

Ingeniería del Software Tema 1

Dr. Vicent-Ramon Palasí Lallana
Universidad Francisco Gavidia.
Junio 2005.

Programa del curso

- 1. Objetivos y metodología del curso.
- 2. Repaso del curso anterior.
- 3. Ampliación del lenguaje Java.
- 4. Cookies y rastreo de sesión.
- 5. Otros aspectos.
- 6. Un ejemplo práctico.

Programa del curso

- 1. **Objetivos y metodología del curso.**
- 2. Repaso del curso anterior.
- 3. Ampliación del lenguaje Java.
- 4. Cookies y rastreo de sesión.
- 5. Otros aspectos.
- 6. Un ejemplo práctico.

Objetivos y metodología del curso

- 1.1. Motivación de aprender el lenguaje Java.
- 1.2. Objetivos del curso.
- 1.3. Enfoque que se usará.
- 1.4. Metodología.

Objetivos y metodología del curso

- 1.1. **Motivación de aprender el lenguaje Java.**
- 1.2. Objetivos del curso.
- 1.3. Enfoque que se usará.
- 1.4. Metodología.

Algunos datos

- El 75% de las aplicaciones corporativas están escritas en Java (BZ Search).
- Java es una industria de 100 mil millones de dólares por año y 110 mil millones en tecnologías relacionadas con Java.
- Mayor implantación en servicios Web, procedimientos almacenados en BD.
- Java es el lenguaje estándar para importantes productos de software:
 - La base de datos Oracle.
 - El ERP SAP.
- El lenguaje único de los servidores de aplicaciones (excepto uno): BEA WebLogic, IBM Websphere, Oracle 9i, Macromedia JRun, JBoss, etc.

Vicent Palasí, PhD, MBA, MEd. Todos los derechos reservados.

Mail: palasi@palasi.com Web: www.palasi.com

Pero además

- **Java es la plataforma que tiene más número de herramientas.** Una inmensa cantidad de ellos: Motores de persistencia, frameworks MVC, librerías de todos los tipos, herramientas de desarrollo y un larguísimo etcétera.
- **Lo que es mejor: la inmensa mayoría de ellos son de código abierto.** Por ejemplo, en Sourceforge, hay **quince mil proyectos en Java**. Pero hay otros sitios: java.net, fundación Apache (aquí veremos algunos: Tomcat, Eclipse, Lomboz, Hibernate).
- Para cualquier área : estadísticas, redes neuronales, ERP, gráficos, celulares, etc. seguro que hay algún software de código abierto en Java.

En los celulares

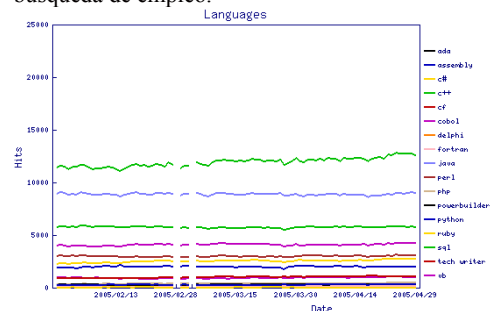
- Hay cerca de 100 implantaciones de Java en celulares y 579 millones de celulares con Java.
- 7 de cada 10 aplicaciones inalámbricas en construcción usarán Java.
- El mercado de juegos Java en celulares es estimado en 3 mil millones de dólares.

En el mercado de trabajo

- Globalmente cerca de 4.5 millones de desarrolladores trabajan con Java.
- Java es el lenguaje de programación más demandado en las búsquedas de empleo (excepto SQL). Por ejemplo, Monster.com.

Por ejemplo, Dice.com

- A parte de SQL (que es diferente), Java encabeza la búsqueda de empleo.



En El Salvador

- La Administración lo ha escogido como el lenguaje estándar de los Ministerios.*
- Es la herramienta que están implementando las grandes empresas: bancos, SIMAN, TACA, etc.
- Sin embargo, desgraciadamente, el conocimiento de Java por parte de los programadores es bajo. Muchas veces se contrata personal extranjero*.
- Por eso, aprender Java **a nivel empresarial*** es una excelente inversión para la carrera profesional.

Objetivos y metodología del curso

- 1.1. Motivación de aprender el lenguaje Java.
- 1.2. **Objetivos del curso.**
- 1.3. Enfoque que se usará.
- 1.4. Metodología.

Objetivo general del curso

- Ser capaz de desarrollar aplicaciones Web en Java **de carácter empresarial.**
- Cuando se diga de carácter empresarial, se está queriendo decir que contemplen todos los aspectos de una aplicación real en una empresa mediana o grande.
- En el curso anterior, algunas de las complejidades de la programación empresarial se evitaban pues se hacía un mayor énfasis en aprender la estructura de una aplicación n-capas para el Web.
- Ha llegado el momento de prestar atención a los detalles y a los problemas que se presentan todos los días.

Objetivos específicos del curso

- Adquirir buenas prácticas de programación que hagan los programas flexibles y escalables.
- Conocer la inmensa mayoría de la sintaxis del lenguaje Java.
- Conocer los aspectos más usuales de la programación Web.
- Organizar las clases en Java en jerarquías de herencia y de interfaces.
- Aprender algunos patrones de diseño especialmente útiles en la programación diaria.

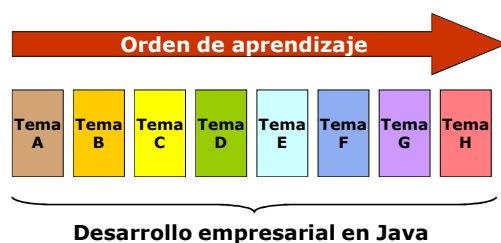
Objetivos y metodología del curso

- 1.1. Motivación de aprender el lenguaje Java.
- 1.2. Objetivos del curso.
- **1.3. Enfoque que se usará.**
- 1.4. Metodología.

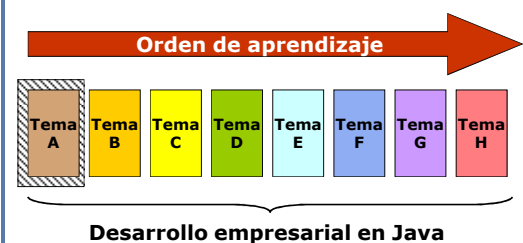
Un enfoque común para aprender Java (temas principales)

- 1. Fundamentos. Compilación, JDK, etc.
- 2. Aspectos procedimentales del lenguaje.
- 3. Programación orientada a objetos.
- 4. Excepciones, librerías, multithreading, etc.
- 5. Interfaz gráfica de escritorio.
- 6. Bases de datos: JDBC
- 7. Programación para el Web: servlets, JSP.
- 8. Arquitectura MVC, n-tier, etc.
- 9. Testing
- 10. Enterprise Javabeans.
- 11. Seguridad.
- 12. Análisis y diseño O-O.
- 13. Buenas prácticas. Patrones de diseño.

Este es un enfoque lineal

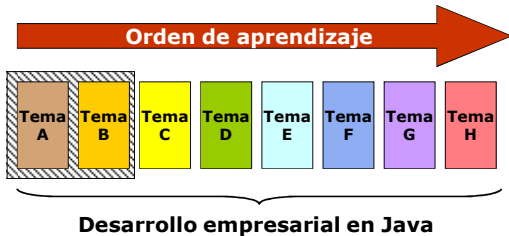


1. El enfoque común para aprender Java



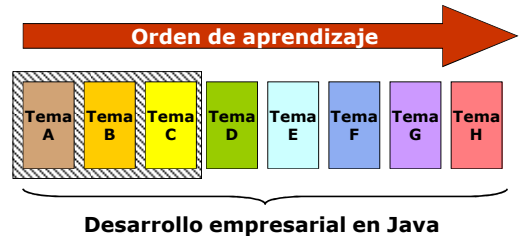
- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

2. El enfoque común para aprender Java



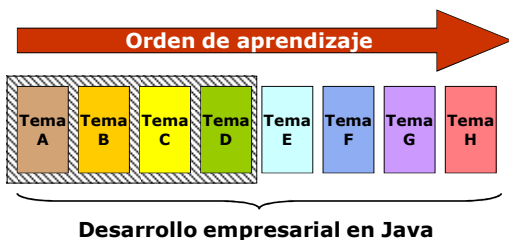
- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

3. El enfoque común para aprender Java



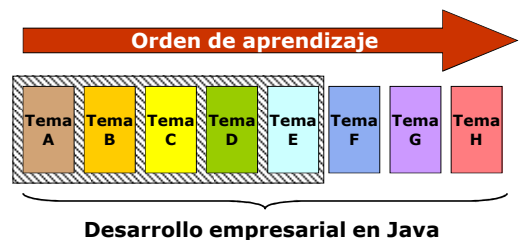
- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

4. El enfoque común para aprender Java



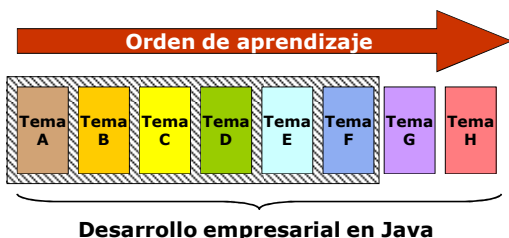
- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

5. El enfoque común para aprender Java



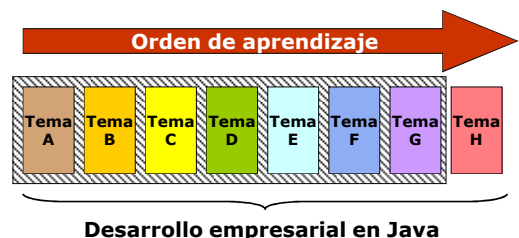
- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

6. El enfoque común para aprender Java



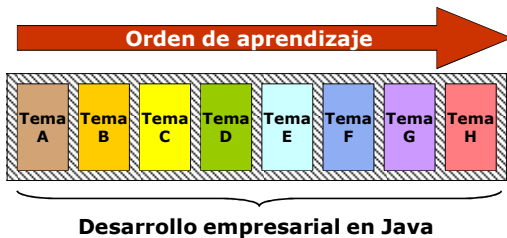
- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

7. El enfoque común para aprender Java



- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

8. El enfoque común para aprender Java



- El cuadrado con trama indica lo que se ha aprendido hasta el momento.

Problemas de este enfoque

- 1. Es un enfoque lineal: se aprende todo un tema y después otro tema.
- 2. El aprendiz tarda mucho tiempo en poder realizar una aplicación medianamente real (hasta JDBC).
- 3. El aprendiz aún tarda más tiempo en poder realizar una aplicación para el Web (el punto fuerte de Java).
- 4. El aprendiz tarda mucho tiempo en saber cómo está estructurada una aplicación Java (hasta arquitectura).
- 5. El aprendiz aprende a hacer las cosas mal y después aprende a hacerlas bien.
- 6. JDBC es difícil de gestionar (excepciones) y no permite fácilmente una metodología O-O.

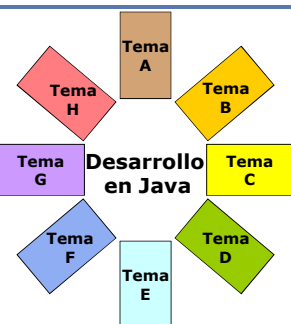
Problemas de este enfoque

- Este enfoque es el causante de que el conocimiento de Java en El Salvador sea tan bajo.
- Como es tan lento, la mayoría de aquellos que reciben cursos de Java sólo aprenden a hacer programas de juguete: con la consola, sin base de datos.
- Hay pocos que sean capaces de programar una aplicación empresarial de tamaño real.
- Aquí es lo que queremos aprender. ¿Cómo lo haremos?

