

PRÁCTICA 1. JAVA

Bases de Datos
Grado en Ingeniería Telemática

Índice

- Descripción
- Sintaxis básica
- Programación Orientada a Objetos
- Arrays y Estructuras
- Entradas y Salidas
- Serialización
- Ejercicio

Descripción

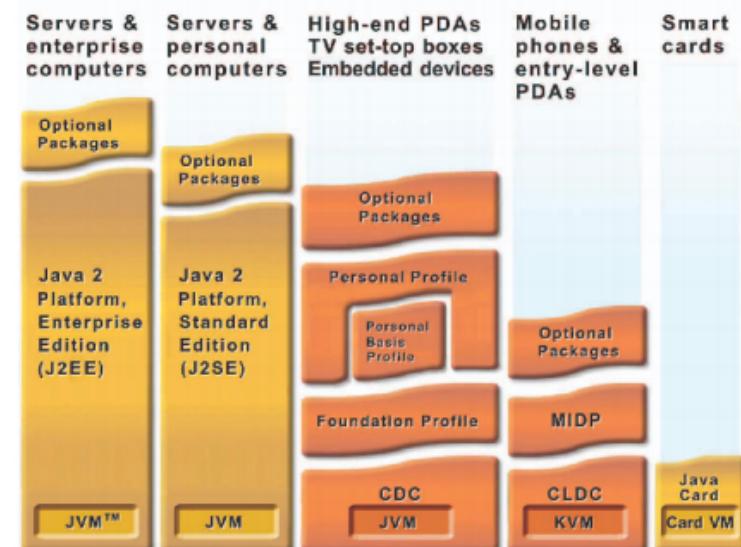
¿Por qué Java?

- Uno de los lenguajes más extendidos
- Es orientado a objetos
- Tiene una librería muy extensa
- Es independiente de la plataforma
 - Muy bueno en programación Web

Descripción

Ediciones (*Platforms*):

- Aplicaciones de propósito general (PC, servidores,...)(J2SE)
- Aplicaciones de gestión de entornos empresariales (J2EE)
- Aplicaciones para teléfonos móviles, PDAs, y otros dispositivos (J2ME)
- Aplicaciones para dispositivos con dispositivos de memoria pequeña (Java Card)



Descripción

- Algunos de los programas que incluyen estas plataformas son:
<http://www.oracle.com/technetwork/es/java/javase/downloads/index.html>
 - Java Development Kit (JDK), permite desarrollar y ejecutar aplicaciones en Java
 - Java Runtime Environment (JRE), permite únicamente la ejecución
 - Server Java Runtime Environment, para desarrollar y monitorear aplicaciones en servidores (incluye JVM)

Descripción

Classpath

- Especifica donde se encuentran las clases para ejecutar la aplicación
- Dos maneras:
 - Variable de entorno CLASSPATH
 - Argumento (-cp o -classpath)
- Por defecto, las clases del JDK (JRE)

Descripción

- **Compilación y Ejecución**

Ventas.java

```
class Producto {  
    int id;  
    string titulo;  
    float precio;  
    int cantidad;  
}  
  
class Usuario{  
}  
  
class VentaInternet {  
}
```

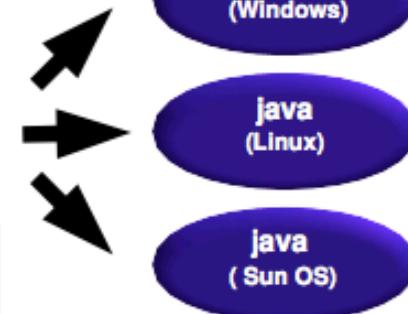
Código fuente



Compilación

Producto.class
Usuario.class
VentaInternet.class

Ficheros de clases en
bytecode o código
intermedio (no son binarios
ejecutables directamente)



Ejecución interpretada o
compilada just-in-time con
máquina virtual Java
específica



Biblioteca
de clases
Java

Descripción

Ficheros JAR

- Java Archive
- Son ficheros ZIP que contienen clases, fuentes, etc.
- Tienen un directorio META-INF donde reside el manifiesto (MANIFEST.MF)
 - Describe el contenido del JAR

Descripción

- Entornos de Desarrollo
 - JDK
 - NetBeans
 - Eclipse
 - Borland Jbuilder
 - Microsoft Visual J++

Sintaxis Básica

```
public class Basic {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int current = 1; current <= 10; current++) {  
            sum += current;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

Sintaxis Básica

- En Java, todo son clases
- Así que cada fichero define una clase
 - public class MiClase
- La clase tiene atributos y métodos
- El método main es especial
 - public static void main(String[] args) {...}
 - Es el punto de arranque de una clase

Sintaxis Básica

Tipos primitivos

- Enteros (se inicializan a 0)
 - Byte: un byte con signo ((byte)12)
 - Short: dos bytes con signo ((short)1231)
 - Int: cuatro bytes con signo (1238921)
 - Long: ocho bytes con signo (728478283L)
- Reales (se inicializan a 0.0)
 - Float: punto flotante 32 bits (1.2342F)
 - Double: punto flotante 64 bits (123.131)
- Booleanos (se inicializa a false)
 - Boolean: true / false
- Carácter (se inicializa al carácter nulo)
 - Char: 'S', 'a'
- El tamaño de datos está definido y es independiente de la plataforma

Sintaxis Básica

Tipos de datos de Referencia

- Objetos
 - Instancias de clases
- Arrays
 - Colección de elementos del mismo tipo, sean básicos o complejos
- Se inicializan a null

Sintaxis Básica

Tipo de operadores	Operadores de este tipo
Operadores posfijos	[] . (parametros) expr++ expr--
Operadores unarios	++expr --expr +expr -expr ~ !
Creación o conversión	new (tipo) expr
Multiplicación	* / %
Suma	+ -
Desplazamiento	<<
Comparación	< <= = instanceof
Igualdad	== !=
AND a nivel de bit	&
OR a nivel de bit	^
XOR a nivel de bit	
AND lógico	&&
OR lógico	
Condicional	? :
Asignación	= += -= *= /= %= &= ^= = <<= ==

Sintaxis Básica

Ejemplos

- ?: (if-then-else)
 $a == b ? c : d$ es igual a $\text{if } (a == b) \text{ then } c \text{ else } d$
- Operadores de Asignacion:
 $op1 += op2$ es igual $op1 = op1 + op2$
- [] - indexación de arrays. Ej: $A[i]$
- . (punto): acceso a métodos y variables
Ej. `MiClase.a = 0; MiClase.CalcularMaximo();`

Sintaxis Básica

Casting

- Cambiar el tipo de una variable
- Cambios de tipo automáticos
- De int a float, de float a int

```
int a;
```

```
float b;
```

```
a = (int) b; // Se pierde información
```

```
b = (float) a; // No es necesario
```

Sintaxis Básica

- if ($x < y$) smaller = x ;
- if ($x < y$){ smaller= x ; sum += x ; }
else { smaller = y ; sum += y ; }
- while ($x < y$) { $y = y - x$; }
- do { $y = y - x$; } while ($x < y$)
- for (int $i = 0$; $i < max$; $i++$) sum += i ;
- PERO: las condiciones son **booleanas** !

Sintaxis Básica

```
switch (n + 1) {  
    case 0: m = n - 1; break;  
    case 1: m = n + 1;  
    case 3: m = m * n; break;  
    default: m = -n; break;  
}
```

- El control de excepciones no lo vamos a ver en esta práctica (*try, Catch*)

Sintaxis Básica

Ámbito de las variables

```
public class Basic {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int current = 1; current <= 10; current++) {  
            sum += current;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

¿Cuál es? ¿Qué tipo tienen? Alcance

Sintaxis Básica

Ámbito de las variables

```
public class Basic {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int current = 1; current <= 10; current++) {  
            sum += current;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

¿Cuál es? ¿Qué tipo tienen? Alcance

Sintaxis Básica

Ámbito de las variables

```
public class Basic {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int current = 1; current <= 10; current++) {  
            sum += current;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

¿Cuál es? ¿Qué tipo tienen? Alcance

Sintaxis Básica

myArray =	3	6	3	1	6	3	4	1
	0	1	2	3	4	5	6	7

Lista de elementos:

- Del mismo tipo
- Tamaño fijo
- Tamaño declarado cuando se define y no pueden cambiar durante la ejecución
- Se accede a través de índices, comenzando por 0

Sintaxis Básica

int myArray[];

 Array de integer

myArray = new int[8];

 Array de 8 enteros de *myArray[0]* a *myArray[7]*

int myArray[] = new int[8];

 Combina ambas expresiones

- Acceso por índices

myArray[0] = 3;

myArray[1] = 6;

myArray[2] = 3; ...

