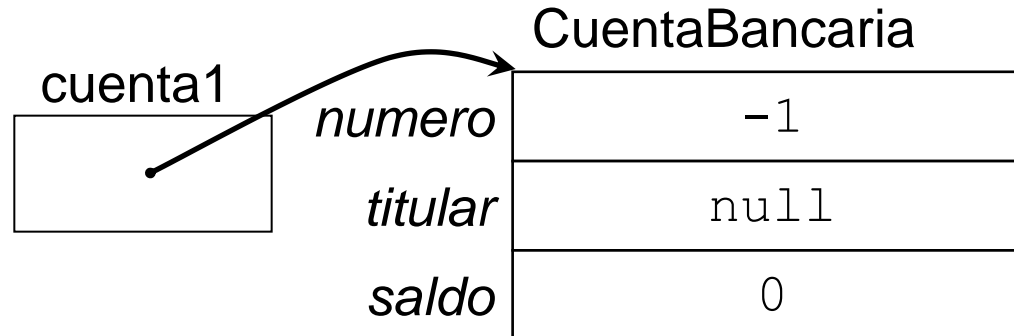
Tipos de datos vs. Clases de objetos

- El tipo de datos de una variable (o constante) se declara:
`saldo` y `cantidad` se han declarado de tipo `long`
`cuenta1` se ha declarado de tipo `CuentaBancaria`
- El tipo de datos de una expresión lo puede inferir el compilador
la expresión `saldo - cantidad` es de tipo `long`
- El tipo de datos limita los posibles valores que la variable o expresión puede tomar durante la ejecución
- En Java, si una variable es de un tipo referencia (no primitivo), entonces contendrá una referencia a un objeto, cuya clase debe ser *compatible* con el tipo declarado para esa variable
- Volveremos sobre esa compatibilidad al hablar de subclases

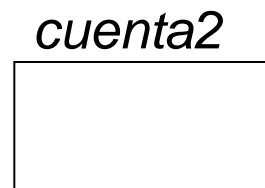
Creación de objetos

Los objetos siempre utilizan memoria dinámica

```
CuentaBancaria cuenta1 = new CuentaBancaria();
```



```
CuentaBancaria cuenta2;           // no está creado
```



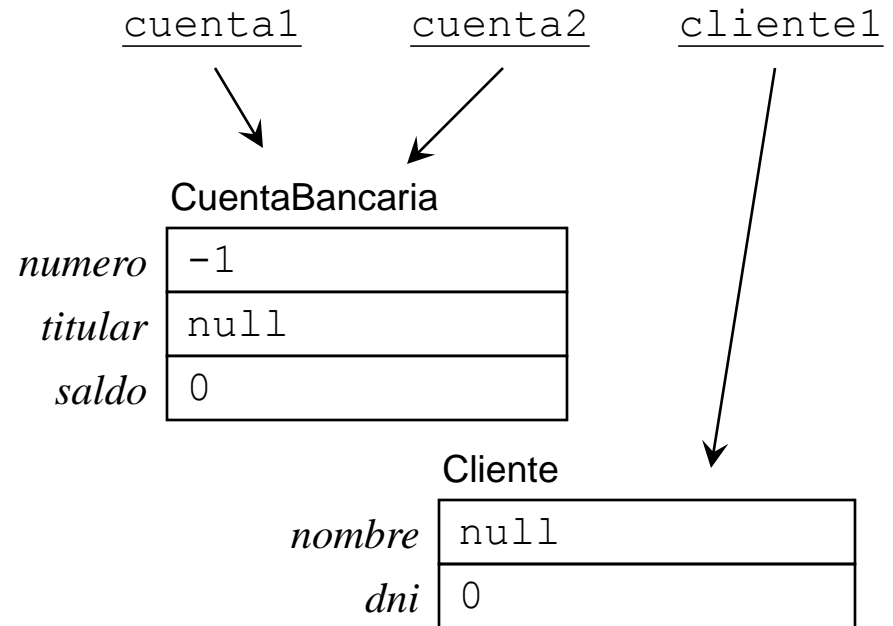
Creación de objetos y asignación

```
CuentaBancaria cuenta1, cuenta2;
```

```
cuenta1 = new CuentaBancaria();
```

```
cuenta2 = cuenta1;
```

```
Cliente cliente1 = new Cliente();
```



Acceso directo a variables de instancia (**public**)

```
CuentaBancaria cuenta1, cuenta2;
```

```
cuenta1 = new CuentaBancaria();
```

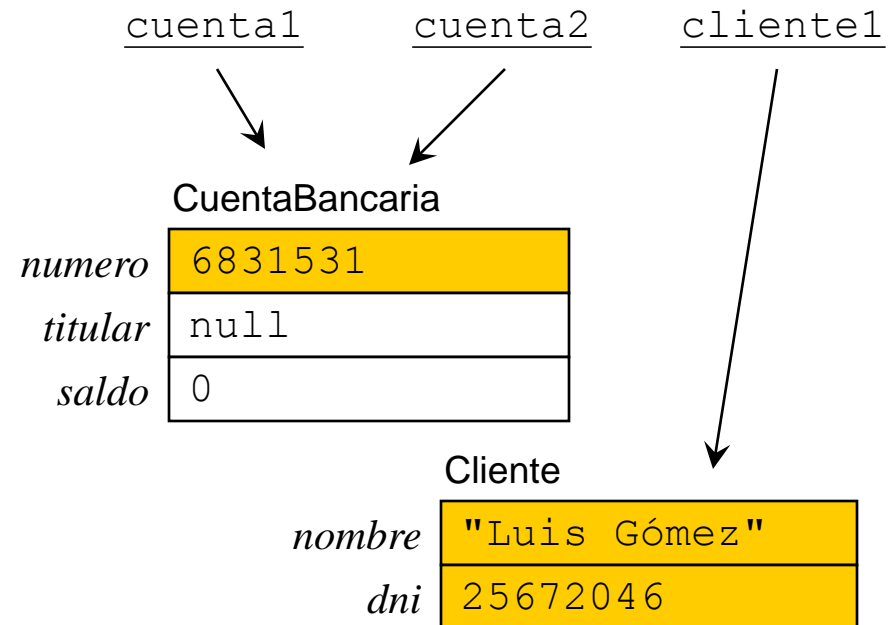
```
cuenta2 = cuenta1;
```

```
Cliente cliente1 = new Cliente();
```

```
cliente1.nombre = "Luis Gómez";
```

```
cliente1.dni = 25672046;
```

```
cuenta1.numero = 6831531;
```



Acceso directo a variables de instancia

```
CuentaBancaria cuenta1, cuenta2;
```

```
cuenta1 = new CuentaBancaria();
```

```
cuenta2 = cuenta1;
```