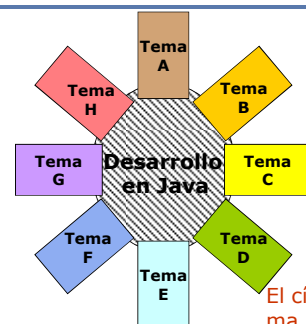
Lo haremos con un enfoque diferente

- 1. Repaso de HTML. Introducción a JSP.
- 2. Elementos de guión (Aspectos procedimentales).
- 3. Programación O-O.
- 4. Bases de datos: Motores de persistencia
- 5. Ampliación O-O.
- 6. Cookies y rastreo de sesión.
- 7. Multithreading.
- 8. Arquitectura MVC.
- 9. Más sobre el Web: cookies, seguridad.
- 10. Análisis y diseño O-O.
- 11. Testing.
- 12. Enterprise Javabeans.
- 13. Seguridad.

Nuestro enfoque es concéntrico

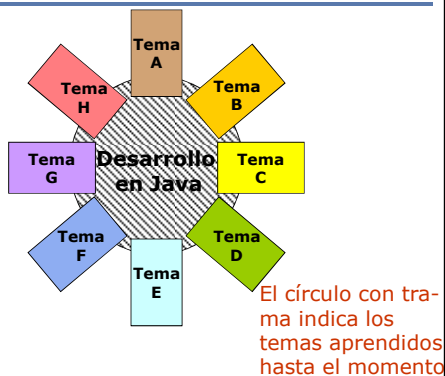


1. Nuestro enfoque es concéntrico

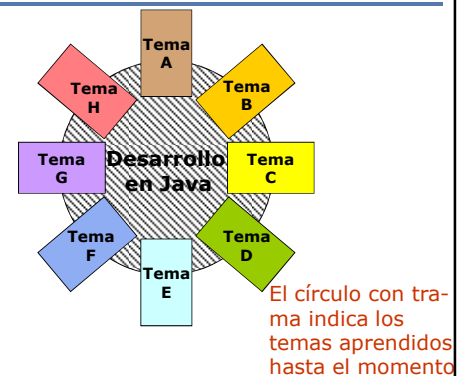


El círculo con trama indica los temas aprendidos hasta el momento

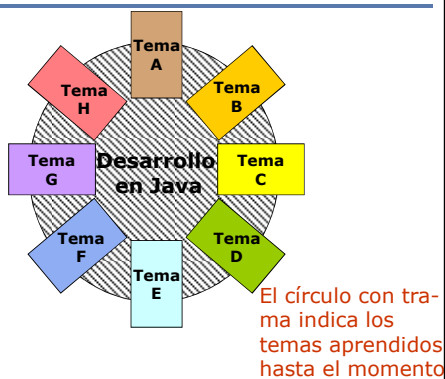
2. Nuestro enfoque es concéntrico



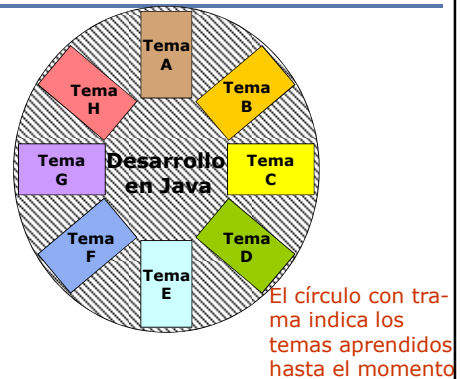
3. Nuestro enfoque es concéntrico



4. Nuestro enfoque es concéntrico



5. Nuestro enfoque es concéntrico



Ventajas de este enfoque

- 1. Es un enfoque concéntrico: se aprende lo principal de los temas y después se va ampliando.
- 2. El aprendiz sabe desde el principio programar una aplicación para el Web.
- 3. El aprendiz sabe tempranamente cómo está estructurada una aplicación Java.
- 4. El aprendiz aprende a hacer las cosas bien desde el principio (n-tier, buenas prácticas y patrones de diseño se aprenden sobre la marcha).
- 5. No se usa JDBC, sino motores de persistencia que son más sencillos y permiten fácilmente una metodología O-O.

El curso anterior

- 1. Repaso de HTML. Introducción a JSP.
- 2. Elementos de guión (Aspectos procedimentales).
- 3. Programación O-O.
- 4. Bases de datos: Motores de persistencia
- 5. Ampliación O-O.
- 6. Cookies y rastreo de sesión.
- 7. Multithreading.
- 8. Arquitectura MVC.
- 9. Más sobre el Web: cookies, seguridad.
- 10. Análisis y diseño O-O.
- 11. Testing.
- 12. Enterprise Javabeans.
- 13. Seguridad.

Este curso

- 1. Repaso de HTML. Introducción a JSP.
- 2. Elementos de gui3n (Aspectos procedimentales).
- 3. Programaci3n O-O.
- 4. Bases de datos: Motores de persistencia
- 5. Ampliaci3n O-O.
- 6. Cookies y rastreo de sesi3n.
- 7. Multithreading??.
- 8. Arquitectura MVC.
- 9. M3s sobre el Web: cookies, seguridad.
- 10. An3lisis y dise1o O-O.
- 11. Testing.
- 12. Enterprise Javabeans.
- 13. Seguridad.

Este curso es de continuaci3n

- El curso anterior hab3amos visto c3mo se estructuraba una aplicaci3n en n-capas para el Web y que hac3a uso de una base de datos.
- En este curso, se partir3 de esta estructura y se har3n mejoras a los diversos componentes.
- La ventaja es que tenemos una visi3n de conjunto y sabremos ubicar las mejoras que se hagan.

Objetivos y metodolog3a del curso

- 1. Motivaci3n de aprender el lenguaje Java.
- 2. Objetivos del curso.
- 3. Enfoque que se usar3.
- 4. Metodolog3a.

Metodolog3a utilizada

- Presentaciones en Powerpoint. Para transmitir los conceptos te3ricos del curso.
- Ejercicios pr3cticos. Para asimilar los conceptos en programas pr3cticos.
- Aula virtual. Para plantear dudas y resolverlas entre todos.

Ventajas de esta metodolog3a

- Combinaci3n de teor3a y pr3ctica.
- Enfoque que va de lo abstracto a lo concreto.
- Fomenta la participaci3n del alumno.
- Asegura el aprendizaje

Algunas indicaciones

- Se valora la participaci3n: tanto espont3nea como programada.
- Se anima a hacer tantas preguntas como se desee: no hay ning3n momento en que una pregunta sea inadecuada.

Prerrequisitos del curso

- El curso empieza desde el conocimiento del curso anterior.
- Dado que ha pasado mucho tiempo desde el curso anterior, vamos a hacer un repaso del mismo.
- Pero esto es un repaso, no volver a dar lo mismo. Se irá a mucha velocidad.
- Se les recomienda que repasen las transparencias del curso anterior y que pregunten dudas.

Notación de estas transparencias

- Las instrucciones MS-DOS se escribirán en **Courier New negrita**.
- Los textos que aparezcan en Windows (textos de menús, de botones, etc) se escribirán en **Arial Negrita**.
- El código fuente en Java (o en general, cualquier contenido de un archivo de texto) se escribirá en letra **Courier New negrita** (normalmente con fondo amarillo).
- Si en alguna parte del código fuente, hay un dato que varía según el programa o el caso concreto (por ejemplo el nombre del programa) se escribirá en **Arial negrita y cursiva**. Por ejemplo, ***class nombreClase***

Notación de estas transparencias

- Cuando se trata de acciones de la computadora (Windows, programas, Web, etc.), la barra vertical | se utiliza para separar las acciones consecutivas. Significa "a continuación".
- Así **Inicio|Panel de control|Sistema** es la forma abreviada de decir.
 - Hacer clic en **Inicio**.
 - A continuación, hacer clic en **Panel de control**.
 - A continuación, hacer clic **en Sistema**.
- **Cuando hay algo entre corchetes [], es que es opcional.**

Programa del curso

- 1. Objetivos y metodología del curso.
- **2. Repaso del curso anterior.**
- 3. Ampliación del lenguaje Java.
- 4. Cookies y rastreo de sesión.
- 5. Otros aspectos.
- 6. Un ejemplo práctico.

2. Repaso del curso anterior

- 2.1. Programación orientada a objetos.
- 2.2. Arquitectura en n-capas.
- 2.3. Capa de dominio.
- 2.4. Capa de datos.
- 2.5. Capa de negocio.
- 2.6. Capa de presentación.

2. Repaso del curso anterior

- **2.1. Programación orientada a objetos.**
- 2.2. Arquitectura en n-capas.
- 2.3. Capa de dominio.
- 2.4. Capa de datos.
- 2.5. Capa de negocio.
- 2.6. Capa de presentación.

Variables en Java

- Declarar una variable
`tipo nombre; int precio;`
- Recuperar el valor de una variable.
`nombre precio`
- Guardar el valor de una variable.
`nombre = expresion;`
`precio = 10;`
- Declarar una variable guardando un valor inicial.
`tipo nombre = expresion;`
`int precio = 10;`

Tipos primitivos. Los más usados están en rojo

Nombre	Valores
byte	Números enteros de -128 a 127
short	Enteros de -32768 a 32767
int	Enteros de -2147483648 a 2147483647
long	Enteros de -922117036854775808 a 922117036854775807
float	Reales de ±3.40282347e+38 a ±1.40239846e-45
double	Reales de ±1.79769313486231570e+308 a ±4.94065645841246544e-324
char	Cualquier carácter Unicode.
boolean	true o false

Un tipo de datos que no es primitivo

- **String**
- Contiene cadenas de caracteres.
- Es una clase no un tipo primitivo.
- Recuerden: se escribe con mayúscula inicial, pues es una clase. Los tipos primitivos se escriben con minúscula inicial.

Operadores de tipo String

- `s1.equals(s2)` Muestra si dos cadenas s1 y s2 son iguales. **NO UTILIZAR ==**
- `+` Concatena dos cadenas.
"Juan "+"Carlos" → "Juan Carlos"
- Si **var** es una variable de tipo **String**
`var.length()` devuelve la longitud de la cadena.
`var.charAt(i)` devuelve carácter en la posición i
`var.substring(i, j)` Devuelve la subcadena que empieza en la posición i y acaba en la posición j-1
(Atención: el primer carácter está en la posición 0)

Estructuras condicionales

```
if (condicion) {
    sentencias
}

if (condicion) {
    sentencias1
} else {
    sentencias2
}

if (condicion1) {
    sentencias1
} else if (condicion2) {
    sentencias2
} else if...
....
} else {
    sentenciasn
}

if (saldo > 0) {
    out.println("Positivo");
} else if (saldo < 0) {
    out.println("Negativo");
} else {
    out.println("Cero");
}
```

Estructuras repetitivas

```
while (condicion) {
    sentencias
}

do {
    sentencias
} while (condicion);

for (inicialización;condicion;actualización) {
    tratar el elemento actual
}

for (int=1; i<=100; i++) {
    out.println(i);
}
```


La programación orientada a objetos se basa en el concepto de objeto

- Un objeto de software es análogo a un objeto de la realidad.
- Por ejemplo, un objeto real es el auto que es propiedad del licenciado.
- De un objeto real, nos interesan dos cosas.



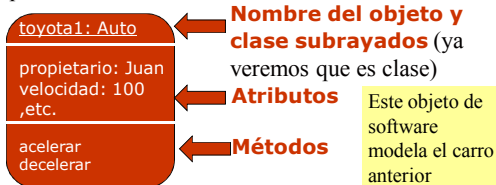
Las dos cosas que nos interesan de un objeto real

- Sus características (su color, tamaño, su potencia, si tiene caja de cambios automática, etc.). A esto se le llama **atributos** (también “propiedades” o “campos”).
- El tipo de acciones que pueden realizarse con él (acelerar, decelerar, encender el motor, etc). A esto se le llama **métodos**.
- A atributos y métodos, se les llama **miembros**



Un objeto de software es

- Es un conjunto de datos y acciones relacionados que
 - un programa trata como una unidad
 - que modela un objeto real.
- Al igual que el objeto real nos interesan sus atributos o propiedades (datos) y las cosas que pueden hacerse con esos datos: sus métodos.