- Inicialización en un paso:

int myArray[] = {3, 6, 3, 1, 6, 3, 4, 1};

String languages [] = {"Prolog", "Java"};

Truco para recorrer un array en un *for*:

```
for (int i=0;i<myArray.length; i++) {  
    myArray[i] = getsomevalue();  
}
```

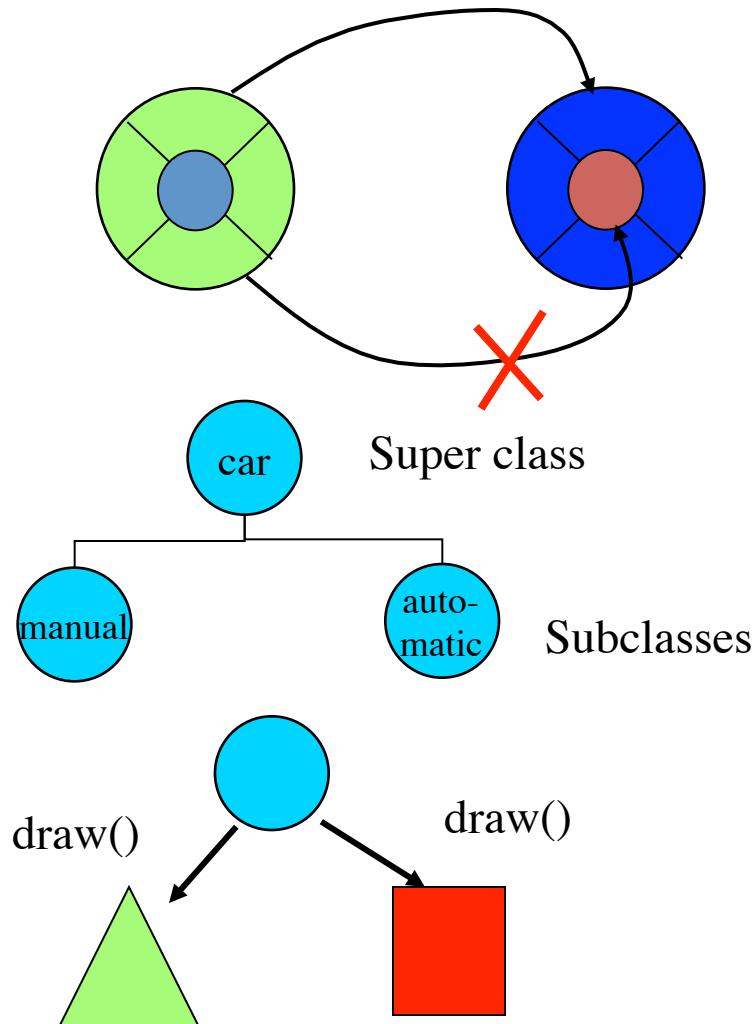
Otro

```
for (int i : myArray) {  
    myArray[i] = getsomevalue();  
}
```

Programación O.O.

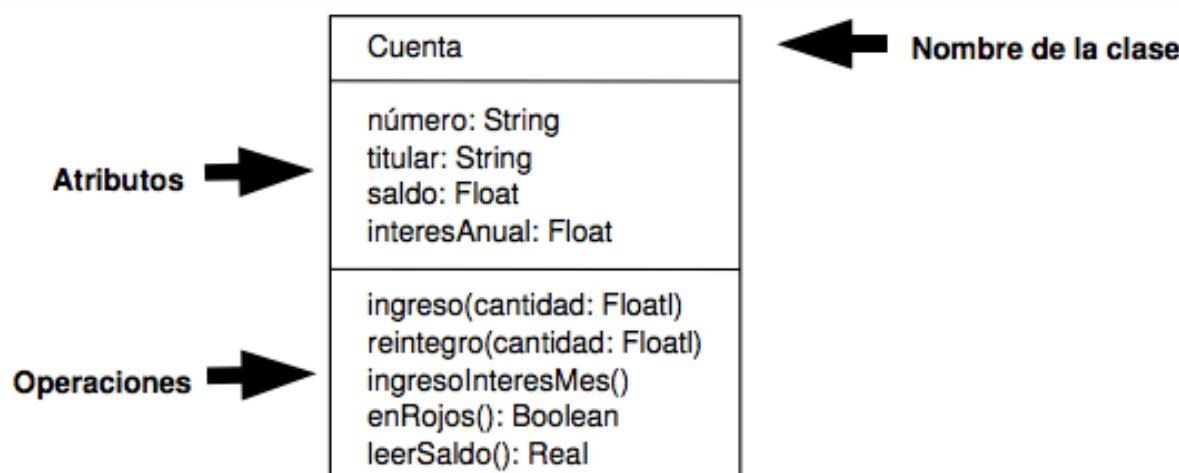
Principios de la POO

- Encapsulación
 - Los objetos tienen métodos y datos (variables de instancia)
- Herencia
 - Cada subclase hereda todas las variables de su superclase
- Polimorfismo
 - La misma interfaz para diferentes tipos de datos



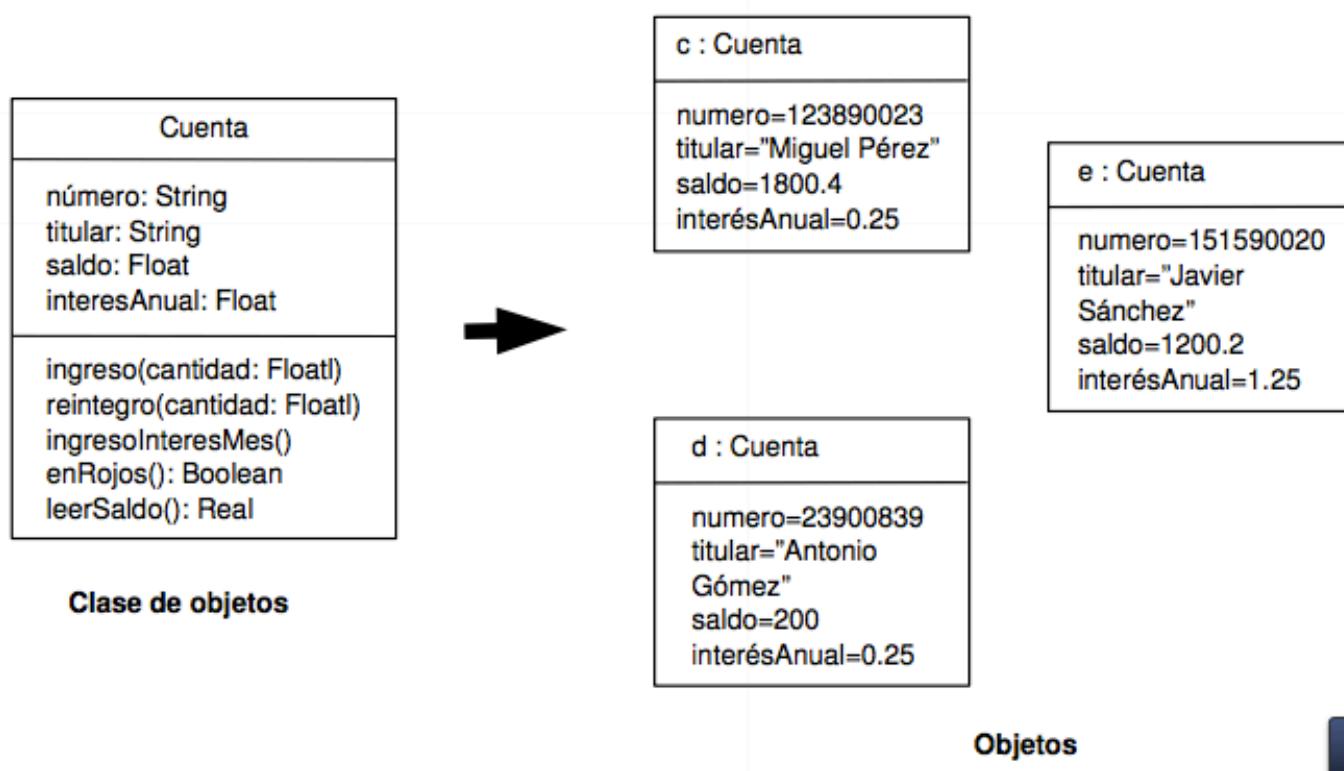
Programación O.O.