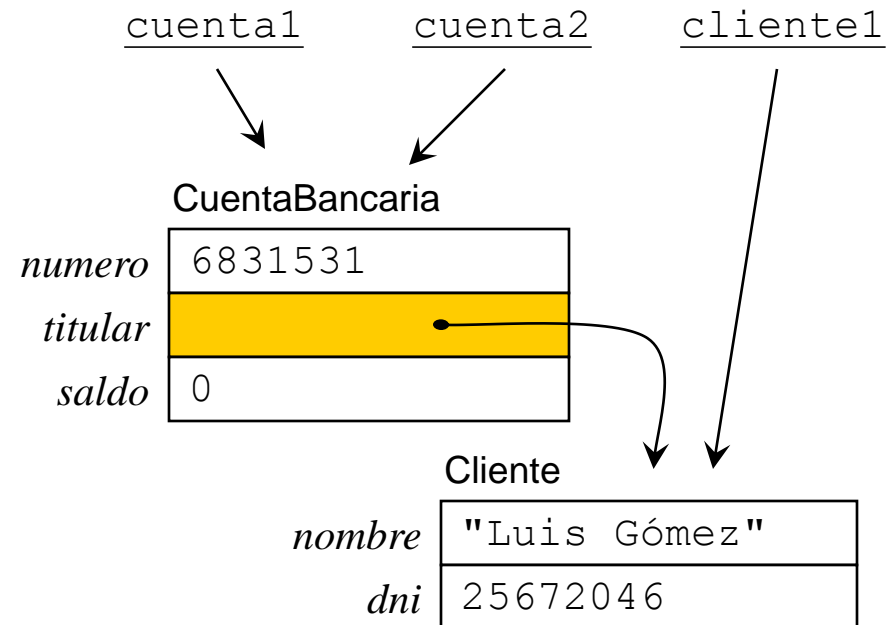
```
Cliente cliente1 = new Cliente();
```

```
cliente1.nombre = "Luis Gómez";
```

```
cliente1.dni = 25672046;
```

```
cuenta1.numero = 6831531;
```

```
cuenta1.titular = cliente1;
```



Variables de instancia: enlazando objetos

```
CuentaBancaria cuenta1, cuenta2;
```

```
cuenta1 = new CuentaBancaria();
```

```
cuenta2 = cuenta1;
```

```
Cliente cliente1 = new Cliente();
```

```
cliente1.nombre = "Luis Gomez";
```

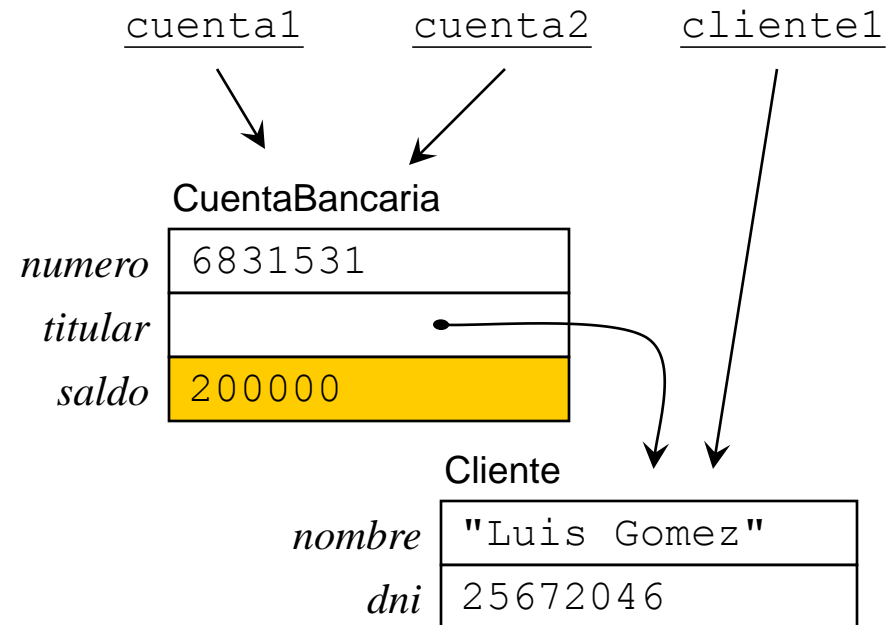
```
cliente1.dni = 25672046;
```

```
cuenta1.numero = 6831531;
```

```
cuenta1.titular = cliente1;
```

```
cuenta1.saldo = 100000;
```

```
cuenta2.saldo = 200000;
```



Acceso a través de varios objetos

```
CuentaBancaria cuenta1, cuenta2;
```

```
cuenta1 = new CuentaBancaria();
```

```
cuenta2 = cuenta1;
```

```
Cliente cliente1 = new Cliente();
```

```
cliente1.nombre = "Luis Gomez";
```

```
cliente1.dni = 25672046;
```

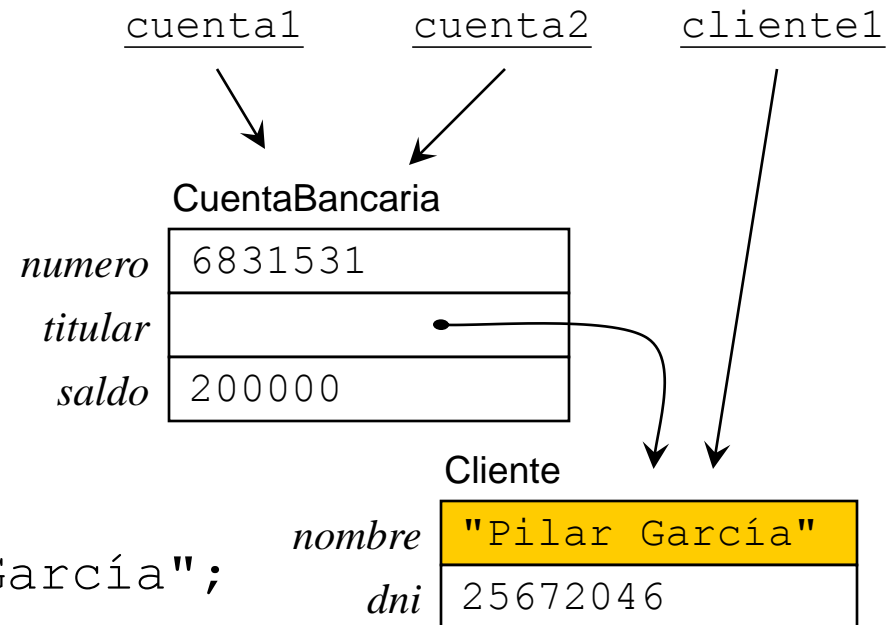
```
cuenta1.numero = 6831531;
```

```
cuenta1.titular = cliente1;
```

```
cuenta1.saldo = 100000;
```

```
cuenta2.saldo = 200000;
```

```
cuenta2.titular.nombre = "Pilar García";
```



Reasignación de referencias a objetos

```
CuentaBancaria cuenta1, cuenta2;
```

```
cuenta1 = new CuentaBancaria();
```

```
cuenta2 = cuenta1;
```

```
Cliente cliente1 = new Cliente();
```

```
cliente1.nombre = "Luis Gomez";
```

```
cliente1.dni = 25672046;
```

```
cuenta1.numero = 6831531;
```