Ejemplo

- Supongamos que estamos creando un programa para gestionar el parqueo de los diferentes carros de los empleados de una empresa.
- Hay una serie de objetos reales, que son el auto del licenciado, el auto de la empresa, etc.
- Nuestro programa puede tener muchos objetos carroDelLicenciado, carroDeLaEmpresa. Cada uno modela a un auto real.
- Cada uno de estos objetos de software tendrá unos atributos (velocidad, propietario) y unos métodos (acelerar, decelerar) que se corresponderán con los atributos y métodos de los objetos reales.

Clases y objetos

- Un programa utiliza una serie de objetos y normalmente muchos de ellos son similares.
- Al conjunto de estos objetos similares se le llama **clase**



Un hecho incuestionable

- En Java (como en otros lenguajes O-O) **todos los objetos pertenecen a una clase.**
- No hay objetos sin clase.
- Por eso, cada vez que creamos un objeto debemos especificar a qué clase pertenecerá el nuevo objeto.

Un hecho importante

- Las clases son tipos de datos.
- En Java todas las clases son tipos de datos.
- Los objetos de una clase son los **valores** del tipo de datos que define la clase.

En Java, todas las clases son tipos de datos

- En Java hay dos tipos de datos:
- Los tipos primitivos o básicos que hemos visto hasta ahora. Van en minúscula. **int**
- Las clases, incluyendo **String**. Van en mayúscula.

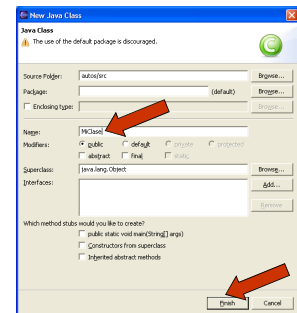
Cómo programamos clases

- En Eclipse, en un proyecto creado, hacemos **File|New|Class**



Cómo se crea una nueva clase

- Se escribe el nombre de la clase al lado de **Name**. Clic en **Finish**

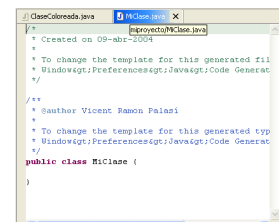


Paréntesis: el nombre de una clase

- En Java, la convención es que las clases empiezan sus nombres con mayúscula.
- Si tienen varias palabras cada una empieza con mayúscula. Ejemplos:

Auto
ClaseNueva
MiClase
Empleado
JefeSupervisor

Cómo se crea una nueva clase



- Aparecerá una ventana, donde podemos escribir el código de la clase.
- Pero, ¿qué código es éste?

Código para declarar (programar) clases en Java

```
package nombrepaquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- Este es un código simplificado. Más adelante veremos que hay más opciones.

Un poco de terminología

```
package nombrepaquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- Al conjunto de atributos y métodos de una clase se les llama **miembros** de la clase.
- Utilizaremos miembros cuando no queramos usar la expresión “atributos y métodos” por muy larga.

Convenciones de notación en la declaración de clases

```
package nombrepaquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- El nombre del paquete debe ir todo en minúsculas.
- El nombre de la clase con la primera letra de cada palabra en mayúscula: **ClaseNueva**.
- El nombre de atributos y métodos comienza en minúscula y las siguientes palabras comienzan en mayúscula: **atributo**, **metodoDePrueba**.

Paquetes

```
package nombrepaquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- Las clases se agrupan en paquetes para organizarlos mejor.
- Piensen en los paquetes como directorios o carpetas de clases.

Paquetes

```
package nombrepaquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- El nombre del paquete debe ir todo en minúsculas y puede tener puntos.
- Cada punto refleja un subpaquete (un subdirectorio de clases)

paquete.subpaquete.subsubpaquete

Paquetes

```
package nombrepaquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- Nombres posibles de paquetes:

```
mipaquete
ufg.encuesta.calculo
com.aurumsol.cursojava.tema2.ejemplos
```


Paquetes

- Es una práctica muy extendida (y una buena práctica) que el nombre del paquete comience con la dirección Web de la empresa invertida (sin www).

```
com.aurumsol.cursojava.tema2.ejemplos
sv.edu.ufg.encuesta.calculo
org.eclipse.plugins.junit
```

- De esta forma, los paquetes (y las clases que contienen) son únicos y no pueden tener conflictos con ningún otro paquete (o clase) del mundo.

Paquetes

- También es una buena práctica que después del URL invertido de la empresa, haya un paquete por cada aplicación y un subpaquete por cada capa.

- Es decir, los paquetes deben tener la estructura **urlinvertida.aplicacion.capa**

- Así,

```
com.aurumsol.planilla.dominio
sv.edu.ufg.matricula.datos
```

- Ya veremos que es una capa. Por ahora nos quedaremos hasta el paquete de la aplicación.

Paquetes

- Si dentro de una clase A, queremos usar otra clase B, hay dos posibilidades:
- Si las clases son del mismo paquete no hay problema.
- Si son de diferente paquete, la clase A debe tener una línea del estilo:


```
import paquetedeB.nombredeB;
```

 o bien


```
import paquetedeB.*;
```
- esta línea se encuentra debajo de la instrucción **package**.

Nombre de la clase

```
package nombrepquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- Recuerden que el nombre de la clase empieza por mayúscula y que las siguientes palabras del nombre comienzan por mayúscula.

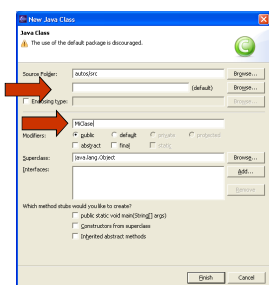
Nota: el nombre de la clase.

- A veces, nos referimos a una clase por su nombre. Es lo más común.
- A veces nos referimos por **nombrepquete.nombreClase**. Por ejemplo, el compilador suele dar los errores con esa nomenclatura.

Importante: cuando creamos una clase en Eclipse con File|New|Class

- Hay que poner el nombre de la clase y del paquete en la ventana que aparece.

- Lo otro puede cambiarse más fácilmente, pero el nombre de la clase y el paquete es mejor ponerlo bien desde el principio.



Declaración de atributos

```
package nombrepaquete;
public class NombreClase {
    Declaración de atributos
    Declaración de métodos
}
```

- Los atributos se declaran parecido a una declaración de variables pues no son más que un caso específico de variables.
- Se incluye el nombre, el tipo y (opcionalmente) un valor inicial.

Sintaxis de declaración de atributos

ámbito **tipo** **nombre** = valorinicial ;

- Los elementos en negro son opcionales.
- Es una declaración de variables y, cómo tal, tiene un nombre del atributo y un tipo.
- La única diferencia con una declaración de variables es que se puede poner un ámbito.

Modificadores de acceso (ámbito)

- **public**. El acceso al atributo es público. Se puede utilizar el atributo en cualquier circunstancia.
- **protected**. **No lo veremos por ahora.**
- **Sin modificador**. El atributo es accesible dentro del mismo paquete. Se dice que el atributo tiene ámbito "package".
- **private**. El atributo sólo puede ser usado dentro de la clase que lo define, lo que impide su uso desde otras clases.

Sintaxis de declaración de atributos

ámbito **tipo** **nombre** = valorinicial ;

- El tipo puede ser cualquier tipo en Java. Por tanto, también las clases.
- Esto permite que una clase tenga objetos de otras clases como atributos

Sintaxis de declaración de atributos

ámbito **tipo** **nombre** = valorinicial ;

- El nombre sigue las reglas de las variables: comienza en minúscula y las siguientes palabras comienzan en mayúscula.
- Como en cualquier declaración de variables, puede asignarse un valor inicial.

Sintaxis de declaración de atributos

ámbito **tipo** **nombre** = valorinicial ;

- Ejemplos de declaraciones válidas:

```
public String color;
public Auto auto;
private int velocidad = 0;
```


Sintaxis de declaración de métodos

ámbito *tipores* **nombre**(listaparam) {
 sentencias
}

- Los elementos en negro son opcionales
- **ámbito** es un modificador de acceso.
- **nombre** es el nombre del método. Misma política de mayúsculas y minúsculas que los atributos.
- **sentencias** el conjunto de sentencias que ejecutará el método. **Se le llama CUERPO del método.**

Sintaxis de declaración de métodos

ámbito *tipores* **nombre**(listaparam) {
 sentencias
}

- **listaparam** es la lista de parámetros en formato **tipo1 nombre1, tipo2 nombre2 ...** (es decir, como declaraciones de variables separadas por comas).
- **tipores** es el tipo del resultado que se devuelve. Si no devuelve nada, es **void**.

Modificadores de acceso (ámbito)

- **public**. La ejecución del método es pública. Se puede ejecutar el método en cualquier circunstancia.
- **protected**. **No lo veremos por ahora.**
- **Sin modificador**. El método puede ejecutarse dentro del mismo paquete (se dice que el método tiene ámbito "package").
- **private**. El método sólo puede ejecutarse dentro de la clase que lo define, lo que impide su ejecución desde otras clases.

Una práctica recomendable

- Es mejor que los atributos sean **private** (o **protected**) para proteger.
- El acceso a los atributos se hará mediante métodos **public** (o "package", es decir, sin modificador de acceso).
- Así, se controla el acceso a los atributos y se pueden incluir verificaciones sobre los datos que se guardan en ellos.

Por ejemplo, clase Cliente

- Modela un cliente de un banco.
- De un cliente, nos interesa su nombre y el crédito que le da el banco.
- Queremos iniciar un cliente (en este caso, el crédito será cero), fijar el crédito del cliente, obtener el nombre del cliente y obtener del crédito.

Por ejemplo, clase Cliente

```
package com.aurumsol.banco.dominio;
public class Cliente {
    private String nombre;
    private int credito;
    public void iniciar(String nombre){
        this.nombre = nombre;
        this.credito = 0;
    }
    public void setCredito(int credito){
        this.credito = credito;
    }
    public String getNombre(){
        return this.nombre;
    }
    public int getCredito(){
        return this.credito;
    }
}
```


Ejercicio

- Programen una clase que modela una libreta de banco, de la que nos interesa un número y un saldo. Podemos iniciar la libreta (con saldo 0), depositar una cantidad de dinero, retirar una cantidad de dinero, obtener el saldo y el número de la libreta.

Por ejemplo, clase Libreta (1)

```
package com.aurumsol.banco.dominio;
public class Libreta {
    private int numero;
    private int saldo;
    public void iniciar(int numero){
        this.numero = numero;
        this.saldo = 0;
    }
    public int getNumero(){
        return this.numero;
    }
    public int getSaldo(){
        return this.saldo;
    }
}
```

Por ejemplo, clase Libreta (2)

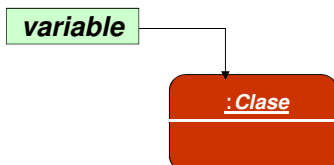
```
public void depositar(int monto){
    this.saldo += monto;
}
public boolean retirar(int monto){
    if (monto > this.saldo){
        return false;
    } else {
        this.saldo -= monto;
        return true;
    }
}
```

Uso de objetos (Creación)

- Crear un objeto (en dos pasos)
 - Declarar una variable de la clase del objeto.
Clase variable;
 - Crear un nuevo objeto y poner una referencia en la variable.
variable = new Clase();
- Crear un objeto (en un solo paso).
Clase variable = new Clase();

Nota: la variable no contiene un objeto sino una referencia a un objeto

Clase variable = new Clase();



- :Clase quiere decir en UML, un objeto de la clase "Clase"

Uso de objetos (Atributos)

- Recuperar el valor de un atributo
 - Desde otro objeto.
varObjeto.nombreAtributo
 - Desde el mismo objeto.
this.nombreAtributo
- Guardar un valor en un atributo.
 - Desde otro objeto.
varObjeto.nombreAtributo = expresion;
 - Desde el mismo objeto.
this.nombreAtributo = expresion;

Uso de objetos (Métodos)

- Ejecutar un método

- Desde otro objeto.

`varObjeto.nombreMetodo(param)`

- Desde el mismo objeto.

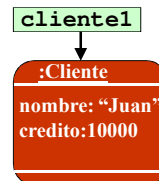
`this.nombreMetodo(param)`
`nombreMetodo(param)`

- **param** es una lista de parámetros separados por comas.

Ejemplo