Clase sencilla



Programación O.O.

Objetos



Programación O.O.

Definición de una clase sencilla

```
class Cuenta {  
    long numero;  
    String titular;  
    float saldo;  
    float interesAnual;  
  
    void ingreso(float cantidad) {  
        saldo += cantidad;  
    }  
  
    void reintegro(float cantidad) {  
        saldo -= cantidad;  
    }  
  
    void ingresoInteresMes () {  
        saldo += interesAnual * saldo / 1200;  
    }  
  
    boolean enRojos() { return saldo < 0; }  
    float leerSaldo() { return saldo; }  
}
```

Programación O.O.

Brevemente... algunos aspectos sobre

Las clases:

- Pueden ser privadas/protegidas/públicas
- Siempre hay una clase main
- Están organizadas en paquetes (*packages*)

Los datos (variables):

- Pueden ser privadas/protegidas/públicas
- Pueden ser de instancia o de clase (*static*) (Valor compartido por todas las instancias de la clase).

Los métodos:

- Pueden ser privadas/protegidas/públicas
- Pueden ser de instancia o de clase (*static*) (*Acceso a variables de instancia o de clase*)

Programación O.O.

Datos/métodos de clase o estáticos

```
class Cuenta {  
    long numero;  
    String titular;  
    float saldo;  
    float interesAnual;  
  
    // Contador de operaciones  
    static int nOp = 0;  
  
    static int leerNOperaciones() { return nOp; }  
  
    // Operación estática auxiliar de conversión  
    static long eurosAPesetas(float euros) { return euros * 166.386f; }  
  
    void ingreso(float cantidad) { saldo += cantidad; ++nOp; }  
  
    void reintegro(float cantidad) { saldo -= cantidad; ++nOp; }  
}
```

Programación O.O.

Encapsulación

Cuenta
-numero: Long
-titular: String
-saldo: Float
-interésAnual: Real
+ingreso(cantidad: Integer)
+reintegro(cantidad: Integer)
+ingresosInteresMes()
+enRojos(): Boolean
+leerSaldo(): Integer

```
class Cuenta {  
    private long numero;  
    private String titular;  
    private float saldo;  
    private float interesAnual;  
  
    public void ingreso(float cantidad) {  
        saldo += cantidad;  
    }  
  
    public void reintegro(float cantidad) {  
        saldo -= cantidad;  
    }  
  
    public void ingresosInteresMes() {  
        saldo += interesAnual * saldo / 1200;  
    }  
  
    public boolean enRojos() { return saldo < 0; }  
    public float leerSaldo() { return saldo; }  
}
```

Programación O.O.

Paquetes:

Permiten
Agrupar,
clases,
interfaces,
excepciones,
constantes,
etc.

Esta clase está
en
Cuenta.java
y en el paquete
Bancopkg

```
class Cuenta {  
    long numero;  
    String titular;  
    private float saldo;  
    float interesAnual;  
  
    public void ingreso(float cantidad) {  
        saldo += cantidad;  
    }  
  
    public void reintegro(float cantidad) {  
        saldo -= cantidad;  
    }  
  
    public void ingresoInteresMes() {  
        saldo += interesAnual * saldo / 1200;  
    }  
  
    public boolean enRojos() { return saldo < 0; }  
    public float leerSaldo() { return saldo; }  
}  
  
class Banco {  
    Cuenta[] c; // vector de cuentas  
    ...  
}
```

Puede estar en el
mismo archivo

Programación O.O.

Encapsulación de clases

```
public class Cuenta {  
    private long numero;  
    private String titular;  
    private float saldo;  
    private float interesAnual;  
  
    public void ingreso(float cantidad) {  
        saldo += cantidad;  
    }  
  
    public void reintegro(float cantidad) {  
        saldo -= cantidad;  
    }  
  
    public void ingresoInteresMes() {  
        saldo += interesAnual * saldo / 1200;  
    }  
  
    public boolean enRojos() { return saldo < 0; }  
    public float leerSaldo() { return saldo; }  
}
```

- Java sólo permite una clase pública por fichero fuente
- El nombre de la clase y el fichero deben coincidir obligatoriamente
- Dentro de un paquete, no existen restricciones respecto a la utilización de una clase por las otras

Programación O.O.

Constructores

```
public class Cuenta {  
    private long numero;  
    private String titular;  
    private float saldo;  
    private float interesAnual;  
  
    Cuenta(long aNumero, String aTitular, float aInteresAnual) {  
        numero = aNumero;  
        titular = aTitular;  
        saldo = 0;  
        interesAnual = aInteresAnual;  
    }  
  
    public void ingreso(float cantidad) {  
        saldo += cantidad;  
    }  
  
    // Resto de operaciones de la clase Cuenta a partir de aquí
```

Programación O.O.

Sobrecarga de operadores

```
// Importar todas las clases del paquete java.io
import java.io.*;

public class Cuenta {
    private long numero;
    private String titular;
    private float saldo;
    private float interesAnual;

    // Constructor general
    Cuenta(long aNumero, String aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;
    }
}
```

Programación O.O.

```
// Constructor para obtener los datos de la cuenta de un fichero
Cuenta(long aNumero) throws FileNotFoundException, IOException,
    ClassNotFoundException {

    FileInputStream fis = new FileInputStream(aNumero + ".cnt");
    ObjectInputStream ois = new ObjectInputStream(fis);
    numero = aNumero;
    titular = (String) ois.readObject();
    saldo = ois.readFloat();
    interesAnual = ois.readFloat();
    ois.close();
}

public void ingreso(float cantidad) {
    saldo += cantidad;
}

public void reintegro(float cantidad) {
    saldo -= cantidad;
}

public void ingresoInteresMes() {
    saldo += interesAnual * saldo / 1200;
}

public boolean enRojos() { return saldo < 0; }
public float leerSaldo() { return saldo; }
```



Programación O.O.

- Constructor por defecto
- Encapsulación de constructores
(son públicos)

```
Cuenta() {  
    numero = "00000000";  
    titular = "ninguno";  
    saldo = 0;  
    interesAnual = 0;  
}
```

```
// Constructor general  
public Cuenta(long aNumero, String aTitular, float aInteresAnual) {  
    numero = aNumero;  
    titular = aTitular;  
    saldo = 0;  
    interesAnual = aInteresAnual;  
}  
  
// Constructor para obtener los datos de la cuenta de un fichero  
public Cuenta(long aNumero) throws FileNotFoundException,  
    IOException, ClassNotFoundException {  
    FileInputStream fis = new FileInputStream(aNumero + ".cnt");  
    ObjectInputStream ois = new ObjectInputStream(fis);  
    numero = aNumero;  
    titular = (String)ois.readObject();  
    saldo = ois.readFloat();  
    interesAnual = ois.readFloat();  
    ois.close();  
},
```

Programación O.O.

Definición de Objetos

```
Cuenta c; // Una referencia a un objeto de la clase Cuenta  
c = new Cuenta(18400200, "Pedro Jiménez", 0.1f);
```

```
Cuenta c;  
float in;  
long num;  
  
in = 0.1f;  
num = 18400200;  
  
c = new Cuenta(num, "Pedro Jiménez", in);
```

Programación O.O.

Observad que ...

- Las cadenas de caracteres se implementan con una clase (String). Sin embargo no suele ser necesaria su creación de manera explícita, ya que Java lo hace de manera automática al asignar una cadena constante

```
String s; // Una referencia a un objeto de la clase String  
  
// Conexión de la referencia s con un objeto String  
// creado dinámicamente e inicializado con la constante "Pedro"  
s = "Pedro";  
  
// Sería equivalente a:  
// char[] cc = {'P', 'e', 'd', 'r', 'o'}  
// s = new String(cc);
```

- Los arrays también deben ser creados dinámicamente con new como si fueran objetos

```
int[] v; // Una referencia a un vector de enteros  
v = new int[10] // Creación de un vector de 10 enteros
```

Programación O.O.

Y que ...