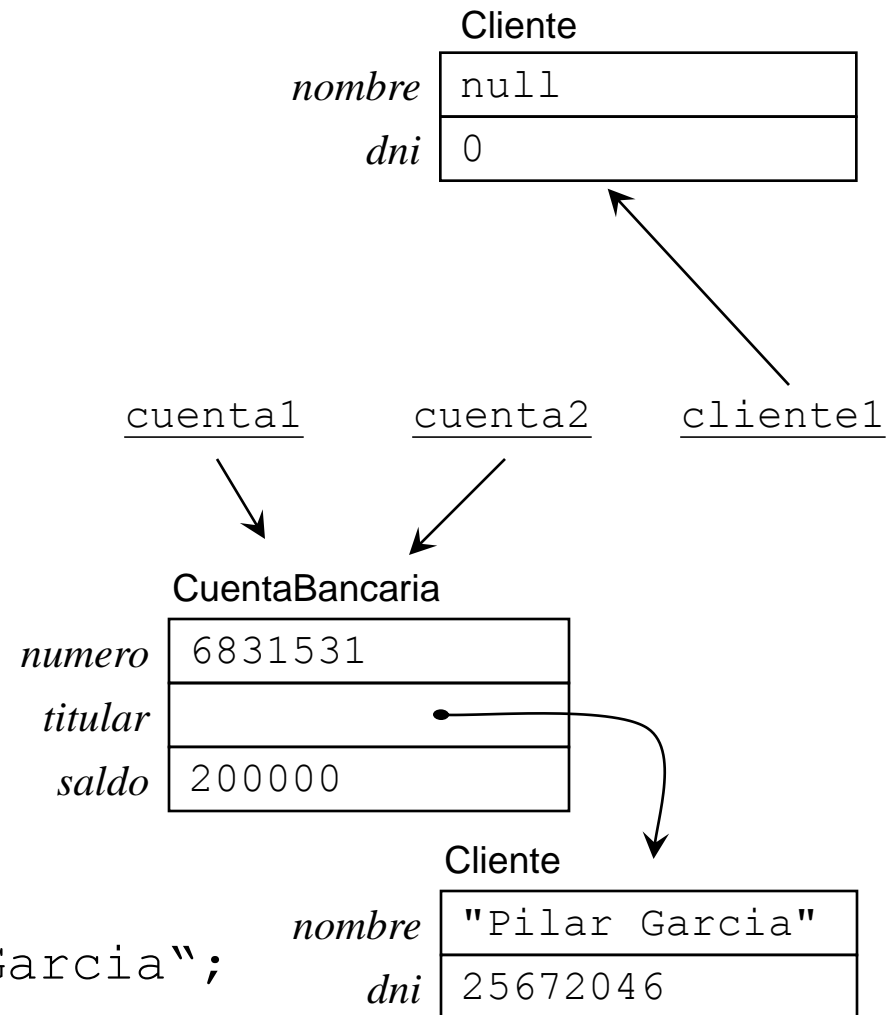
```
cuenta1.titular = cliente1;
```

```
cuenta1.saldo = 100000;
```

```
cuenta2.saldo = 200000;
```

```
cuenta2.titular.nombre = "Pilar Garcia";
```

```
cliente1 = new Cliente();
```



Gargabe collection (sin liberación explícita)

```
CuentaBancaria cuenta1, cuenta2;
```

```
cuenta1 = new CuentaBancaria();
```

```
cuenta2 = cuenta1;
```

```
Cliente cliente1 = new Cliente();
```

```
cliente1.nombre = "Luis Gomez";
```

```
cliente1.dni = 25672046;
```

```
cuenta1.numero = 6831531;
```

```
cuenta1.titular = cliente1;
```

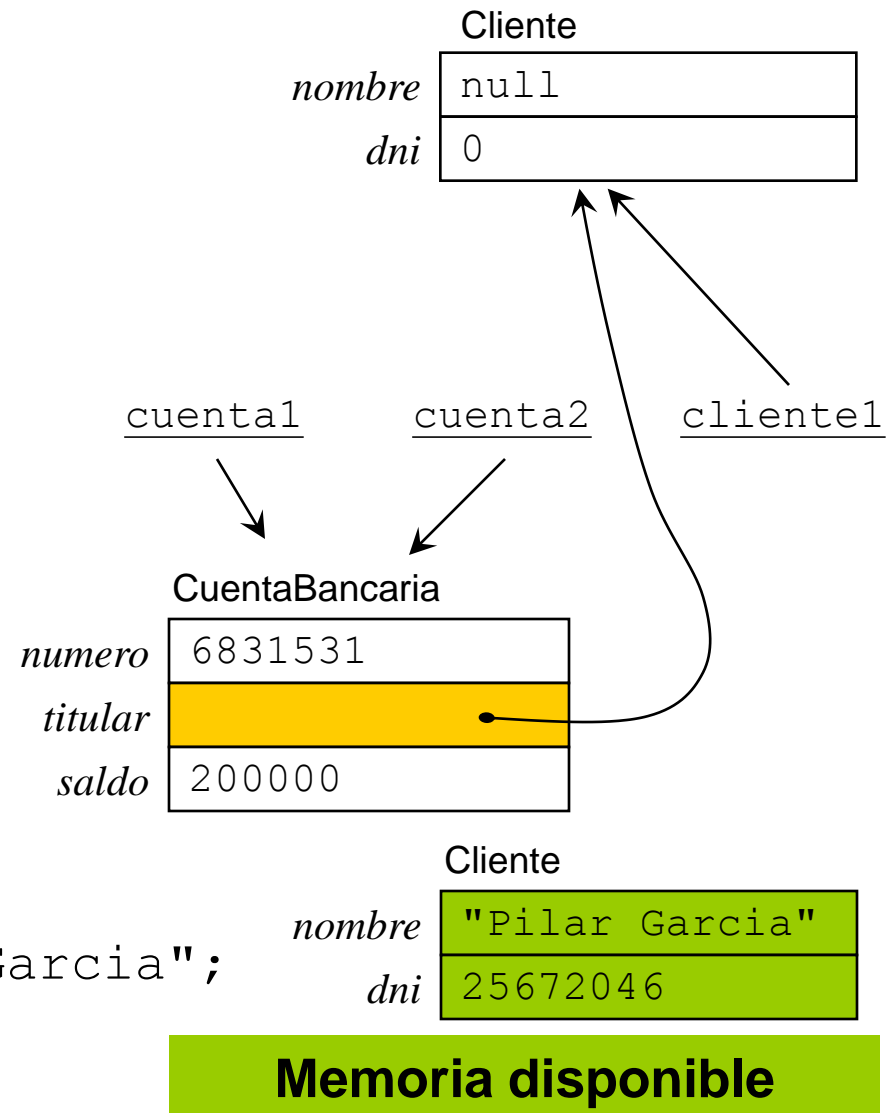
```
cuenta1.saldo = 100000;
```

```
cuenta2.saldo = 200000;
```

```
cuenta2.titular.nombre = "Pilar Garcia";
```

```
cliente1 = new Cliente();
```

```
cuenta1.titular = cliente1;
```



Métodos

- Son procedimientos o funciones definidas en una clase
- Y son parte de la estructura (tipo) de datos que define la clase (igual que C permite almacenar punteros a función en `struct`)
- Los métodos pueden referenciar directamente a los atributos de esa clase, y a otros métodos de la misma clase
- Hay **métodos de instancia** y **métodos de clase**
- Los métodos de instancia están asociados con cada objeto
- Un método de instancia se invoca sobre un objeto de la clase que define dicho método