```
Cliente cliente1 = new Cliente();
cliente1.iniciar("Juan");
cliente1.setCredito(10000);
```

- Si hiciéramos `cliente1.getCredito()` devuelve 10000.



Ejercicio

- Escribir un fragmento de código que cree una nueva libreta, deposite 1000 dólares y retire 200 dólares.

Solución

```
Libreta libreta1 = new Libreta();
libreta1.iniciar(123);
libreta1.depositar(1000);
boolean exito = libreta1.retirar(200);
```

- Si hiciéramos `libreta1.getSaldo()` devuelve 800.

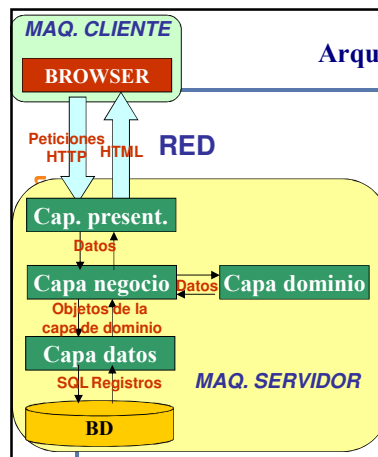


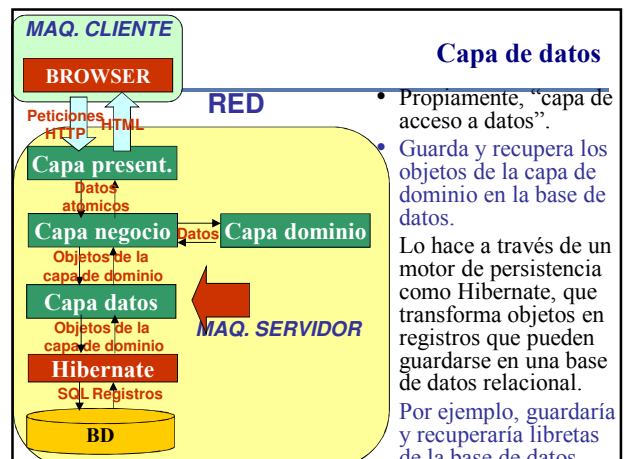
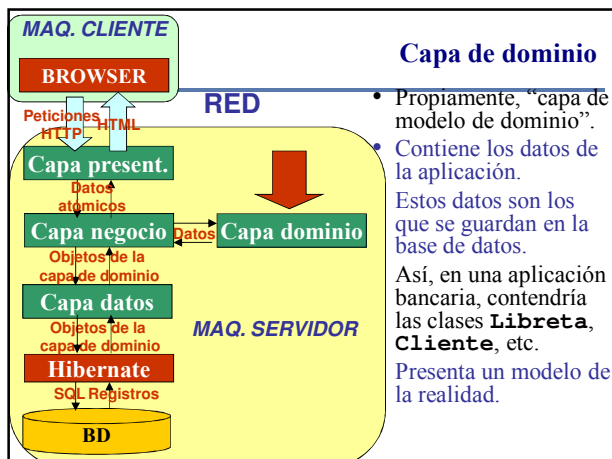
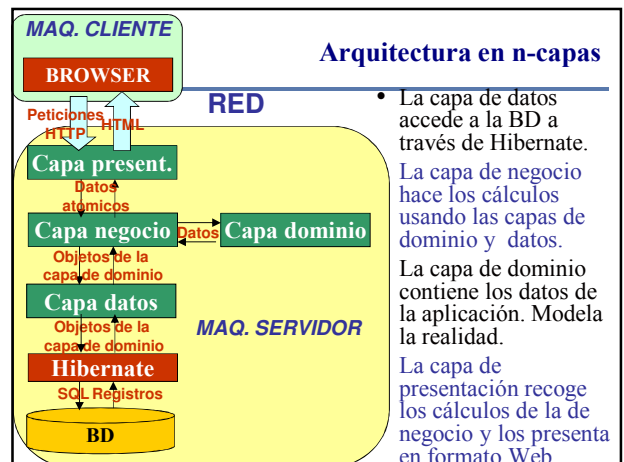
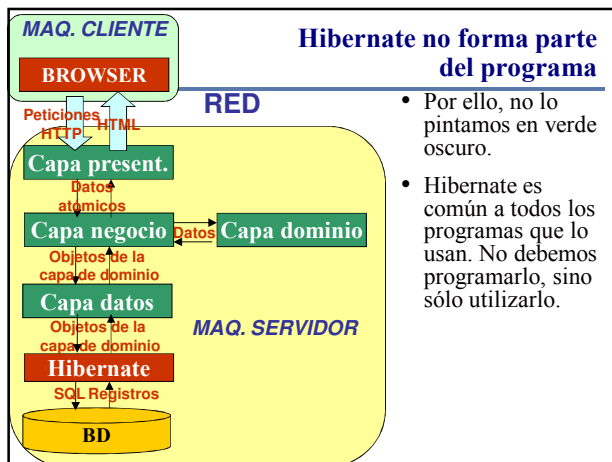
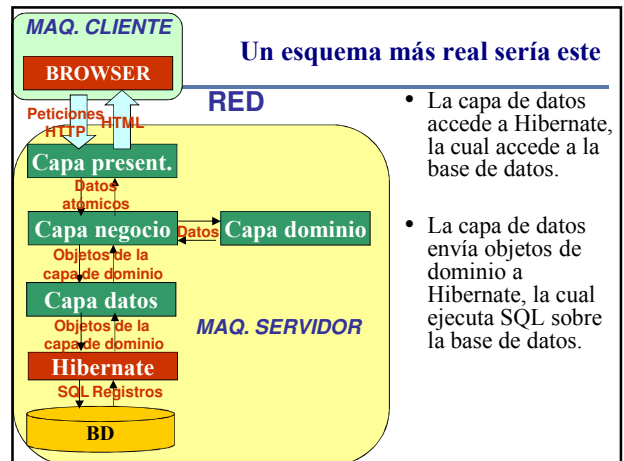
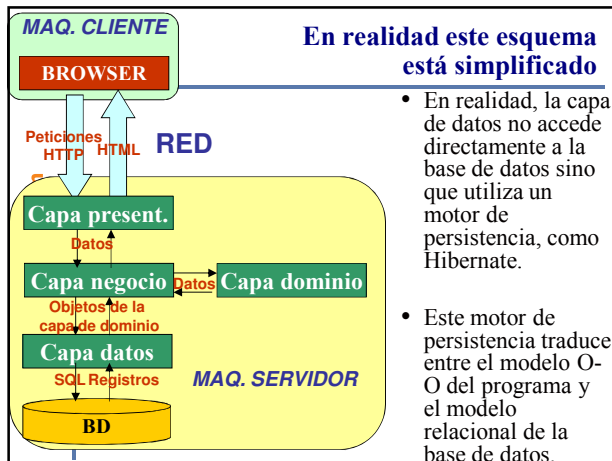
2. Repaso del curso anterior

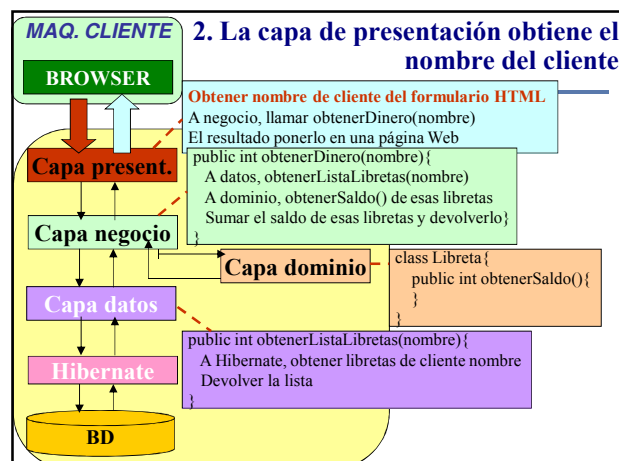
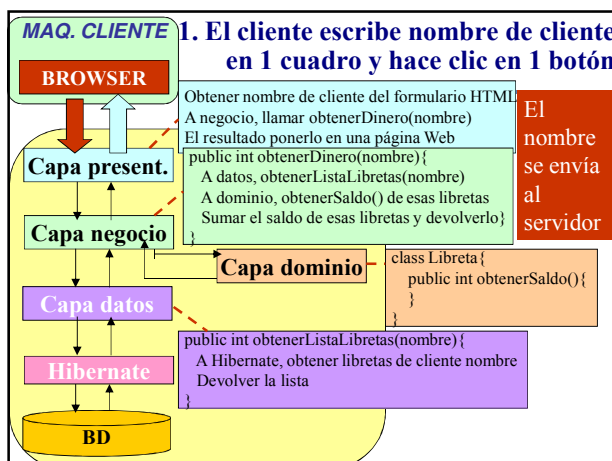
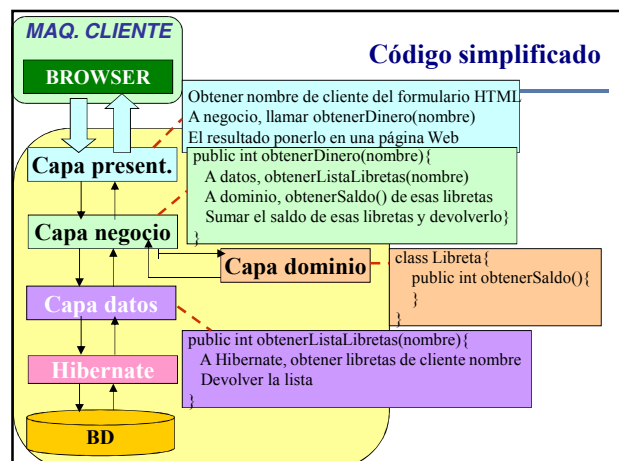
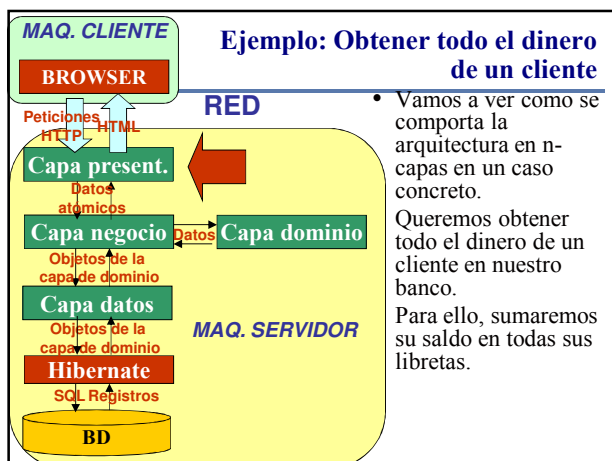
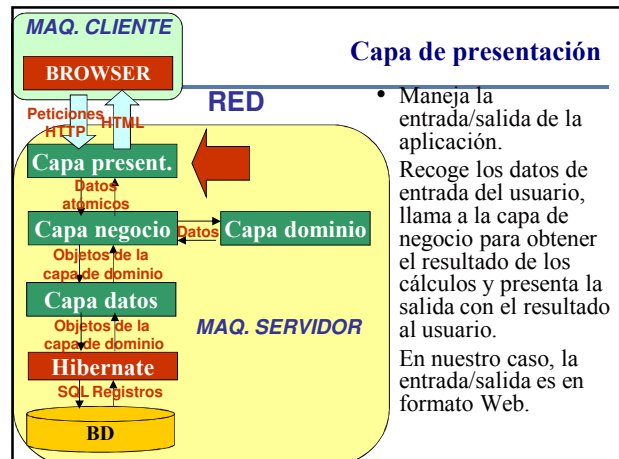
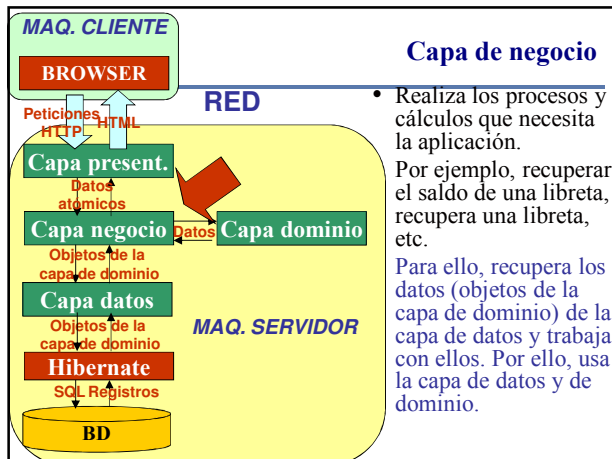
- 2.1. Programación orientada a objetos.
- 2.2. Arquitectura en n-capas.
- 2.3. Capa de dominio.
- 2.4. Capa de datos.
- 2.5. Capa de negocio.
- 2.6. Capa de presentación.

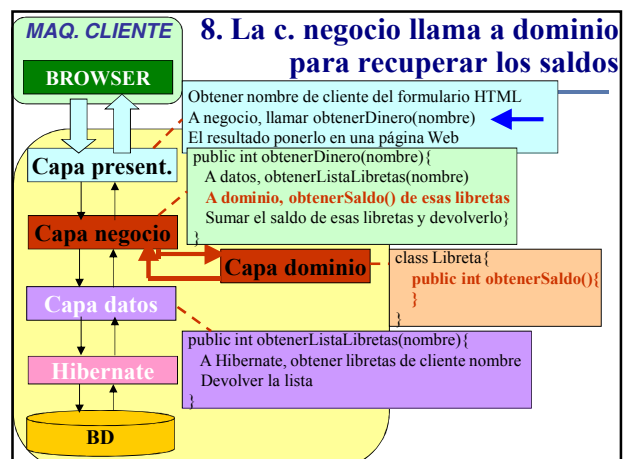
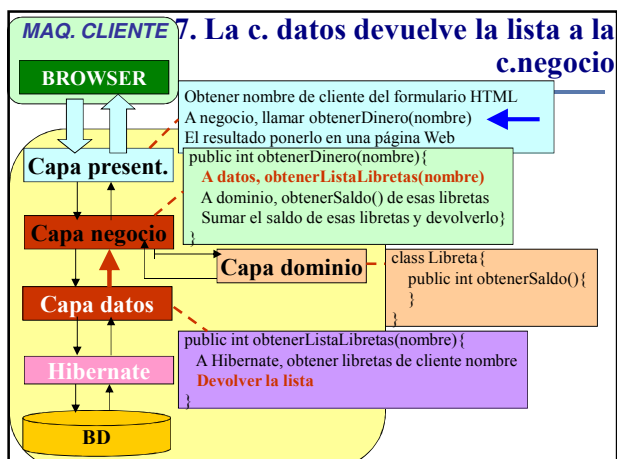
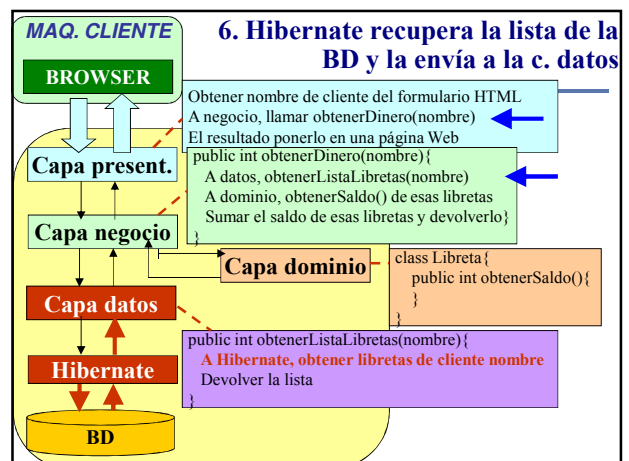
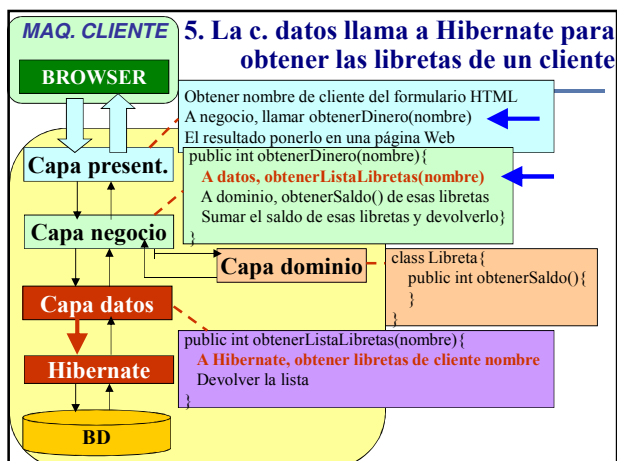
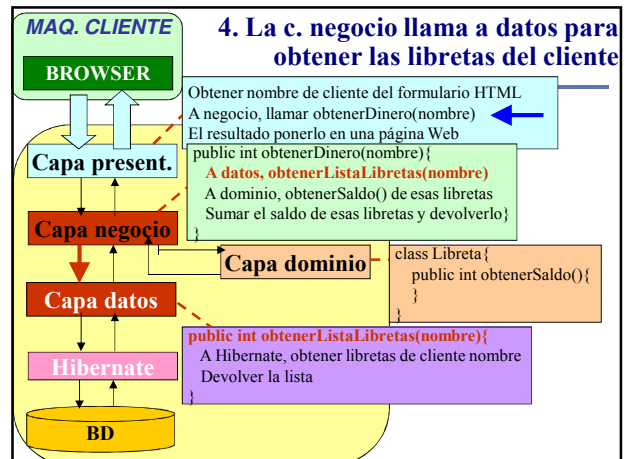
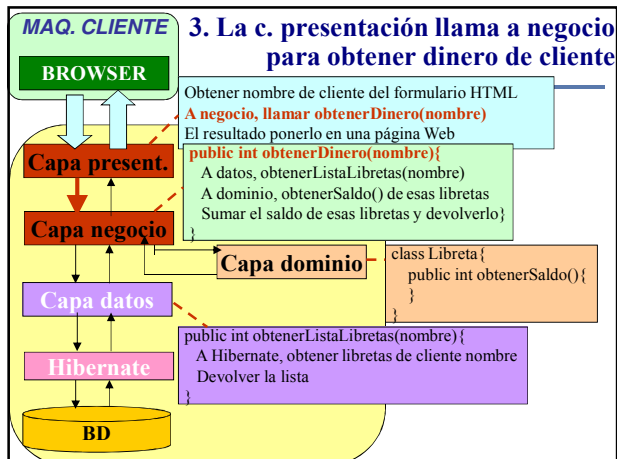
Arquitectura en n-capas

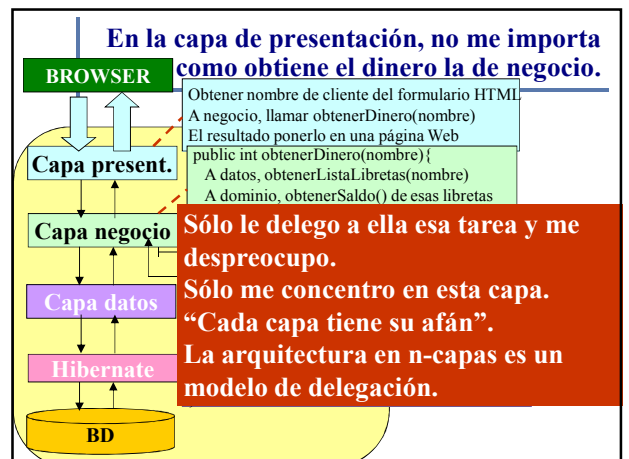
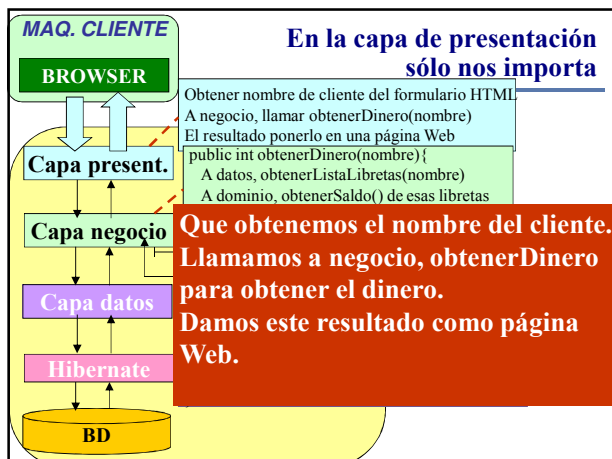
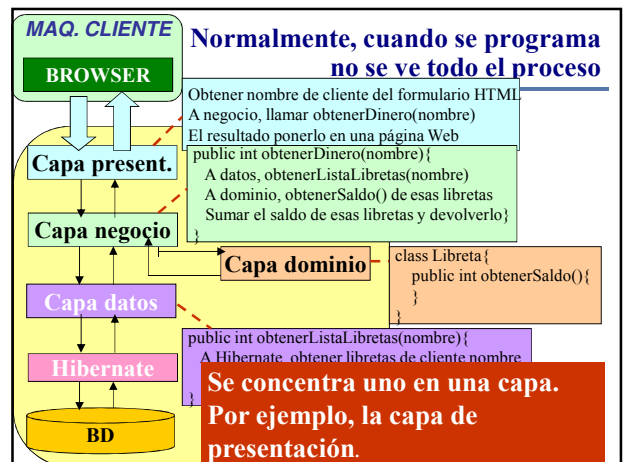
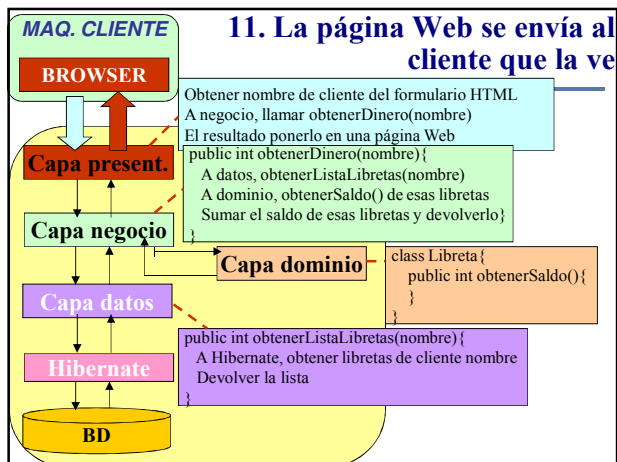
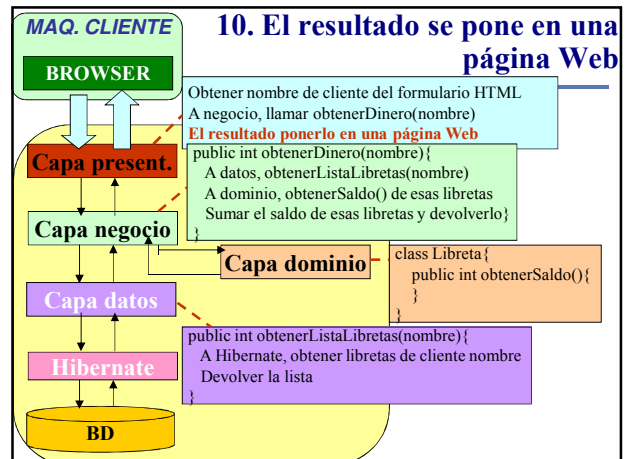
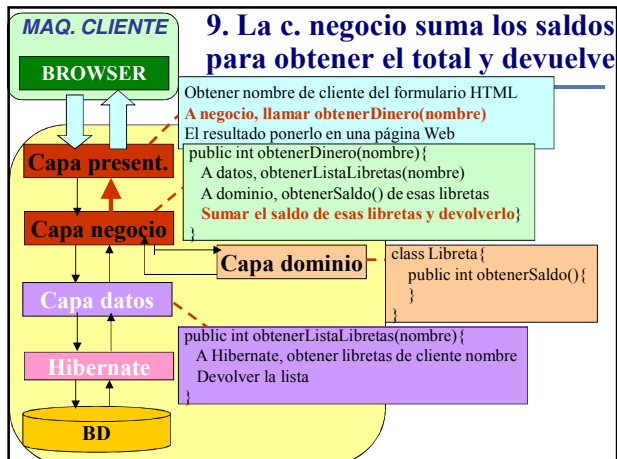