- Si el array es de referencias a objetos, habrá que crear además cada uno de los objetos referenciados por separado

```
Cuenta[] v; // Un vector de referencias a objetos de la clase Cuenta
int c;

v = new Cuenta[10] // Crear espacio para 10 referencias a cuentas
for (c = 0; c < 10; c++)
    v[c] = new Cuenta(18400200 + c, "Cliente n. " + c, 0.1f);
```

- La destrucción de los objetos se realiza de manera automática cuando el recolector de basura detecta que el objeto no está siendo usado, es decir, no está conectado a ninguna referencia

```
Cuenta c1 = new Cuenta(18400200, "Cliente 1", 0.1f);
Cuenta c2 = new Cuenta(18400201, "Cliente 2", 0.1f);

c1 = c2
// El objeto asociado a la cuenta 18400200 ha
// quedado desconectado y será eliminado por el
// recolector de basura
```

Programación O.O.

Otra observación: Wrappers:

Java proporciona objetos que representan los tipos de datos primitivos. Ej: Integer, Float, Double, Boolean, etc.

```
Integer n = new Integer ("4");
int m = n.intValue();
```

```
String x = "234"
int i = Integer.parseInt (x);
```

Programación O.O.

Trabajar con objetos

```
Cuenta c1 = new Cuenta(18400200, "Pedro Jiménez", 0.1f);
Cuenta c2 = new Cuenta(18400201);

c2.reintegro(1000);
c1.ingreso(500);

if (c2.enRojos())
    System.out.println("Atención: cuenta 18400201 en números rojos");

System.out.println("Saldo actual de la cuenta 18400201: " + c1.leerSaldo());

c1 = new Cuenta(18400202);
// El objeto asociado a la Cuenta 18400200 queda desconectado
c1.ingreso(500);

System.out.println("Saldo actual de la cuenta 18400202: " + c1.leerSaldo());
```

Programación O.O.

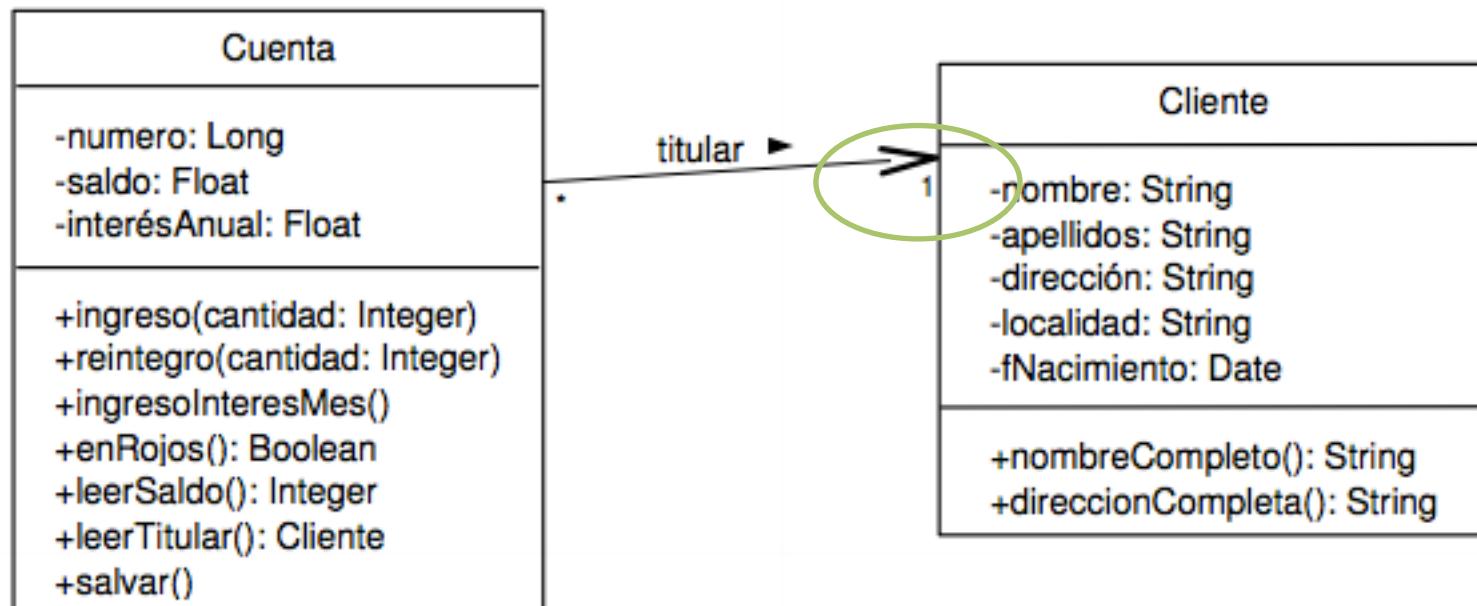
```
BufferedReader br = new BufferedReader(InputStreamReader(System.in));  
  
long nc;  
float mi;  
try {  
    System.out.println("Introduzca núm. de cuenta: ");  
    nc = Long.parseLong(br.readLine());  
  
    System.out.println("Introduzca importe a retirar: ");  
    mi = Float.parseFloat(br.readLine());  
}  
catch(Exception e) {  
    System.out.println("Error al leer datos");  
    return;  
}  
  
Cuenta c;  
try {  
    c = new Cuenta(nc);  
}  
catch(Exception e) {  
    System.out.println("Imposible recuperar cuenta");  
    return;  
}  
  
if (c.leerSaldo() < mi)  
    System.out.println("Saldo insuficiente");  
else  
    c.reintegro(mi);  
c.salvar();
```

Se pide el número de cuenta del Usuario y la cantidad a retirar

Ojo con la lectura, se verá más adelante

Programación O.O.

Asociación



Programación O.O.

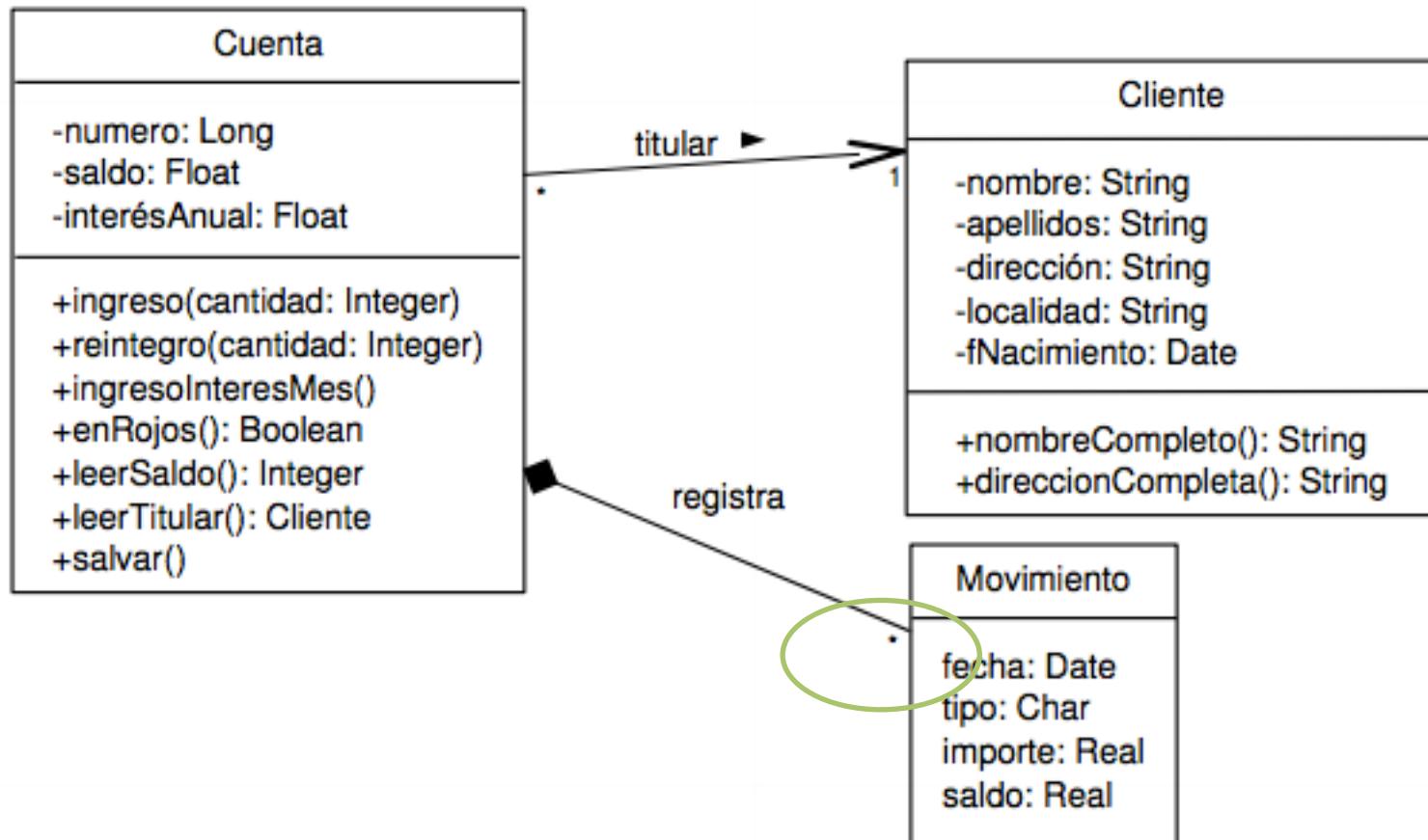
Asociación

```
public class Cliente {  
    private String nombre, apellidos;  
    private String direccion, localidad;  
    private Date fNacimiento;  
  
    Cliente(String aNombre, String aApellidos, String aDireccion,  
            String aLocalidad, Date aFNacimiento) {  
        nombre = aNombre;  
        apellidos = aApellidos;  
        direccion = aDireccion;  
        localidad = aLocalidad;  
        fNacimiento = aFNacimiento;  
    }  
  
    String nombreCompleto() { return nombre + " " + apellidos; }  
    String direccionCompleta() { return direccion + ", " + localidad; }  
}
```

```
public class Cuenta {  
    private long numero;  
    private Cliente titular;  
    private float saldo;  
    private float interesAnual;  
  
    // Constructor general  
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual)  
    {  
        numero = aNumero;  
        titular = aTitular;  
        saldo = 0;  
        interesAnual = aInteresAnual;  
    }  
  
    Cliente leerTitular() { return titular; }  
  
    // Resto de operaciones de la clase Cuenta a partir de aquí
```