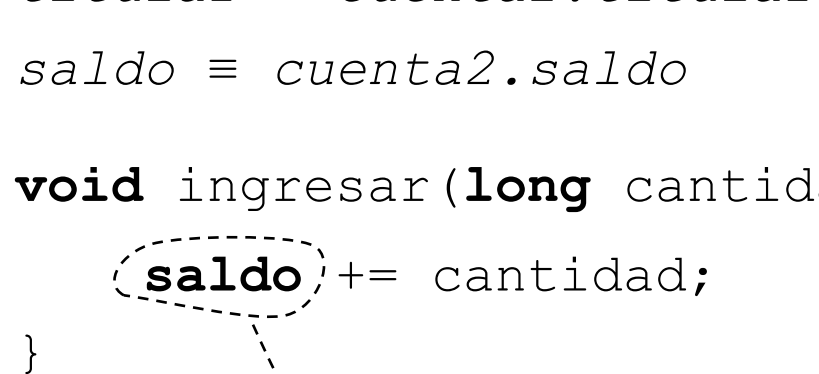
`cuenta2 . ingresar (1000) ;`

- `cuenta2` es como un parámetro implícito

Invocación de métodos de instancia

- Los métodos de instancia se invocan sobre un objeto de la clase a la que pertenecen
- Al ejecutar un método invocado sobre un objeto, las referencias a los atributos de *su* clase desde dentro del método se refieren a los atributos correspondiente del propio objeto sobre el que se invoca el método

`cuenta2.ingresar(1000);`



```
numero ≡ cuenta2.numero  
titular ≡ cuenta2.titular  
saldo ≡ cuenta2.saldo  
  
void ingresar(long cantidad) {  
    saldo += cantidad;  
}
```

`cuenta2.saldo`

Llamadas a métodos desde un método

- Los métodos de instancia pueden invocar *directamente* a los métodos de la misma clase
- Al ejecutar un método sobre un objeto, las llamadas a métodos de la clase de ese objeto se ejecutan sobre ese mismo objeto, a menos que se invoquen explícitamente sobre otro objeto

```
CuentaBancaria cuenta3 = new CuentaBancaria();  
cuenta2.emitirTransferencia(cuenta3, 1000);
```

```
class CuentaBancaria { . . .  
  void emitirTransferencia(CuentaBancaria destino, long cantidad) {  
    if (cantidad <= saldo ) {  
      retirar(cantidad);  
      destino.ingresar(cantidad);  
    }  
  }  
} // emitirTransferencia
```

cuenta2.retirar (cantidad)

Los métodos de instancia se ejecutan en el contexto de un objeto

- Un método de instancia puede acceder a:
 1. Objeto de la invocación, atributos y métodos implícitamente
 2. Objeto definido en variable local o pasada como argumento
 3. Objetos almacenados en atributos (de clase o de instancia).
- En C, el objeto de la invocación sería como un argumento más
- En POO, el objeto de la invocación juega un papel principal:
el método invocado pertenece al objeto y no a la inversa
- En un método, el objeto de la invocación está accesible explícitamente con la variable predefinida **this**

Objetos accedidos desde un método

```
ClaseA obj1 = new ClaseA();      obj1.f(7, new Y());
```

```
class X { String nombre; }
```

```
class Y { int i; }
```

```
class Z { String nombre; }
```

```
class ClaseA {  
    static int w;  
    int num;  
    X obj4;
```

```
    void f(int n, Y obj3) {
```

```
        Z obj2 = new Z();
```

```
        obj4.nombre = obj2.nombre
```

```
        num = obj3.i + w + n;
```

```
    }
```

```
}
```

1. Variable del objeto de la invocación **obj1**
2. Objeto definido en variable local del método
3. Objeto pasado como argumento al método
4. Objeto almacenado en atributo de instancia
5. Atributo de clase de la clase del objeto **obj1**

*El objeto de la invocación **obj1** no se ve como los otros objetos (2, 3, 4) pero está implícito: el método **f** accede a sus variables y también a través de **this***

El objeto sobre el que se invoca el método

```
public class Punto {  
    private long x, y;    // abcisa y ordenada de cada punto  
  
    public Punto(long x, long y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

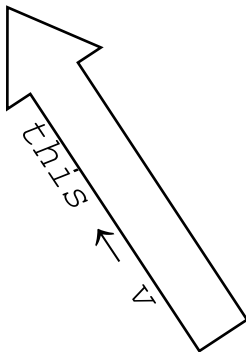
The diagram illustrates the use of the `this` keyword in a Java constructor. A dashed oval encloses the expressions `this.x` and `this.y` within the constructor body. Arrows point from `this.x` to the parameter `x` and from `this.y` to the parameter `y`, indicating that the instance variables are being assigned the values of the constructor parameters. A dashed arrow points from the `this` keyword to the `private long x, y;` declaration above, showing that `this` refers to the current object of the class.

La palabra reservada **this** es una referencia al objeto sobre el que se ha invocado el método. Se puede acceder a sus componentes, y se puede pasar como parámetro a otros métodos. Solo tiene sentido usarla dentro de métodos de instancia

Uso de variable **this** como parámetro

Un método de instancia puede utilizar el objeto sobre el que ha sido invocado dicho método para pasarlo como parámetro a otro método

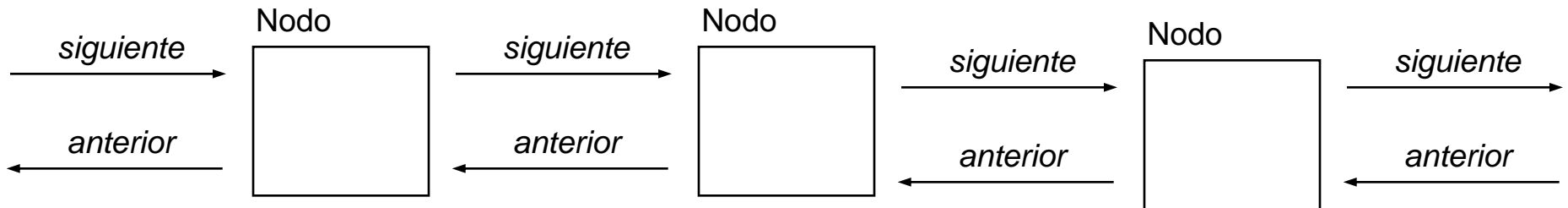
```
public class Vector3D {  
    private double x, y, z;  
    ...  
    public double productoEscalar(Vector3D u) {  
        return x * u.x + y * u.y + z * u.z;  
    }  
    public double modulo() {  
        return Math.sqrt(productoEscalar(this));  
        //Math.sqrt(this.productoEscalar(this));  
    }  
}
```



```
// Bloque main  
Vector3D v = new Vector3D(2, -2, 1);  
v.modulo();
```

La variable **this** para enlaces inversos

```
public class Nodo {  
    private Nodo anterior;  
    private Nodo siguiente;  
    ...  
    public void conectarConSgte (Nodo sgte) {  
        siguiente = sgte;  
        sgte.anterior = this;  
    }  
}
```



Atributos de clase (**static**)