- Estándar actual de programación.
- Permite un código más sencillo, legible, fácil de mantener y cambiar.
- Permite una aplicación más flexible a los cambios.
- En su versión más simple O-O tiene cuatro capas.

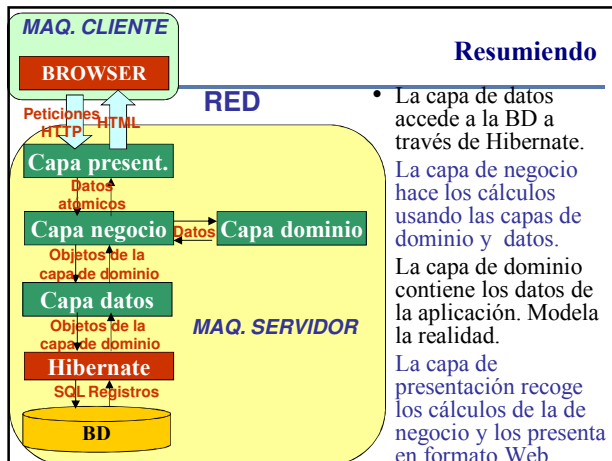












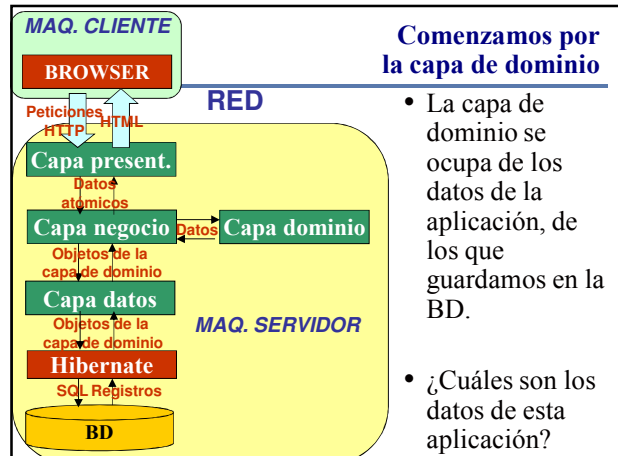
Vamos a hacer un ejemplo

- Queremos programar una aplicación Web de contabilidad de automóviles. La aplicación maneja autos. De cada auto, nos interesa: un nombre que lo identifica, el año de fabricación y su valor (en dólares), lo que será un **double**. Hay un método que nos permite el valor de un auto en un determinado año teniendo en cuenta la depreciación (los autos se deprecian un 5% a partir del año de fabricación).
- La aplicación Web nos permitirá crear un nuevo auto y obtener el valor de los autos de una determinada marca en un determinado año (contando la depreciación). Fijense que esto son dos páginas Web.

2. Repaso del curso anterior

- 2.1. Programación orientada a objetos.
- 2.2. Arquitectura en n-capas.
- 2.3. Capa de dominio.
- 2.4. Capa de datos.
- 2.5. Capa de negocio.
- 2.6. Capa de presentación.

Comenzamos por la capa de dominio



- La capa de dominio se ocupa de los datos de la aplicación, de los que guardamos en la BD.
- ¿Cuáles son los datos de esta aplicación?

Los datos de la aplicación son autos.

- Deberemos programar la clase **Auto**.
- Recordemos que de cada auto, nos interesa: un nombre que lo identifica, el año de fabricación y su valor (en dólares), lo que será un **double**. Hay un método que nos permite el valor de un auto en un determinado año teniendo en cuenta la depreciación (los autos se deprecian un 5% a partir del año de fabricación).

La capa de dominio (1)