Programación O.O.

Colecciones



Programación O.O.

Colecciones

```
import java.util.Date

class Movimiento {
    Date fecha;
    char tipo;
    float importe;
    float saldo;

    public Movimiento(Date aFecha, char aTipo, float aImporte, float aSaldo) {
        fecha = aFecha;
        tipo = aTipo;
        importe = aImporte;
        saldo = aSaldo;
    }
}
```

Programación O.O.

Colecciones

```
public class Cuenta {  
    private long numero;  
    private Cliente titular;  
    private float saldo;  
    private float interesAnual;  
    private LinkedList movimientos; // Lista de movimientos  
  
    // Constructor general  
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {  
        numero = aNumero; titular = aTitular; saldo = 0; interesAnual = aInteresAnual;  
        movimientos = new LinkedList();  
    }  
  
    // Nueva implementación de ingreso y reintegro  
    public void ingreso(float cantidad) {  
        movimientos.add(new Movimiento(new Date(), 'I', cantidad, saldo += cantidad));  
    }  
  
    public void reintegro(float cantidad) {  
        movimientos.add(new Movimiento(new Date(), 'R', cantidad, saldo -= cantidad));  
    }  
  
    public void ingresoInteresMes() { ingreso(interesAnual * saldo / 1200); }  
  
    // Resto de operaciones de la clase Cuenta a partir de aquí
```

Programación O.O.

Programación O.O.

Colecciones e Iteradores

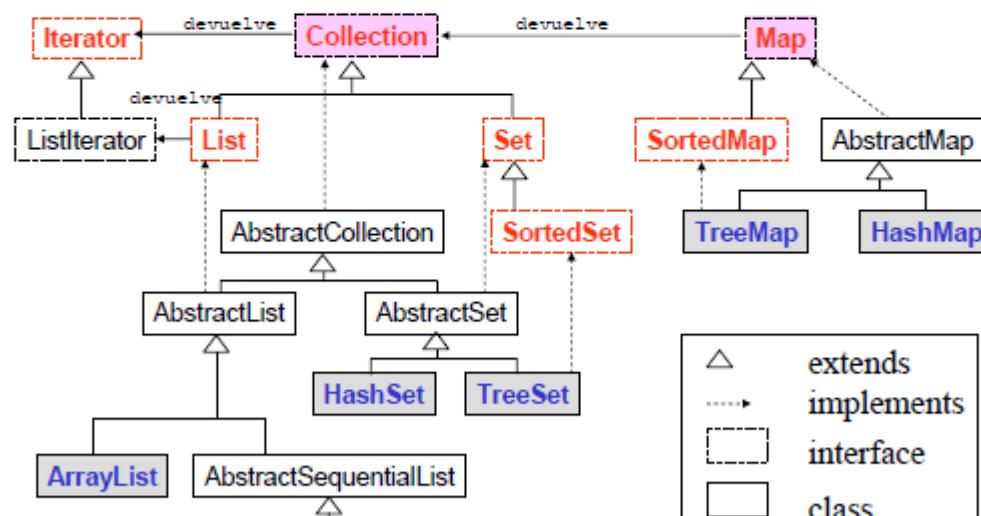
- interfaz Collection:

- ArrayList,
- LinkedList,
- HashSet,
- TreeSet

- interfaz Map

- TreeMap,
- HashMap

- Iteradores: interfaz Iterator



3

Programación O.O.

Colecciones:(paquete **java.util**)

List: Son listas, y los elementos se puede repetir

-**ArrayList:** Lista utilizando un array modifiable. Es costoso añadir o modificar un elemento cerca del principio de la lista, si ésta es grande. Poco costoso de crear, y rápido en accesos aleatorios.

-**LinkedList:** Lista doblemente enlazada. Modificación poco costosa para cualquier tamaño, pero el acceso aleatorio es lento. Util para colas y pilas.

.

Programación O.O.

Colecciones:(paquete **java.util**)

Set: Son conjuntos y los elementos no se pueden repetir

-**HashSet.** Conjunto implementado mediante una tabla Hash. Buena implementación de propósito general, por lo que la búsqueda, adición y eliminación son insensibles al tamaño de los contenidos.

-**TreeSet:** Conjunto implementado mediante un árbol binario. Es más lento para insertar o modificar que el anterior, pero mantiene los elementos ordenados. Asume que los elementos son comparables.

Programación O.O.

- Ejemplo

```
import java.util.*; interfaz  
  
public class TestListas { Clase concreta  
    public static void main( String args[] ) {  
        List<Integer> lista = new ArrayList<Integer>();  
  
        for( int i=0; i < Integer.parseInt(args[0]); i++ )  
            lista.add(i); autoboxing  
  
        //recorrido con iterador  
        Iterator<Integer> it = lista.iterator();  
        while( it.hasNext() )  
            System.out.println(it.next());  
  
        //recorrido bucle "for each"  
        for (Integer ent : lista)  
            System.out.println(ent);  
    }  
}
```

Programación O.O.

Colecciones:(paquete **java.util**)

Maps: Son Conjuntos de datos que tienen dos valores <clave, valor>

-**HashMap**. Tiempos cortos de búsqueda e inserción.

-**TreeMap**: Conjunto implementado mediante un árbol binario. Datos ordenados, por tanto, tiempos de búsqueda rápidos por clave.

Programación O.O.

```
public class TestMapa{
    public static void main(String [] args){
        Map<String,Empleado> plantilla =
            new HashMap<String,Empleado>();
        plantilla.put("34806408V", new Empleado("Pedro Mtnez"));
        plantilla.put("34186581A", new Empleado("Pablo Fdez"));
        plantilla.remove("34806408V");
        System.out.println(plantilla.get("34186581A"));
        //recorrer todas las entradas del mapa
        for (Map.Entry<String,Empleado> entrada : plantilla.entrySet())
        {
            String clave = entrada.getKey();
            Empleado e = entrada.getValue();
            System.out.println("clave = "+clave+", valor = "+e);
        }
    }
}
```

Programación O.O.