```
public class Punto {  
    private long x, y;    // abcisa y ordenada de cada punto  
    private static long nmrPuntos = 0; // atributo de clase  
  
    public Punto(long x, long y) {  
        this.x = x;  
        this.y = y;  
        nmrPuntos++; // contar cada punto  
    }  
    ...  
}
```

static “no pertenece a las instancias de la clase sino a la clase”

Si no fuese **private** sería accesible también desde fuera de la clase de dos formas: **Puntos.nmrPuntos**

y **p.nmrPuntos** // suponiendo **Puntos p;**

Métodos de clase (**static**)

```
public class Math {    // clase predefinida en java.lang
```

```
    // variable de clase de valor final, o constante
```

```
public static final double PI = 3.141592653589793;
```

```
    static long round(double a) {    // método de clase
```

```
        ...
```

```
    }
```

```
    static double sin(double a) { ... }
```

```
    ...
```

```
}
```

static “no pertenece a las instancias de la clase sino a la clase”

De hecho, **Math** no tiene miembros de instancia sino sólo

miembros de clase **Math.sin(Math.PI / 2)**

Ejercicio

Escribe un programa Java que permita crear Piezas con un determinado peso. A cada pieza se le asignará automáticamente un identificador único. La clase Pieza debe contener un método estático que devuelva la pieza más pesada creada hasta el momento.

Paso de argumentos: siempre por valor (en Java)

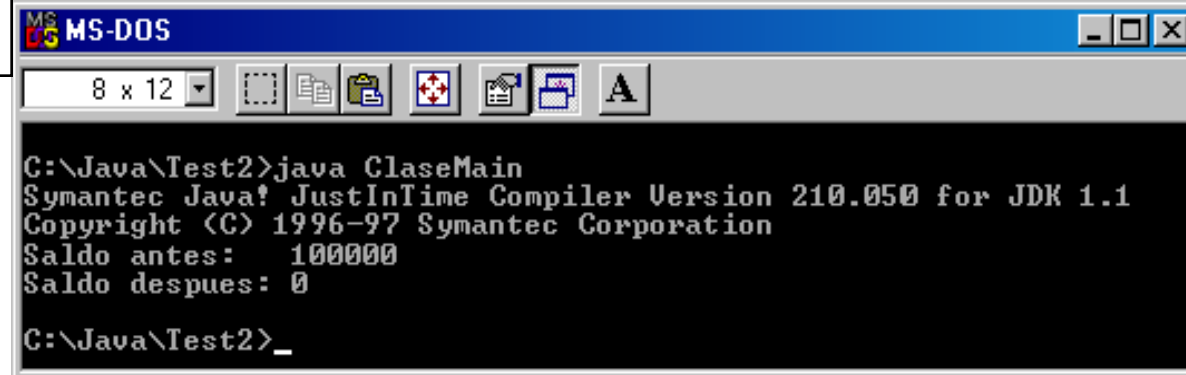
```
class ClaseMain {  
    public static void main(String[] args) {  
        int n = 5;  
        System.out.println("Antes:  " + n);  
        f(n);  
        System.out.println("Despues: " + n);  
    }  
    static void f(int i) {  
        i = i + 1;  
        System.out.println("Dentro:  " + i);  
    }  
}
```

Paso de argumentos: siempre por valor (en Java)

```
class ClaseMain {  
    public static void main(String[] args) {  
        int n = 5;  
        System.out.println("Antes:  " + n);  
        f(n);  
        System.out.println("Despues: " + n);  
    }  
    static void f(int i) {  
        i = i + 1;  
        System.out.println ("Dentro:  " + i);  
    }  
} // produce la salida:  
//Antes:  5  
//Dentro:  6  
//Despues: 5
```

Paso por valor: de referencias a objetos (I)

```
class ClaseMain {  
    public static void main(String[] args) {  
        CuentaBancaria cuenta = new CuentaBancaria();  
        cuenta.saldo = 100000;  
        System.out.println("Saldo antes: " + cuenta.saldo);  
        arruinar(cuenta);  
        System.out.println("Saldo despues:" + cuenta.saldo);  
    } // end main  
  
    static void arruinar(CuentaBancaria cta) {  
        cta.saldo = 0;  
        cta = null; // ¿?  
    }  
}  
// su salida es:  
//Saldo antes: 100000  
//Saldo despues: 0
```



```
MS-DOS  
8 x 12  
C:\Java\Test2>java ClaseMain  
Symantec Java! JustInTime Compiler Version 210.050 for JDK 1.1  
Copyright (C) 1996-97 Symantec Corporation  
Saldo antes: 100000  
Saldo despues: 0  
C:\Java\Test2>_
```

Paso por valor: de referencias a objetos (I)

```
class ClaseMain {  
    public static void main (String[] args) {  
        int a[] = {5, 4, 3, 2, 1};  
        System.out.println("Antes: " + a[3]);  
        f(a);  
        System.out.println("Despues: " + a[3]);  
    }  
    static void f(int[] x) {  
        x[3] = 0;  
        x = new int[8];  
        x[3] = 5;  
    }  
}  
//salida:  
//Antes: 2  
//Despues: 0
```

Sobrecarga de métodos: Ejemplo

```
public class Plano3D {  
    private double a, b, c, d;  
    // Plano con ecuación  $a*x + b*y + c*z + d = 0$   
    public Plano3D (double aa, double bb,  
                    double cc, double dd) {  
        a = aa; b = bb; c = cc; d = dd;  
    }  
    public boolean esParaleloA(Plano3D p) {  
        Vector3D u = new Vector3D(a, b, c);  
        Vector3D v = new Vector3D(p.a, p.b, p.c);  
        return u.esParaleloA(v);  
    }  
    public boolean esParaleloA(Recta3D r) {  
        Vector3D u = new Vector3D(a, b, c);  
        return u.esPerpendicularA(r.getVector());  
    }  
}
```

Mismo nombre,
distinta (*signature*) firma



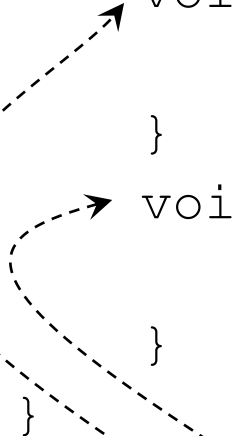
Llamada a métodos sobrecargados: Ejemplo

```
public class Recta3D {  
    private Punto3D punto;  
    private Vector3D vector;  
    public Recta3D (Punto3D p, Vector3D v) {  
        punto = p; vector = v;  
    }  
    public Vector3D getVector() { return vector; }  
}
```

```
// Bloque main  
Plano3D p1 = new Plano3D (2, 4, 3, 1);  
Plano3D p2 = new Plano3D (1, 0, -2, 1);  
Recta3D r = new Recta3D (new Punto3D (1, 0, 1),  
                        new Vector3D (1, 1, -1));  
  
p1.esParaleloA(p2);  
p1.esParaleloA(r);
```

Sobrecarga de métodos: ambigüedad

```
class A {  
    void f (int n) {  
        System.out.println ("Tipo int");  
    }  
    void f (float x) {  
        System.out.println ("Tipo float");  
    }  
}
```



*Se ejecuta la definición
compatible más específica*