```
package com.aurumsol.cursojava.auto.dominio;
public class Auto {
    private String nombre, marca;
    private int anyo, valor;
    public String getNombre(){
        return this.nombre;
    }
    public void setNombre(String nuevoNombre){
        this.nombre = nuevoNombre;
    }
    public String getMarca(){
        return this.marca;
    }
    public void setMarca(String nuevaMarca){
        this.marca = nuevaMarca;
    }
}
```


La capa de dominio (2)

```
public int getValor(){
    return this.valor;
}
public void setValor(int nuevoValor){
    this.valor = nuevoValor;
}
public int getAnyo(){
    return this.anyo;
}
public void setAnyo(int nuevoAnyo){
    this.anyo = nuevoAnyo;
}
```

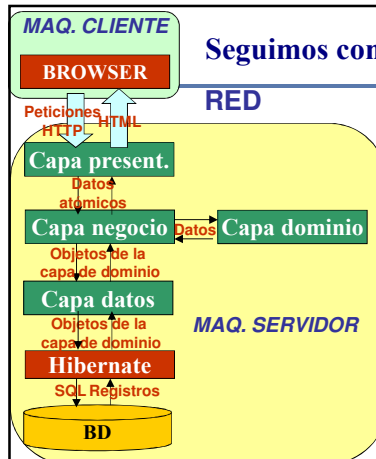
La capa de dominio (3)

```
public double getValorAmortizadoEnAnyo
(int anyoCalc){
    double valorAmortizado;
    if (anyoCalc < this.anyo){
        valorAmortizado = 0.0;
    } else {
        valorAmortizado =
            this.valor-0.05*this.valor *
            (anyoCalc - this.anyo);
        if (valorAmortizado < 0.0){
            valorAmortizado = 0.0;
        }
    }
    return valorAmortizado;
}
}
```

2. Repaso del curso anterior

- 2.1. Programación orientada a objetos.
- 2.2. Arquitectura en n-capas.
- 2.3. Capa de dominio.
- 2.4. Capa de datos.
- 2.5. Capa de negocio.
- 2.6. Capa de presentación.

Seguimos con la capa de datos



- La capa de datos se encarga de guardar y recuperar objetos de dominio de la base de datos.
- En este caso, se encargará de guardar y recuperar objetos **Auto** de la base de datos.

El patrón DAO dice

- Que todos los métodos para acceder a objetos de la misma clase de dominio en la base de datos, deben agruparse en una misma clase de datos (llamada **clase DAO o DAO**).
- En nuestro caso, tendremos que todos los métodos para actualizar y recuperar autos en la base de datos deben ir en una única clase, que llamaremos **DAOAuto**.

Esqueleto de la clase DAOAuto

```
package com.aurumsol.cursojava.auto.datos;
import com.aurumsol.cursojava.auto.dominio.*;
public class DAOAuto {
    /* Método que mira si ya existe un auto en la
    BD con el nombre que se pasa como parámetro */
    public boolean existeNombre(String nombre){
    }
    /* Método que guarda un auto en la BD*/
    public void guardarNuevo(Auto auto){
    }
    /* Método que obtiene los autos de un año*/
    public List obtenerAutosMarca(String marca){
    }
}
```


¿Por qué estos métodos y no otros?

- Estos métodos son los que necesita la capa de negocio.
- Realmente, en una aplicación real se diseñan al mismo tiempo la capa de negocio y la interfaz de la capa de datos.
- Aquí no lo hacemos así porque se trata de un repaso.

Las ventajas de la encapsulación

- Fijémonos que estos métodos tienen parámetros de tipo Auto. Por ejemplo,

```
public void guardarNuevo(Auto auto) {
}
```
- Esto contrasta con la versión no orientada a objetos

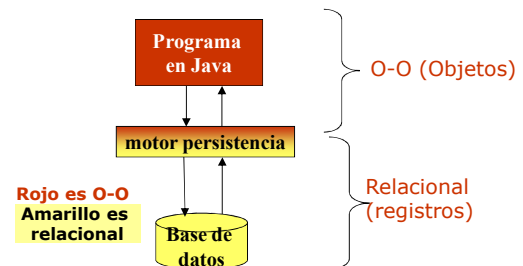
```
public void guardarNuevo(String nombre,
String marca, int anyo, int valor) {
}
```
- En este caso, Deberíamos cambiar multitud de métodos en diferentes sitios si cambiamos el auto.
- En la versión O-O, si cambiamos los datos de un auto, sólo deberemos cambiar la clase Auto, pero todos los métodos que la utilizan (como guardarNuevo) no cambian.
- El cambio está localizado y se facilita el mantenimiento.

¿Cómo implementaremos estos métodos?

- Utilizaremos el motor de persistencia Hibernate.
- Con él accederemos a la base de datos.

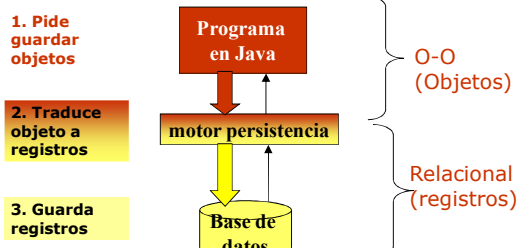
Motor de persistencia (Hibernate y otros)

- Hace la traducción entre el programa O-O y la BD relacional (y viceversa).



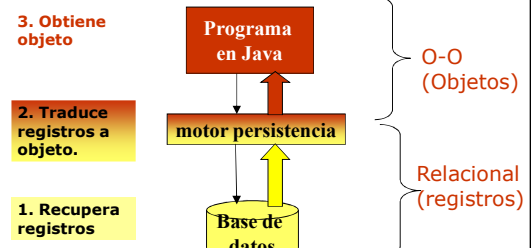
Guardando un objeto

- El programa sólo ve que pide guardar un objeto y el objeto se guarda.



Recuperando un objeto

- El programa sólo ve que pide recuperar un objeto y el objeto se recupera.



Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Adaptar la clase persistente a Hibernate

- La clase persistente es la clase de la capa de dominio que vamos a guardar en la BD.
- Son las de la capa de dominio.
- En nuestro caso, es la clase **Auto**.
- Vamos a adaptarla para que pueda trabajar con Hibernate.

¿Es necesario adaptar la clase persistente para Hibernate?

- En realidad, no.
- Cualquier clase que tengamos se puede tratar con Hibernate sin hacer ningún cambio.
- Los cambios que veremos aquí son **recomendados, no obligatorios**.

Una adaptación conveniente

- La adaptación que vamos a explicar tampoco es obligatoria para lo que haremos aunque sí para algunas funciones más avanzadas de Hibernate.
- Es una adaptación ampliamente seguida por los programadores que usan Hibernate.

La adaptación se trata

- De crear un atributo para almacenar la clave primaria de la tabla. Lo llamaremos “atributo identidad”. Normalmente, se llamará “**id**”.
- Este atributo deberá ser privado.

Auto queda así (1)

```
package com.aurumsol.cursojava.auto.dominio;
public class Auto {
    private int id; //Campo identidad
    private String nombre, marca;
    private int anyo, valor;
    public String getNombre(){
        return this.nombre;
    }
    public void setNombre(String nuevoNombre){
        this.nombre = nuevoNombre;
    }
    public String getMarca(){
        return this.marca;
    }
    public void setMarca(String nuevaMarca){
        this.marca = nuevaMarca;
    }
}
```

Auto queda así (2)

```
public int getValor(){
    return this.valor;
}
public void setValor(int nuevoValor){
    this.valor = nuevoValor;
}
public int getAnyo(){
    return this.anyo;
}
public void setAnyo(int nuevoAnyo){
    this.anyo = nuevoAnyo;
}
```

Auto queda así (3)