Autoreferencias: THIS

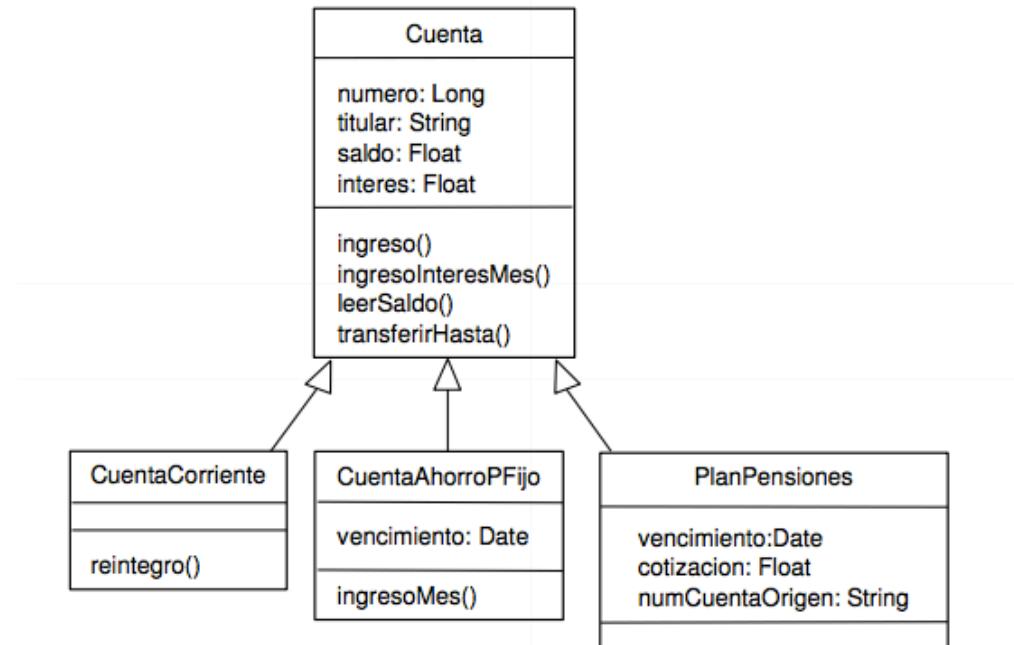
```
public class Cuenta {  
    private long numero;  
    private Cliente titular;  
    private float saldo, interesAnual;  
  
    public void ingresoInteresMes() { this.ingreso(interesAnual * saldo / 1200); }  
  
    // Resto de las operaciones de la clase Cuenta
```

```
public class Cuenta {  
    private long numero;  
    private Cliente titular;  
    private float saldo, interesAnual;  
  
    public void transferirHasta(Cuenta c, float cant) {  
        reintegro(cant); c.ingreso(cant);  
    }  
    public void transferirDesde(Cuenta c, float cant) {  
        c.transferirHasta(this, cant);  
    }  
    // Resto de las operaciones de la clase Cuenta
```

Nos permite implementar la operación transferirDesde() llamando a una operación transferirHasta(),

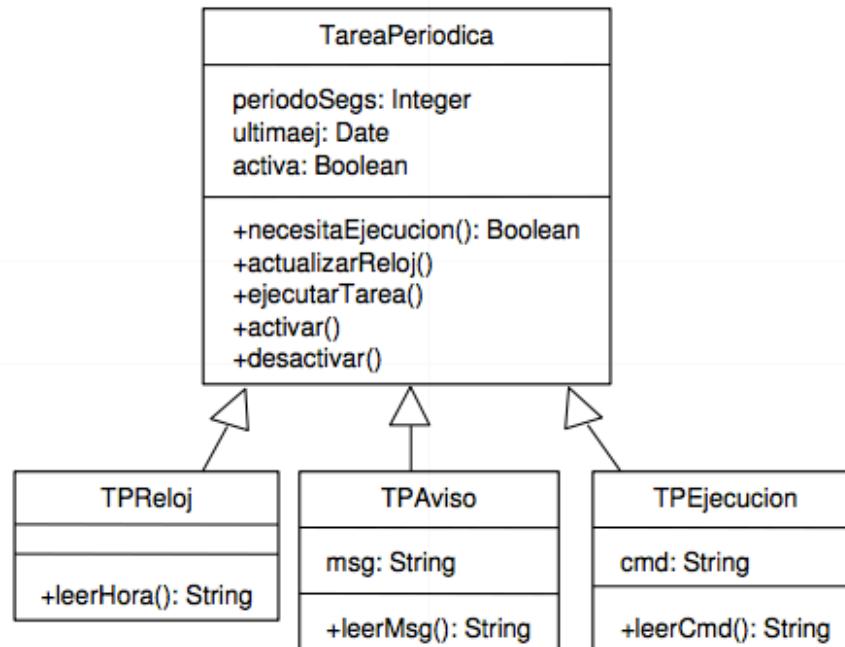
Programación O.O.

Herencia



Programación O.O.

Herencia: Ejemplo



Programación O.O.

Herencia: Ejemplo

Clase TareaPeriódica

```
import java.util.*;  
  
public class TareaPeriodica {  
    int periodoSegs; // Periodo de ejecución  
    Date ultimaEj; // Hora de última ejecución  
    boolean activa;  
  
    public TareaPeriodica(int aPeriodoSegs) {  
        periodoSegs = aPeriodoSegs;  
        actualizarReloj();  
        activa = true;  
    }  
  
    // Constructor para ejecuciones cada segundo  
    public TareaPeriodica() {  
        this(1);  
    }  
  
    // Establecer la última ejecución a la hora actual  
    public void actualizarReloj() {  
        ultimaEj = new Date(); // Hora actual  
    }  
}
```

Programación O.O.

Herencia: Ejemplo

```
public boolean necesitaEjecucion() {  
    if (!activa)  
        return false;  
  
    // Calcular la hora de la próxima ejecución  
    Calendar calProximaEj = new GregorianCalendar();  
    calProximaEj.setTime(ultimaEj);  
    calProximaEj.add(Calendar.SECOND, periodoSegs);  
  
    Calendar calAhora = new GregorianCalendar();  
  
    // Comprobar si ha pasado a la hora actual  
    return (calProximaEj.before(calAhora));  
}  
  
public void ejecutarTarea() {  
    System.out.println("Ejecucion de tarea");  
}  
  
public void activar() { activa = true; }  
public void desactivar() { activa = false; }  
}
```

Programación O.O.

Herencia:

Para que una clase herede de otra se usa el operador **extends**

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TPReloj extends TareaPeriodica {

    public TPReloj() {
        periodoSegs = 60; // Comprobar cada minuto
        actualizarReloj();
        activa = true;
    }

    public String leerHora() {
        Calendar cal = new GregorianCalendar();
        return cal.get(Calendar.HOUR_OF_DAY) + ":" + cal.get(Calendar.MINUTE);
    }
}
```

Ojo, como definimos el constructor – no hay nada extraordinario

Programación O.O.

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
        msg = aMsg;
    }

    public String leerMsg() { return msg; }
}

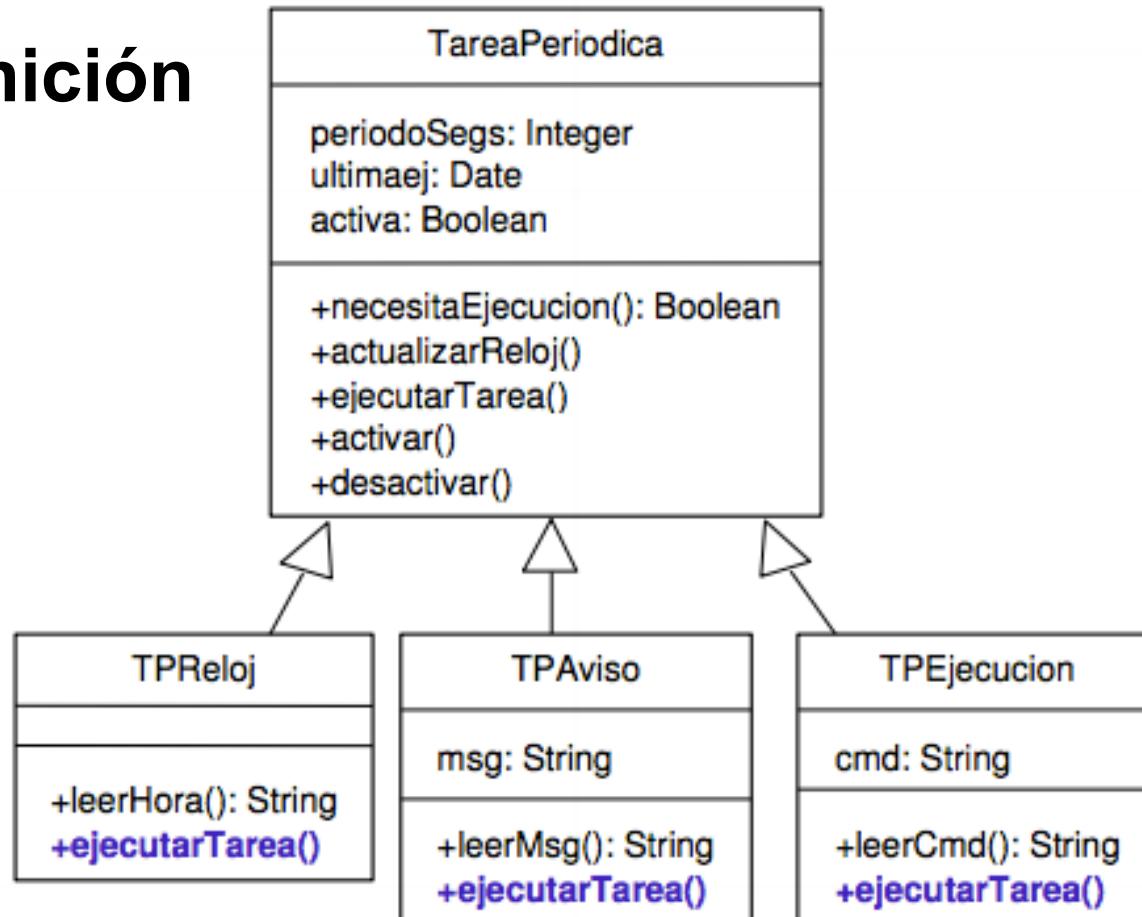
import java.lang.Runtime;
import java.io.IOException;

public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        periodoSegs = aPeriodoSegs;
        actualizarReloj();
        activa = true;
        cmd = aCmd;
    }

    String leerCmd() { return cmd; }
}
```