```
// Bloque main  
A a = new A();  
byte b = 3;  
long l = 3;  
double d = 3;  
a.f(l);  
a.f(b);  
a.f(d); // ERROR: necesita cast  
         // explícito
```

Constructores

- No son exactamente métodos (aunque se declaran casi igual)
- No se invocan sobre un objeto, sino con **new** `UnaClase(...)`
- No están accesibles directamente desde otros métodos, pero sí están accesibles desde otros constructores.
- Son necesarios para construir objetos:
asignar memoria e inicializar componentes del objeto
- Se declaran como un método, pero sin tipo de dato de retorno, a veces con parámetros, y siempre con el mismo nombre de la clase. Con frecuencia son **public** pero pueden no serlo.
- Puede haber más de un constructor para la misma clase, pero siempre con distintos parámetros (en número o tipo).

Constructores *públicos*: ejemplos

```
public class Cliente {  
    private String nombre;  
    private long dni;
```

```
    public Cliente(String str, long num) {  
        nombre = str; dni = num;  
    }
```

```
    // ... metodos
```

```
}
```

```
class CuentaBancaria {  
    private long numero;  
    private Cliente titular;  
    private long saldo;
```

```
    public CuentaBancaria(long num, Cliente tit) {  
        numero = num; titular = tit; saldo = 0;  
    }
```

```
    // ... métodos
```

```
}
```

Constructores *públicos*: ejemplos

```
class CuentaBancaria {  
    private long numero;  
    private Cliente titular;  
    private long saldo;
```

```
    public CuentaBancaria(long num, Cliente tit) {  
        numero = num; titular = tit; saldo = 0;  
    }
```

```
    public CuentaBancaria(long num, Cliente tit, long s) {  
        numero = num; titular = tit; saldo = s;  
    }
```

```
    // ... métodos  
}
```

Constructores *públicos*: ejemplos

```
class CuentaBancaria {  
    private long numero;  
    private Cliente titular;  
    private long saldo;
```

```
    public CuentaBancaria(long num, Cliente tit) {  
        this(num, tit, 0);  
    }
```

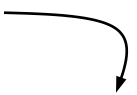
```
    public CuentaBancaria(long num, Cliente tit, long s) {  
        numero = num; titular = tit; saldo = s;  
    }
```

```
    // ... métodos  
}
```

Creación de objetos con constructores

Los constructores se ejecutan al crear los objetos con **new**

```
Cliente cliente1 = new Cliente("Luis Gomez", 272046);
```



| Cliente | |
|---------------|--------------|
| <i>nombre</i> | "Luis Gomez" |
| <i>dni</i> | 272046 |

```
CuentaBancaria cuenta1 =  
    new CuentaBancaria(683531, cliente1);
```

```
CuentaBancaria cuenta2 =  
    new CuentaBancaria(835284,  
                        new Cliente("Pilar Garcia", 151442),  
                        2000);
```

Constructor (sin parámetros) por defecto

- Si no se definen constructores, java proporciona uno por defecto

```
class ClaseX {  
    // se define implícitamente ClaseX() { }  
    // sirve para crear objetos ClaseX x = new ClaseX();  
}
```

- Si se define un constructor, el constructor por defecto no existe

```
public class Cliente {  
    ...  
    public Cliente (String str, long num) { ... }  
}
```

```
// metodo main
```

```
Cliente cliente1 = new Cliente();
```

```
// Error: No constructor matching Cliente() found in Cliente
```

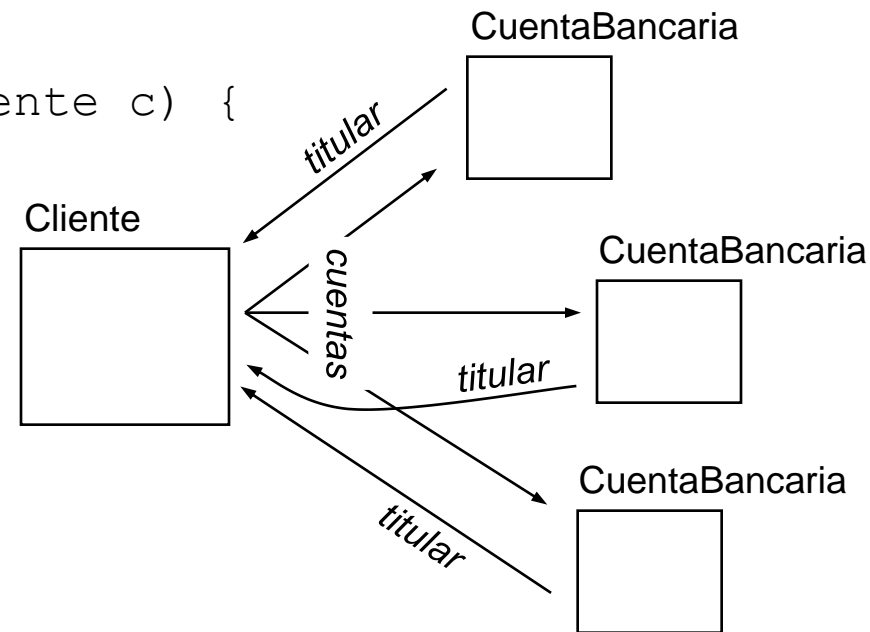
La variable **this** en constructores

```
public class CuentaBancaria {  
    private long numero;  
    private Cliente titular;  
    private long saldo = 0;  
    public CuentaBancaria(long num, Cliente c) {  
        numero = num; titular = c;  
        c.nuevaCuenta(this);  
    }  
}
```

Similar al uso para enlaces inversos

```
public class Cliente {  
    // ... nombre, dni, ... constructor ...
```

```
    public static final int MAX_CTAS = 20;  
    private CuentaBancaria cuentas[] = new CuentaBancaria[MAX_CTAS];  
    int ncuentas = 0;  
    void nuevaCuenta(CuentaBancaria cuenta) {  
        if (ncuentas < MAX_CTAS) cuentas[ncuentas++] = cuenta;  
    }  
}
```



Sobrecarga de constructores: firmas distintas

```
public class Punto3D {  
    private double x, y, z;  
    public Punto3D(double xx, double yy, double zz) {  
        x = xx; y = yy; z = zz;  
    }  
}
```

Constructores con firma (*signature*)
distinta

```
public class Vector3D {  
    private double x, y, z;  
  
    // Vector con origen en (0,0,0)  
    // y dadas las coordenadas de su vértice  
    public Vector3D(double xx, double yy, double zz) {  
        x = xx; y = yy; z = zz;  
    }
```

```
    // Vector dadas las coordenadas de su origen y vértice  
    public Vector3D(Punto3D p, Punto3D q) {  
        x = q.x - p.x; y = q.y - p.y; z = q.z - p.z;  
    }
```

...