```
public double getValorAmortizadoEnAnyo
(int anyoCalc){
    double valorAmortizado;
    if (anyoCalc < this.anyo){
        valorAmortizado = 0.0;
    } else {
        valorAmortizado =
            this.valor-0.05*this.valor *
            (anyoCalc - this.anyo);
        if (valorAmortizado < 0.0){
            valorAmortizado = 0.0;
        }
    }
    return valorAmortizado;
}
```

Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Se necesitan tres tipos de archivos de configuración

- El archivo **context.xml** que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
 - Sólo hay uno.
 - Su nombre es **hibernate.cfg.xml**
- Los archivos de correspondencia que indican las clases que se van a persistir
 - Hay uno por cada clase persistente.
 - Su nombre es **NombreClase.hbm.xml**

Se necesitan tres tipos de archivos de configuración

- El archivo **context.xml** que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
 - Sólo hay uno.
 - Su nombre es **hibernate.cfg.xml**
- Los archivos de correspondencia que indican las clases que se van a persistir
 - Hay uno por cada clase persistente.
 - Su nombre es **NombreClase.hbm.xml**

Creemos context.xml

```
<Context path="/" docBase="ROOT">
<Resource name="jdbc/nombreJNDI"
scope="Shareable"
type="javax.sql.DataSource"
factory="org.apache.tomcat.dbcp.dbcp.
BasicDataSourceFactory"
url="URLdeLaBD"
driverClassName="ClaseDelDriver"
username="usernameBD"
password="passwordBD" maxWait="3000"
maxIdle="100" maxActive="10">
</Resource>
</Context>
```

- El archivo se colocará en **webapps\ROOT\META-INF** del directorio de instalación de Tomcat.

En el anterior archivo

- nombreJNDI**: Es un nombre arbitrario pero es bueno que coincida con el nombre de la BD.
- URLdeLaBD**: URL para acceder a la BD. En MySQL es **jdbc:mysql://localhost/nombre**, donde **nombre** es el nombre de la BD.
- ClaseDelDriver**: es la clase del driver JDBC. En MySQL es **com.mysql.jdbc.Driver**.
- usernameBD** y **passwordBD**: son el username y el password para acceder a la BD.

Por ejemplo, para la base de datos "prueba" puede ser así

```
<Context path="/" docBase="ROOT">
<Resource name="jdbc/prueba"
scope="Shareable"
type="javax.sql.DataSource"
factory="org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory"
url="jdbc:mysql://localhost/prueba"
driverClassName="com.mysql.jdbc.Driver"
username="root" password="admin"
maxWait="3000" maxIdle="100"
maxActive="10">
</Resource>
</Context>
```

- El archivo se colocará en **webapps\ROOT\META-INF** del directorio de instalación de Tomcat.

Se necesitan tres tipos de archivos de configuración

- El archivo **context.xml** que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
 - Sólo hay uno.
 - Su nombre es **hibernate.cfg.xml**
- Los archivos de correspondencia que indican las clases que se van a persistir
 - Hay uno por cada clase persistente.
 - Su nombre es **NombreClase.hbm.xml**

hibernate.cfg.xml (colocar en el directorio de clases)

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.datasource">
java:comp/env/jdbc/nombreJNDI </property>
<property name="show_sql">false</property>
<property name="dialect"> dialecto</property>
archivos de correspondencia de cada clase persist.
</session-factory>
</hibernate-configuration>
```

En el archivo anterior

- nombreJNDI**: Es el nombre que hemos puesto en el archivo de configuración XML que hay en **conf\Catalina\localhost** (mirar antes).
- dialecto**: Dialecto del SQL de la BD. Se encuentra en la documentación de Hibernate. Para MySQL es **org.hibernate.dialect.MySQLDialect**
- archivos de correspondencia**: son un conjunto de líneas

```
<mapping resource="nombre1" />
...
<mapping resource="nombreN" />
```

donde **nombre1**,..., **nombreN** son los nombres de los archivos de correspondencia. Hay un archivo por cada clase persistente.

Por ejemplo, para nuestro caso

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.datasource">
            java:comp/env/jdbc/prueba</property>
        <property name="show_sql">false</property>
        <property name="dialect">
            org.hibernate.dialect.MySQLDialect</property>
        <mapping resource="Auto.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Se necesitan tres tipos de archivos de configuración

- El archivo **context.xml** que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
 - Sólo hay uno.
 - Su nombre es **hibernate.cfg.xml**
- Los archivos de correspondencia que indican las clases que se van a persistir
 - Hay uno por cada clase persistente.
 - Su nombre es **NombreClase.hbm.xml**

Archivo de correspondencia para una clase persistente

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD
    3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="clase" table="tabla">
        atributos
    </class>
</hibernate-mapping>
```

En el archivo anterior

- **clase**: Clase que queremos persistir con el formato **nombrepaquete.nombreClase**
- **tabla**: Nombre de la tabla de la base de datos (podemos poner cualquier nombre pero después hemos de usar ese nombre en la base de datos).
- **atributos**: especificación de atributos de la clase.

Para cada atributo

- Si es un atributo diferente del "atributo identidad":

```
<property name="nombreAtributo"
    access="field"/>
```

- Donde **nombreAtributo** es el nombre del atributo que coincidirá con el nombre del campo de la tabla de la BD donde se guarda.
- También puede ser que sean nombres diferentes, pero la sintaxis en este caso no la veremos por ahora.

Si es el atributo identidad

```
<id name="nombreAtributo" type="int"
    access="field">
    <column name="nombreCampo"/>
    <generator class="identity"/>
</id>
```

- **nombreAtributo** es el nombre del atributo identidad.
- **nombreCampo** es el nombre del campo en que se guarda este atributo. Es decir, la clave primaria.

En nuestro caso, Auto.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/
hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class
name="com.aurumsol.cursojava.auto.dominio.Auto"
table="auto">
<id name="id" type="int" access="field">
<column name="id"/>
<generator class="identity"/>
</id>
<property name="nombre" access="field"/>
<property name="marca" access="field"/>
<property name="valor" access="field"/>
<property name="anyo" access="field"/>
</class></hibernate-mapping>
```

¿Cuándo se cambian estos archivos?

- El archivo que indica la conexión de la base de datos (**context.xml**).

Si cambia la BD, el contexto, clave o contraseña

- El archivo donde se especifican la configuración de Hibernate (**hibernate.cfg.xml**).

Si cambia la BD o el conjunto de clases persistentes

- Los archivos de correspondencia que indican las clases que se van a persistir

Cada vez que cambian las clases persistentes

Pero la base de datos cambia con poca frecuencia

- Normalmente lo que cambia es:
 - Los archivos de correspondencia.
 - La lista de clases persistentes en **hibernate.cfg.xml**.
- Aún esto, puede generarse automáticamente con herramientas como Xdoclet o Middlegen o plugins de Eclipse para Hibernate.

¿Dónde se despliegan estos archivos?

- El archivo que indica la conexión de la base de datos (**context.xml**).

webapps\ROOT\META-INF

- El archivo donde se especifican la configuración de Hibernate (**hibernate.cfg.xml**)

webapps\ROOT\WEB-INF\classes

- Los archivos de correspondencia que indican las clases que se van a persistir

webapps\ROOT\WEB-INF\classes

Programar en Hibernate

1. Adaptar la clase persistente a Hibernate.
2. Escribir los archivos de configuración.
3. Crear el esquema de la BD.
4. Programar el acceso a la BD.
5. Desplegar.

Primero, creamos la base de datos que contendrá las clases que vamos a guardar

- Hibernate tiene herramientas para hacerlo automáticamente a partir de los archivos de correspondencia (e incluso para crear las clases persistentes a partir de los archivos de correspondencia).
- Pero nosotros lo haremos a mano.

Creemos una base de datos “prueba”

- Creemos una tabla llamada “auto”.
- Allí guardaremos nuestros autos.

¿Qué campos debe tener una tabla que guarda una clase?

- Debe tener un campo para guardar cada atributo de la clase.
- El campo para el atributo “identidad” debe ser de tipo “autoincremento” (llamado en algunos SGBDs, “identity”).

En nuestro caso, ¿qué campos debe tener la tabla que guarda Auto?

- Si la clase **Auto** tiene estos atributos:

```
private int id; //Campo identidad
private String nombre, marca;
private int anyo, valor;
```

- Deberemos tener dos campos enteros: **anyo** y **valor**.
- Además dos campos de carácter: **nombre** y **marca**.
- Además, tendremos un campo autoincremento que será clave primaria. Lo llamaremos “**id**”.

En resumen, en la tabla “auto” tenemos

id	integer NOT NULL AUTO_INCREMENT
anyo	integer NOT NULL
valor	integer NOT NULL
nombre	varchar (20) NOT NULL
marca	varchar (40) NOT NULL

Nota: En MySQL es bueno que definir las tablas como InnoDB

Creando la tabla.

- Desde MySQL Query Browser

Column Name	Datatype	PK	FK	Flags	Default Value	Comment
id	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/> UNSIGNED <input type="checkbox"/> ZEROFILL		
nombre	VARCHAR(20)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> BINARY		
marca	VARCHAR(40)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> BINARY		
anyo	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/> UNSIGNED <input type="checkbox"/> ZEROFILL	0	
valor	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/> UNSIGNED <input type="checkbox"/> ZEROFILL	0	

- Otras formas:
 - **SchemaExport** y **SchemaUpdate** crean automáticamente las instrucciones como CREATE TABLE a partir de los archivos de configuración.
 - Hay plugins de Eclipse que van más allá y crean automáticamente la tabla a partir de los archivos de configuración.

Nota

- Fíjense que **el nombre de los campos** de la base de datos **es exactamente el mismo que el nombre de los atributos** de la clase.
- En realidad, no tiene porque ser así, pero es lo más sencillo y la opción por defecto.
- Es la opción que usaremos aquí.

Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Conceptos básicos de Hibernate

- **Session** (Sesión). Una conexión de corta duración a la base de datos. Normalmente, cuando queremos hacer una operación en la base de datos.
 - Obtenemos una sesión.
 - Hacemos la operación.
 - Cerramos la sesión lo más pronto posible.
- **SessionFactory** (fábrica de sesiones). Una clase que nos permite obtener sesiones.

Esquema de programación en Hibernate

- 1. Obtener una **SessionFactory**.
- 2. Obtener una **Session**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Esquema de programación en Hibernate

- 1. Obtener una **SessionFactory**.
- 2. Obtener una **Session**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Obtener una SessionFactory

- Hay varias formas. La que usaremos es:
- ```
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
```

- Así se creará una SessionFactory con las opciones de configuración que hay en el archivo **hibernate.cfg.xml** (lo veremos después)
- Nota: Para usar estos objetos debemos:

```
import org.hibernate.*
import org.hibernate.cfg.*
```

### Sólo se debería obtener una única SessionFactory para todo el programa

```
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
```

- Sin embargo, por ahora no sabemos cómo hacer esto. Requiere de un patrón llamado “Singleton” que no hemos visto.
- Por eso, obtendremos una cada vez que accedamos a la BD.
- Esto es más lento, pero, por ahora servirá.



### Esquema de programación en Hibernate

- 1. Obtener una `SessionFactory`.
- 2. Obtener una `Session`.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

### Obteniendo una Session

```
Session session = sessionFactory.openSession();
```

- Hay que obtener una sesión cada vez que accedemos a la BD.
- Una sesión debería durar poco.
- Normalmente, usaremos una sesión para cada método de la clase DAO (más adelante, se verán mejores formas).
  - La abriremos al principio del método.
  - La cerraremos al final del método.

### Esquema de programación en Hibernate

- 1. Obtener una `SessionFactory`.
- 2. Obtener una `Session`.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

### Operaciones sobre la base de datos

- Guardar un objeto nuevo  
`session.save (objeto)`
- Actualizar un objeto que ya existía.  
`session.update (objeto)`
- Borrar un objeto.  
`session.delete (objeto)`

### Operaciones sobre la base de datos

- Recuperar objetos de la base de datos.

```
List lista = session.createQuery (condicion).list();
```

- Recupera todos los objetos de la base de datos que cumplen la condición y los guarda en una lista. Podría hacerse en dos pasos.

```
Query consulta = session.createQuery (condicion);
List lista = consulta.list();
```

- `Query` es una clase para consultas a la base de datos.