Redefinición



Programación O.O.

Redefinición

Ojo con la definición de los constructores
Uso del operador SUPER

```
import java.util.Calendar;
import java.util.GregorianCalendar;

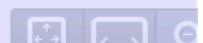
public class TPReloj extends TareaPeriodica {

    public TPReloj() {
        super(60);
    }

    public String leerHora() {
        Calendar cal = new GregorianCalendar();
        return cal.get(Calendar.HOUR_OF_DAY) + ":" + cal.get(Calendar.MINUTE);
    }

    public void ejecutarTarea() {
        Calendar cal = new GregorianCalendar();
        int min = cal.get(Calendar.MINUTE);

        if (min == 0 || min == 30)
            System.out.println("Hora: " + cal.get(Calendar.HOUR_OF_DAY)
                + " " + min);
    }
}
```



Programación O.O.

Redefinición

```
import java.lang.Runtime;
import java.io.IOException;

public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        super(aPeriodoSegs);
        cmd = aCmd;
    }

    String leerCmd() { return cmd; }

    public void ejecutarTarea() {
        try {
            Runtime.getRuntime().exec(cmd);
        }
        catch(IOException e) {
            System.out.println("Imposible ejecutar comando: "
                               + cmd);
        }
    }
}
```

Programación O.O.

Redefinición

```
public class TPAviso extends TareaPeriodica {  
    String msg;  
  
    public TPAviso(String aMsg, int aPeriodoSegs) {  
        super(aPeriodoSegs);  
        msg = aMsg;  
    }  
  
    public String leerMsg() { return msg; }  
  
public void ejecutarTarea() {  
    System.out.println("ATENCIÓN AVISO: " + msg);  
    desactivar();  
}  
}
```

Programación O.O.

Redefinición

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TareaPeriodica tp = new TareaPeriodica(5);
        TPAviso tpa = new TPAviso("Estudiar Programación Avanzada !", 60);
        TPEjecucion tpe = new TPEjecucion("rm ~/tmp/*", 3600);

        while (!tp.necesitaEjecucion())
            System.println("Esperando ejecución de tarea periódica...");
        tp.ejecutarTarea();

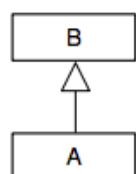
        while (!tpa.necesitaEjecucion())
            System.println("Esperando ejecución de aviso...");
        tpa.ejecutarTarea();

        while (!tpe.necesitaEjecucion())
            System.println("Esperando ejecución de comando...");
        tpe.ejecutarTarea();
    }
}
```

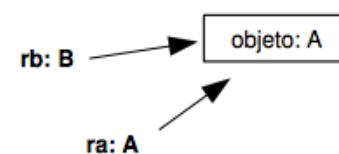
Programación O.O.

Polimorfismo

- Son dos mecanismos relacionados que otorgan a la OOP una gran potencia frente a otros paradigmas de programación
- Únicamente tienen sentido por la existencia de la herencia
- El polimorfismo (o upcasting) consiste en la posibilidad de que una referencia a objetos de una clase pueda conectarse también con objetos de descendientes de ésta



```
A ra = new A(); // Asignación ordinaria  
B rb = ra; // Asignación polimorfa  
B rb = new A(); // Asignacion polimorfa
```



Polimorfismo

Programación O.O.

Polimorfismo

```
public class AppGestorTareas {
    public static void main(String[] args) {
        TPReloj tpr = new TPReloj();
        TPAviso tpa = new TPAviso("Ha pasado un minuto", 60);
        TPEjecucion tpe = new TPEjecucion("/bin/sync", 120);
        TareaPeriodica tp;

        tp = tpr;
        tp.desactivar();

        tp = tpa;
        tp.desactivar();

        tp = tpe;
        tp.desactivar();
    }
}
```

Programación O.O.

Polimorfismo

```
public class AppGestorTareas {
    private static void esperarEjecutar(TareaPeriodica tp)
    {
        while (!tp.necesitaEjecucion());
            tp.ejecutarTarea();
    }

    public static void main(String[] args) {
        TPReloj tpr = new TPReloj();
        TPAviso tpa = new TPAviso("Ha pasado un minuto", 60);
        TPEjecucion tpe = new TPEjecucion("/bin/sync", 120);

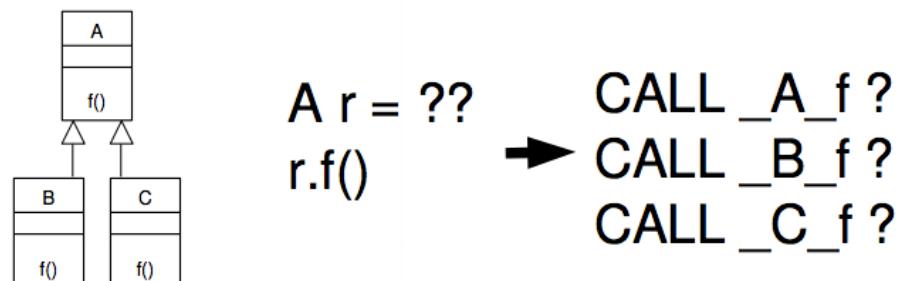
        esperarEjecutar(tpr);
        esperarEjecutar(tpa);
        esperarEjecutar(tpe);
    }
}
```

La mayoría de las veces, las conexiones polimórficas se realizan de manera implícita en el paso de argumentos a una operación. De esta manera es posible escribir operaciones polimórficas que reciban objetos de múltiples clases

Programación O.O.

Ligadura dinámica

Entendemos por resolución de una llamada el proceso por el cual se sustituye una llamada a una función por un salto a la dirección que contiene el código de esta función



La solución consiste en esperar a resolver la llamada al tiempo de ejecución, cuando se conoce realmente los objetos conectados a r, y cuál es la versión de f() apropiada. Este enfoque de resolución de llamadas se denomina ligadura dinámica y es mucho más lenta y compleja que la estática

Programación O.O.

- Ligadura dinámica es la que utiliza Java

```
public class AppGestorTareas {  
    public static void main(String[] args) {  
        TPReloj tpr = new TPReloj();  
        TPAviso tpa = new TPAviso("Ha pasado un minuto", 60);  
        TPEjecucion tpe = new TPEjecucion("/bin-sync", 120);  
        TareaPeriodica tp;  
  
        tp = tpr;  
        tp.ejecutarTarea(); // Versión de TPReloj  
  
        tp = tpa;  
        tp.ejecutarTarea(); // Versión de TPAviso  
  
        tp = tpe;  
        tp.ejecutarTarea(); // Versión de TPEjecucion  
    }  
}
```

Streams and I/O

- * Clases básicas de Entrada y Salida

- FileInputStream, para leer de archivo
 - FileOutputStream, para escribir en archivo

- * Example:

Abrir el archivo "myfile.txt" para **lectura**

```
FileInputStream fis = new FileInputStream("myfile.txt");
```

Abrir un archivo "outfile.txt" para **escritura**