```
}
```

Constructores privados:

Patrón Singleton

```
public class PrintQueue {
    private static PrintQueue INSTANCE;
    ...
    // El constructor privado impide la instanciación desde otras clases...
    private PrintQueue() { }

    public static PrintQueue getInstance() {
        if (INSTANCE==null) INSTANCE = new PrintQueue();
        return INSTANCE;
    }

    public void addJob (Job j) {
        ...
    }
}



---


public class Application {
    ...
    public static void main(String [] args) {
        PrintQueue queue = PrintQueue.getInstance(); // Solo puede existir
        ...                                           // un objeto PrintQueue
    }
}
```


Tipos abstractos de datos (TAD) clásicos

- Estructuras de datos inaccesibles,
excepto por las operaciones explícitamente definidas para ello.
- Elementos y características básicas de TAD en Java:
 - Clase pública
 - Atributos de instancia (y de clase) privadas
 - Constructor(es) público(s)
 - Métodos de acceso a componentes (*getters*)
 - Métodos de establecer valor de componentes (*setters*)
 - Otros métodos que completan la funcionalidad del TAD

Tipos abstractos de datos (TAD) clásicos

```
public class Marquesina {  
    private String texto;  
    private int ancho;  
    private double velocidad;  
    private int posicionX, posicionY;  
    private boolean visible;  
  
    public Marquesina(String t, int a, int x, int y) {  
        texto = t; ancho = a; posicion X = x; posicionY = y;  
        visible = false; velocidad = 1.0;  
    }  
}
```

Diagram illustrating the structure of the `Marquesina` class:

- Estructura privada**: Points to the private fields (`texto`, `ancho`, `velocidad`, `posicionX`, `posicionY`, `visible`).
- Constructor público**: Points to the public constructor (`public Marquesina(String t, int a, int x, int y)`).

```
} // end class Marquesina
```

Tipos abstractos de datos (TAD) clásicos

```
public class Marquesina {  
    private String texto;  
    private int ancho;  
    private double velocidad;  
    private int posicionX, posicionY;  
    private boolean visible;  
  
    // constructor(es)  
  
    // metodos getters  
    public String getTexto() { return texto; }  
    public int getAncho() { return ancho; }  
    public int getX() { return posicionX; }  
    public int getY() { return posicionY; }  
    public boolean getVisible() { return visible; }  
  
} // end class Marquesina
```

Métodos *getter*: devuelven el valor de los componentes del objeto útiles para programar manejando estos objetos, pero sin dependencia directa de la estructura interna.

Tipos abstractos de datos (TAD) clásicos

```
public class Marquesina {  
    private String texto;  
    private int ancho;  
    private double velocidad;  
    private int posicionX, posicionY;  
    private boolean visible;  
  
    // constructor(es)  
    // metodos getter  
  
    // metodos setter  
    public void setTexto(String t) { texto = t; }  
    public void setAncho(int ancho) { ancho = a; }  
    public void setPosicion(int x, int y) {  
        posicionX = x; posicionY = y;  
    }  
    public void setVisible() { visible = true; }  
    public void setInvisible() { visible = false; }  
  
} // end class Marquesina
```

Métodos *setter*: permiten a los programas dar valor a los componentes del objeto de forma adecuada, pero sin conocimiento directo de la estructura interna.

Tipos abstractos de datos (TAD) clásicos

```
public class Marquesina {  
    private String texto;  
    private int ancho;  
    private double velocidad;  
    private int posicionX, posicionY;  
    private boolean visible;
```

```
    // constructor(es)  
    // metodos getter  
    // metodos setter  
    // otros metodos
```

Otros métodos: completan el abanico de operaciones aplicables a los objetos.

```
    public void acelerar() { velocidad = velocidad * 1.25; }  
    public void lentificar() { velocidad = velocidad * 0.8; }  
    public void comenzarScroll() {  
        ...  
    }  
    public void detenerScroll() {  
        ...  
    }  
} // end class Marquesina
```

Tipos de datos genéricos (clases genéricas)

- Se definen mediante clases parametrizables por un tipo base
- Al declarar variables se especializa la clase para un tipo concreto

```
ArrayList<Punto> miListaDePuntos;
```

Ventajas:

- **Maximizar la reutilización de código (facilitar mantenimiento)**
- El compilador no necesita que se hagan ciertos *castings*
- Más errores detectados en compilación, en vez de en ejecución
- Java proporciona muchas clases genéricas, sobre todo para *colecciones*: vector, pila, lista, lista de acceso directo (**ArrayList**)
- También podemos definir nuestras clases como genéricas

Tipo de datos: Pila no específica

Es complicado utilizar una pila sin saber el tipo de su contenido

```
import java.util.Stack;
public class StackExample {
    public static void main(String[] args) {
        Stack pila = new Stack(); // es una pila de objetos

        pila.push("casa");
        pila.push(new Punto(1,2));

        // así daría error de compilación: tipos incompatibles
        // Punto p = pila.pop();
        Punto p1 = (Punto) pila.pop(); // casting necesario

        // si repetimos otro pop identico: error de ejecución
        Punto p2 = (Punto) pila.pop(); // ¿Por qué da error?
    }
```

Tipo de datos genérico: Pila específica

Es mejor utilizar pilas genéricas previamente especializadas

```
import java.util.Stack;
public class StackExample {
    public static void main(String[] args) {
        Stack<String> palabras = new Stack<String>();
        Stack<Punto> puntos = new Stack<Punto>();

        palabras.push("casa");           palabras.push("bloque");
        puntos.push(new Punto(1,2));      puntos.push(new Punto(3,0));

        Punto p1 = puntos.pop(); // no hace falta casting
        String s2 = palabras.pop();

        System.out.println("Palabra: " + palabras.pop());
        System.out.println("Punto: " + puntos.pop());
    } // imprime Palabra: casa y Punto: Punto@1bc4459 ¿Qué!
```

Tipo de datos genérico especializado

Constructor genérico especializado

Clase Punto con conversión a String

El ejemplo anterior visualizaría mejor los objetos de tipo Punto si: definiésemos un método público `toString()`

```
public class Punto {  
    private int x, y;  
    // constructor  
    public Punto(int x, int y) { this.x = x;  this.y = y; }  
    // getters  
    public int getX() {return x;}  
    public int getY() {return y;}  
    // setters  
    public void setX(int x) {this.x = x;}  
    public void setY(int y) {this.y = y;}  
    // conversion a String  
    public String toString() {return "(" + x + "," + y + ")";}  
}  
// ahora el ejemplo anterior imprime  Palabra: casa  y  Punto: (1,2)
```

Todo sobre la clase **Stack** en el API

download.oracle.com/javase/1.4.2/docs/api/

Constructor Summary

Stack() // Creates an empty Stack.

Method Summary

boolean empty() // Tests if this stack is empty.

Object peek() // Looks at the object at the top of this
// stack without removing it from the stack

Object pop() // Removes the object at the top of this
// stack and returns that object as the
// value of this function.

Object push(Object item) // Pushes an item onto the top
// of this stack.

int search(Object o) // Returns the 1-based position
// where an object is on this stack.

Otra clase genérica predefinida: **ArrayList**

- Lista de acceso directo por índice de la posición
- Implementadas con arrays que crecen dinámicamente según se van añadiendo elementos
- Añadir n elementos requiere tiempo $O(n)$
- Con operaciones más eficientes que usando listas enlazadas
- Como ejemplo de uso podemos convertir un archivo de palabras (ya separadas una en cada línea) en una tabla de frecuencias de aparición de cada palabra, usando dos **ArrayList**
uno para las palabras y otro para las frecuencias, pero manteniendo las mismas posiciones relativas en cada lista

Ejemplo de uso de ArrayList

```
import java.io.*;
import java.util.*;
public class Frecuencias {
    public static void main(String[] args) throws IOException {
        BufferedReader buffer =
            new BufferedReader(
                new InputStreamReader(
                    new FileInputStream("palabras") ) );
        ArrayList<String> palabras = new ArrayList<String>();
        ArrayList<Integer> freq = new ArrayList<Integer>();
        String p;
        while ((p = buffer.readLine()) != null) {
            if (palabras.contains(p)) {
                int i = palabras.indexOf(p);
                freq.set(i, freq.get(i) + 1);
            } else { palabras.add(p); freq.add(1); }
        }
        for (int k=0; k < palabras.size(); k++)
            System.out.println(palabras.get(k) + ": " + freq.get(k));
        buffer.close();
    }
} // end class Frecuencias
```

Mejor con un HashMap

Todo sobre la clase `ArrayList` en el API

download.oracle.com/javase/1.4.2/docs/api/

Constructor Summar (solo constructores seleccionados)

`ArrayList()` // Constructs empty list, initial capacity 10

`ArrayList(int initialCapacity)`

Method Summary (solo métodos seleccionados)

`void add(int index, Object element)` // insert at index

`boolean add(Object o)` // Appends element to the end

`void clear()` // Removes all of elements from this list

`boolean contains(Object elem)`

`Object get(int index)` // Returns element at position index

`int indexOf(Object elem)` // Searches for first occurrence

`Object set(int index, Object element)`

`boolean isEmpty()`

`int lastIndexOf(Object elem)`

`Object remove(int index)`

`int size()`

Tipos de datos enumerados

- Más complejos y versátiles que en otros lenguajes (p.ej.: Pascal)
- Se definen mediante clases **enum**
- Cada valor de la enumeración es como una constante
- Cada valor del tipo enumerado se puede imprimir como **String**
- Método de clase **values()** devuelve un array con todos los valores de la enumeración
- La representación interna de cada valor puede quedar oculta, o puede controlarse mediante descripción explícita
- Veamos algunos ejemplos útiles,
pero algunos detalles de enum se entenderán mejor más adelante.

Tipos enumerados: el ejemplo más básico

Enumeración con valores internos ocultos. Los valores de la enumeración se pueden imprimir y se pueden obtener con **values()**

```
public class EnumAsig {
```

```
    enum Asignatura {                // valores de la enumeración  
        ALGEBRA, CALCULO, FISICA, PROGRAMACION, TALLER;  
    }
```

```
    public static void main(String[] args) {
```

```
        Asignatura a = Asignatura.FISICA;
```

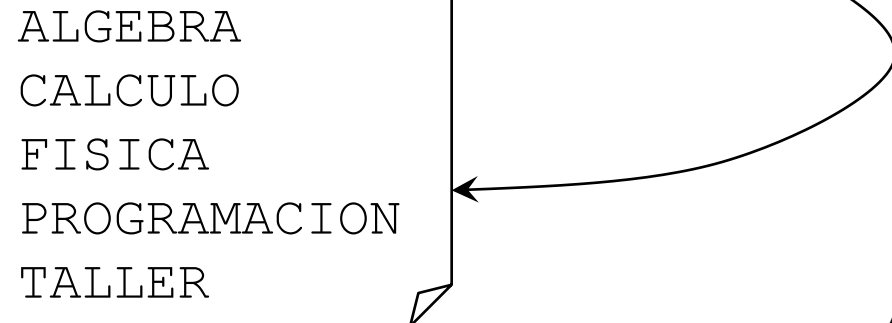
```
        System.out.println("vale: " + a); //imprime vale: FISICA
```

```
        Asignatura[] valores = Asignatura.values();
```

```
        for (Asignatura x : valores) System.out.println(x);
```

```
    }
```

```
}
```



```
ALGEBRA  
CALCULO  
FISICA  
PROGRAMACION  
TALLER
```

Tipos enumerados: fijando valor interno

Valor interno privado pero especificado mediante constructor privado

```
public class EnumDia {
```

```
    enum Dia {  
        LUNES(1), MARTES(2), MIERCOLES(3), JUEVES(4),  
        VIERNES(5), SABADO(6), DOMINGO(0);  
  
        private Dia(int d) { valor = d; } // constructor privado  
        private final int valor;        // valor interno controlado  
  
        public int valor() { return valor; }  
    }
```

```
public static void main(String[] args)
```

```
    Dia dia = Dia.VIERNES;
```

```
    ...
```

```
    Dia[] semana = new Dia[Dia.values().length];
```

```
    for (Dia d : Dia.values()) { semana[d.valor()] = d; }
```

```
    for (Dia d : Dia.values())
```

```
        System.out.println("semana["+d.valor()+"] es " + d);
```

```
    }
```

```
}
```

| | | |
|-----------|----|-----------|
| semana[1] | es | LUNES |
| semana[2] | es | MARTES |
| semana[3] | es | MIERCOLES |
| semana[4] | es | JUEVES |
| semana[5] | es | VIERNES |
| semana[6] | es | SABADO |
| semana[0] | es | DOMINGO |

Enumeraciones: con varios valores internos y con métodos adicionales

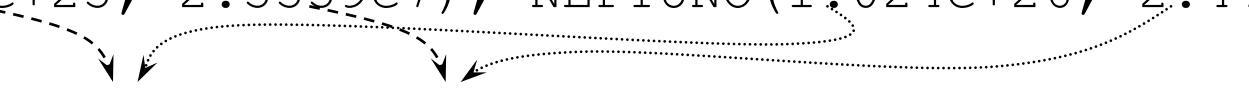
```
public enum Planeta { // en archivo Planeta.java
    MERCURIO(3.303e+23, 2.4397e6), VENUS(4.869e+24, 6.0518e6),
    TIERRA(5.976e+24, 6.37814e6), MARTE(6.421e+23, 3.3972e6),
    JUPITER(1.9e+27, 7.1492e7), SATURNO(5.688e+26, 6.0268e7),
    URANO(8.686e+25, 2.5559e7), NEPTUNO(1.024e+26, 2.4746e7);

    Planeta(double m, double r) { masa = m; radio = r; }

    private final double masa; // kg
    private final double radio; // metros

    private double masa() { return masa; }
    private double radio() { return radio; }

    // cte. gravitation universal
    public static final double G = 6.673E-11;
    // metodos adicionales
    double gravedadSup() { return G * masa / (radio*radio); }
    double pesoSup(double masa) { return masa*gravedadSup(); }
}
```



Enumeraciones como elementos de objetos

```
public class Carta {  
  
    public enum Numero { AS, DOS, TRES, CUATRO, CINCO,  
        SEIS, SIETE, SOTA, CABALLO, REY }  
  
    public enum Palo { OROS, ESPADAS, COPAS, BASTOS }  
  
    // componentes privados del objeto  
    private final Numero numero;  
    private final Palo palo;  
  
    // constructor publico del objeto  
    public Carta(Numero n, Palo p) { numero = n; palo = p; }  
  
    public Numero numero() { return numero; }  
    public Palo palo() { return palo; }  
  
    public String toString() {return numero + " de " + palo;}  
  
}
```

Ejercicio

- Escribe un programa Java para la gestión de pedidos.
- Un pedido está formado por una o más líneas de pedido.
- Una línea de pedido tiene una descripción, cantidad, coste unitario, y tipo de impuesto.
- Los tipos de impuestos pueden ser:
 - General (21%), Reducido (10%) y Superreducido (4%)
- Añade la funcionalidad necesaria para imprimir por pantalla un pedido, así como para calcular el total.