```
FileOutputStream fos = new FileOutputStream ("myfile.txt");
```

Streams and I/O

```
import java.io.*;

public class ESBinaria {
    public static void main(String[] args) {
        DataOutputStream ds =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("datos")));

        ds.writeInt(2);
        ds.writeFloat(4.5);
        ds.writeChars("Hola");
        ds.close();

        ds = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("datos")));

        int k = ds.readInt();
        System.out.println(k);
        ds.close();
    }
}
```

Streams and I/O

- Las clases BufferedReader y BufferedWriter permite leer y escribir información utilizando un buffer intermedio para acelerar las operaciones. Posibilitan el leer/escribir líneas completas
- Las clases InputStreamReader y OutputStreamWriter permiten convertir un stream en un reader/writer
- La clase especializada PrintWriter permite escribir directamente cualquier tipo de dato en un writer. Su uso es más cómodo que el de un BufferedWriter
- La clase especializada Scanner permite leer de manera sencilla cualquier tipo simple de un fichero de texto. Su uso es más cómodo que mediante BufferedReader

Streams and I/O

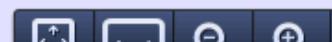
```
import java.io.*;

public class ESTexto {
    public static void main(String[] args) {
        try {
            BufferedWriter bw = new BufferedWriter(new FileWriter("datos.txt"));
            bw.write("Hola");
            bw.newLine();
            bw.write(new Integer(3).toString());
            bw.newLine();
            bw.write(new Float(10.3).toString());
            bw.close();

            PrintWriter pw = new PrintWriter("datos.txt");
            pw.println("Hola");
            pw.println(3);
            pw.println(10.3);
            pw.close();
        }
        catch(IOException e) {
            System.out.println("Error de E/S");
        }
    }
}
```

Dos métodos:

- Wrappers
- Clase PrintWriter



Streams and I/O

```
import java.io.*;
import java.util.Scanner;

public class ESTexto {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(
                new FileReader("datos.txt"));
            String s = br.readLine();
            int k = Integer.parseInt(br.readLine());
            float p = Float.parseFloat(br.readLine());
            br.close();

            Scanner sc = new Scanner(new File("datos.txt"));
            s = sc.nextLine();
            k = sc.nextInt();
            p = sc.nextFloat();
            sc.close();
        }
        catch(IOException e) {
            System.out.println("Error de E/S");
        }
    }
}
```

Dos métodos:
• Wrappers
• Clase Scanner

Serialización

Una de las características más potentes de Java es la posibilidad de serializar un objeto, es decir, **convertirlo en una secuencia de bytes y enviarlo a un fichero en disco**, por un socket a otro ordenador a través de la red, etc. El proceso sería el siguiente:

- Declarar la implementación de la interfaz `Serializable` en la clase que deseemos serializar. Se trata de una interfaz vacía, por lo que no hay operaciones que implementar
- Para serializar el objeto crearíamos un stream `ObjectOutputStream` y escribiríamos el objeto mediante la operación `writeObject()`
- Para "dессerialize" el objeto crearíamos un stream `ObjectInputStream`, leeríamos el objeto mediante `readObject()` y realizaremos un casting a la clase del objeto

Serialización

Vamos a modificar ahora el constructor de la clase Cuenta y la operación **salvar()** para que sean capaces de cargar y salvar el histórico de movimientos. La capacidad de serialización de Java permite salvar la lista enlazada de un golpe

```
import java.io.*;
import java.util.*;

// Es necesario que tanto las clases Cliente como Movimiento implementen la interfaz
// Serializable para que los objetos puedan ser escritos en disco

class Movimiento implements Serializable {
    Date fecha;
    char tipo;
    float importe;
    float saldo;

    public Movimiento(Date aFecha, char aTipo, float aImporte, float aSaldo) {
        fecha = aFecha;
        tipo = aTipo;
        importe = aImporte;
        saldo = aSaldo;
    }
}

public class Cuenta {
    long numero;
    Cliente titular;
    private float saldo;
    float interesAnual;

    LinkedList movimientos;
```

Serialización

```
public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {  
    numero = aNumero;  
    titular = aTitular;  
    saldo = 0;  
    interesAnual = aInteresAnual;  
  
    movimientos = new LinkedList();  
}  
  
Cuenta(long aNumero) throws FileNotFoundException,  
    IOException, ClassNotFoundException {  
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(aNumero + ".cnt"));  
    numero = ois.readLong();  
    titular = (Cliente) ois.readObject();  
    saldo = ois.readFloat();  
    interesAnual = ois.readFloat();  
    movimientos = (LinkedList) ois.readObject();  
    ois.close();  
}  
  
void salvar() throws FileNotFoundException, IOException {  
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(numero + ".cnt"));  
    oos.writeLong(numero);  
    oos.writeObject(titular);  
    oos.writeFloat(saldo);  
    oos.writeFloat(interesAnual);  
    oos.writeObject(movimientos);  
    oos.close();  
}  
  
// Resto de operaciones de la clase Cuenta a partir de aquí
```

Excepciones

Muy breve sobre excepciones:

- se podrá evitar repetir continuamente código, en busca de un posible error, y avisar a otros objetos de una condición anormal de ejecución durante un programa:

```
if ( error == true ) return ERROR;
```

- Tipos
 - Error: Excepciones que indican problemas muy graves, que suelen ser no recuperables y no deben casi nunca ser capturadas.
 - Exception: Excepciones no definitivas, pero que se detectan fuera del tiempo de ejecución.
 - RuntimeException: Excepciones que se dan durante la ejecución del programa.

Excepciones

- Para que el sistema de gestión de excepciones funcione, se ha de trabajar en dos partes de los programas:
 - Definir qué partes de los programas crean una excepción y bajo qué condiciones. Para ello se utilizan las palabras reservadas **throw** y **throws**.
 - Comprobar en ciertas partes de los programas si una excepción se ha producido, y actuar en consecuencia. Para ello se utilizan las palabras reservadas **try**, **catch** y **finally** (se ejecuta tras el try o catch, siempre).

Excepciones

Se interrumpe el método donde se produce la excepción

Se vuelve al método que llamó al actual, buscando un controlador de excepciones

En este método no hay un controlador, por lo que se propaga la excepción

Se sigue la vuelta atrás hasta encontrar un controlador

Método con un controlador específico o genérico de excepciones

Main()
Si se llega aquí sin encontrar un controlador se interrumpe el programa

Excepciones

```
import java.io.*;
import java.util.Scanner;

public class ESTexto {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(
                new FileReader("datos.txt"))
            String s = br.readLine();
            int k = Integer.parseInt(br.readLine());
            float p = Float.parseFloat(br.readLine());
            br.close();

            Scanner sc = new Scanner(new File("datos.txt"));
            s = sc.nextLine();
            k = sc.nextInt();
            p = sc.nextFloat();
            sc.close();
        }
        catch(IOException e) {
            System.out.println("Error de E/S");
        }
    }
}
```

Cuando el programador va a ejecutar un trozo de código que pueda provocar una excepción debe incluirlo en un bloque **try**

Y controlar qué hacer con la posible excepción en el bloque **catch**

Excepciones

- Muchas veces el programador dentro de un determinado método
 - deberá comprobar si alguna condición de excepción se cumple, y si es así lanzarla.

```
tipo_devuelto miMetodoLanzador() throws miExcep1, miExcep2 {  
    if ( condicion_de_excepcion == true )  
        throw new miExcepcion();
```
 - O no controla las excepciones que puede recibir
`void buscaFicha()`
// Lista de excepciones que podría propagar
throws IOException, NullPointerException {...} }
- Debe indicar explícitamente en la cabecera qué excepciones podría retransmitir hacia el método que le haya invocado

Excepciones

- Las excepciones no declaradas han de seguir controlándose en el cuerpo del método
- El mecanismo de propagación es automático, la excepción será enviada al código que hubiese llamado a este método
- También podemos hacernos nuestras propias excepciones. Ir al manual para poder tratar errores propios de nuestro programa. Ir a tutorial:
<http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/excepciones/propias.htm>

Programación O.O.

Y nos queda mucho Java por ver, pero no será en esta práctica, ni en este curso:

- Interfaces
- Clases abstractas
- Excepciones propias
- Subclases
- *final*
- Limpieza de memoria
- Downcasting
- Otras librerías...

Se recomienda acceder a

<http://docs.oracle.com/javase/tutorial/>