### Las listas

- Objetos de tipo `List`. Métodos:
- Si `l` es de tipo lista
  - `l.isEmpty()` Dice si la lista está vacía.
  - `(Clase) l.get (p)` Obtiene el objeto que está en la posición `p` (comienzan de 0) y con la clase `Clase`
  - `l.size()` Obtiene el número de elementos de la lista.
- La clase `List` pertenece al paquete `java.util`, por eso se debe importar.



### Las condiciones

```
List lista =
 sesion.createQuery(condicion).list();
```

- La condición es un **String** que expresa qué objetos queremos recuperar de la base de datos.
- Está en un lenguaje parecido al SQL llamado HQL (**H**ibernate **Q**uery **L**anguage)

### Una condición en HQL

```
from Clase as objeto where condiciones
```

Ejemplos

```
from Libreta as libreta where
 libreta.saldo>1000
```

```
from Libreta as libreta where
 libreta.numero=123
```

### Por ejemplo

```
List lista1 =
 sesion.createQuery("from Libreta as
 libreta where
 libreta.saldo>1000").list();
```

- En la `lista1` se almacenará una lista con todas las libretas con saldo superior a 1000.

### Esquema de programación en Hibernate

- 1. Obtener una **SessionFactory**.
- 2. Obtener una **Session**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

### Guardar los cambios de la sesión (a veces)

- Si hemos grabado, actualizado o borrado.
- No si sólo hemos consultado.

- Fácil, se hace:

```
sesion.flush();
```

### Esquema de programación en Hibernate

- 1. Obtener una **SessionFactory**.
- 2. Obtener una **Session**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.



### Cerrar sesión

- Esto se consigue:

```
sesion.close();
```

### Ejemplo: guardar un nuevo objeto (creado con *objeto* = new *Clase()*)

```
//1. Obtener SessionFactory
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
//2. Obtener Session
Session session = sessionFactory.openSession();
//3. Hacer operaciones
session.save(objeto);
//4. Guardar cambios
session.flush();
//5. Cerrar la sesión
session.close();
```

### Ejemplo: recuperar los autos de año de fabricación menor que 1980

```
//1. Obtener SessionFactory
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
//2. Obtener Session
Session session = sessionFactory.openSession();
//3. Hacer operaciones
List autosViejos = session.createQuery("from Auto
as auto where auto.anyo<1980").list();
//4. No se guardan cambios porque no los hay
//5. Cerrar la sesión
session.close();
```

### Programemos el DAO

```
package com.aurumsol.cursojava.auto.datos;
import com.aurumsol.cursojava.auto.dominio.*;
import java.util.List;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class DAOAuto {
 public boolean existeNombre(String nombre){
 }
 public void guardarNuevo(Auto auto){
 }
 public List obtenerAutosMarca(String marca){
 }
}
```

- Por motivos de espacio pondremos un método por cada transparencia.

### El método que mira si hay un auto con un nombre

```
public boolean existeNombre(String nombre){
 SessionFactory sessionFactory = new
 Configuration().configure().
 buildSessionFactory();
 Session session = sessionFactory.openSession();
 List autos = session.createQuery("from Auto as
 auto where auto.nombre = '"+nombre+"'").list();
 session.close();
 return !(autos.isEmpty());
}
```

### El método que graba un auto

```
public void guardarNuevo(Auto auto){
 SessionFactory sessionFactory = new
 Configuration().configure().
 buildSessionFactory();
 Session session = sessionFactory.openSession();
 session.save(auto);
 session.flush();
 session.close();
}
```



### El método que recupera los autos de una determinada marca

```
public List obtenerAutosMarca(String marca){
 SessionFactory sessionFactory = new
 Configuration().configure().
 buildSessionFactory();
 Session session = sessionFactory.openSession();
 List autos = session.createQuery("from Auto as
 auto where auto.marca = '"+marca+"'").list();
 session.close();
 return autos;
}
```

### Programar en Hibernate

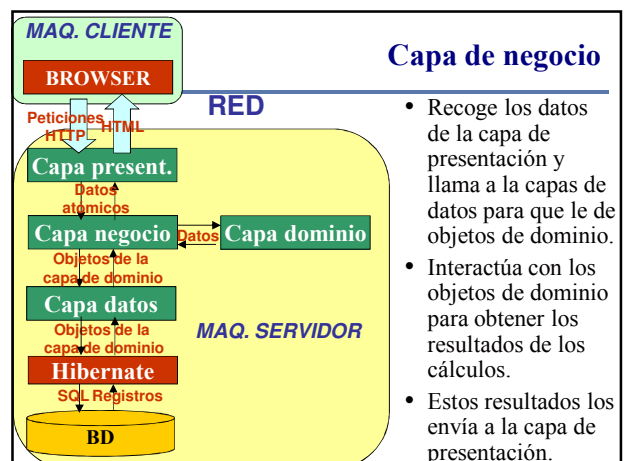
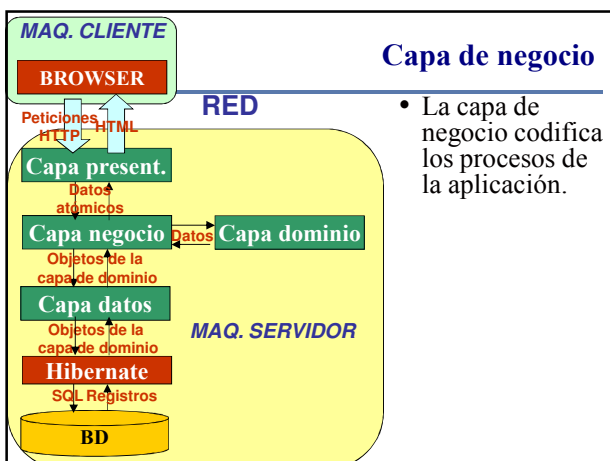
- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

### Desplegar

- Desplegamos la JSP en **webapps\ROOT**
- Desplegamos todas las clases como archivos JAR en **webapps\ROOT\WEB-INF\lib**.
- Colocamos todos los archivos de configuración donde se han dicho.
- Nos aseguramos que el servidor de la BD está funcionando.
- Ejecutamos.

### 2. Repaso del curso anterior

- 2.1. Programación orientada a objetos.
- 2.2. Arquitectura en n-capas.
- 2.3. Capa de dominio.
- 2.4. Capa de datos.
- 2.5. Capa de negocio.
- 2.6. Capa de presentación.





### Capa de negocio

- La capa de negocio implementa los procesos de la aplicación.
- Así, hay un método por cada proceso de datos que queremos que realice la aplicación. (Considerando como proceso lo que hay entre una entrada de datos y la salida correspondiente).
- Aquí hay dos procesos: crear un nuevo auto y obtener el valor de los autos de una determinada marca en un determinado año (contando la depreciación).

### La capa de negocio queda así

```
package
com.aurumsol.cursojava.auto.negocio;
import com.aurumsol.cursojava.auto.datos.*;
import
com.aurumsol.cursojava.auto.dominio.*;
import java.util.List;
public class NgcAuto {
```

### La capa de negocio queda así

```
public boolean crearNuevo(String nombre, String
marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 Auto nuevoAuto = new Auto();
 nuevoAuto.setNombre(nombre);
 nuevoAuto.setAnyo(anyoFabricacion);
 nuevoAuto.setMarca(marca);
 nuevoAuto.setValor(valor);
 daoAuto.guardarNuevo(nuevoAuto);
 return true;
 }
}
```

### La capa de negocio queda así

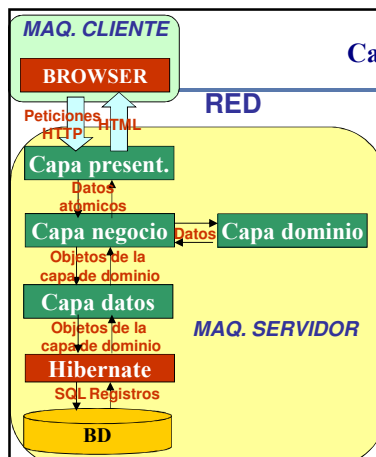
```
public int obtieneValorAutosMarcaAnyo (String
marca, int anyo){
 DAOAuto daoAuto = new DAOAuto();
 List autos =
 daoAuto.obtenerAutosMarca(marca);
 int numeroAutos = autos.size();
 int valorAnyo = 0;
 Auto autoActual;
 for (int i=0; i<numeroAutos;i++){
 autoActual = (Auto)autos.get(i);
 valorAnyo +=
 autoActual.getValorAmortizadoEnAnyo(anyo);
 }
 return valorAnyo;
}
```

## 2. Repaso del curso anterior

- 2.1. Programación orientada a objetos.
- 2.2. Arquitectura en n-capas.
- 2.3. Capa de dominio.
- 2.4. Capa de datos.
- 2.5. Capa de negocio.
- 2.6. Capa de presentación.

### Capa de presentación

- Es la que se ocupa de la entrada/salida.
- Recoge los parámetros de entrada.
- Llama a la capa de negocio para delegar los cálculos en ella.
- Después produce una salida en forma de HTML.





### Capa de presentación

- Formada por dos tipos de archivos.
- Los archivos HTML recogen los datos en forma de formulario.
- Los archivos JSP llaman a la capa de negocio, les envían los datos, recogen el resultado y crean la salida.

### Estructura de los documentos HTML y JSP

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.0 Transitional//EN">
<HTML>
 <HEAD>
 <TITLE>
 Título de la página
 </TITLE>
 </HEAD>
 <BODY>
 Diseño de la página
 </BODY>
</HTML>
```

### Aquí tenemos dos documentos HTML

- Uno para crear un auto nuevo y otro para obtener los valores de los autos de una marca en un año.
- Comenzamos con el de crear un auto nuevo.

#### Crear un nuevo auto

Escriba el nombre:

Escriba la marca:

Escriba el año de fabricación:

Escriba el valor:

### HTML para crear un auto nuevo

```
<html><head><title>Crear un nuevo
auto</title></head>
<body><H1>Crear un nuevo auto</H1>
<FORM ACTION="http://localhost/nuevoauto.jsp">
Escriba el nombre:
<INPUT TYPE="TEXT" NAME="nombre">

Escriba la marca:
<INPUT TYPE="TEXT" NAME="marca">

Escriba el año de fabricación:
<INPUT TYPE="TEXT" NAME="ano">

Escriba el valor:
<INPUT TYPE="TEXT" NAME="valor">

<INPUT TYPE="SUBMIT" VALUE="Crear nuevo">
</FORM></body></html>
```

### HTML para obtener la lista de autos de una marca en un año

```
<html><head><title>Valor autos</title></head>
<body>
<H1>Calcula el valor de los autos de una marca
en un año</H1>
<FORM ACTION="http://localhost/valorautos.jsp">
Escriba la marca:
<INPUT TYPE="TEXT" NAME="marca">

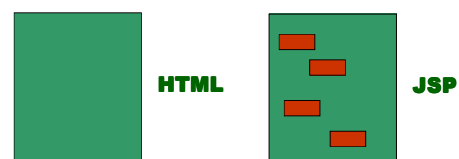
Escriba el año de fabricación:
<INPUT TYPE="TEXT" NAME="ano">

<INPUT TYPE="SUBMIT" VALUE="Calcular valor">
</FORM>
</body>
</html>
```

Como ven, los dos tienen en ACTION, la JSP que se encargará de procesar los datos del formulario.

### La estructura de una página JSP es igual que la de una página HTML

- La página HTML sólo tiene código HTML.
- La página JSP es una página HTML con fragmentos ("trocitos") de Java dentro.
- Si el código HTML es verde y el código Java es rojo, esto se podría representar así:





### Los trocitos se llaman “Elementos de guión” (“Scripting elements”)

- Los elementos de guión son construcciones sintácticas que permiten insertar código Java en una JSP.
- Hay de tres clases (las veremos más adelante):
  1. Expresiones JSP.
  2. Scriptlets.
  3. Declaraciones JSP.

### Elementos de guión

- **1. Expresiones JSP.** Son expresiones en Java (devuelven un valor). Esta expresión se integra en el código HTML. Van entre `<%=` y `%>`
- **2. Scriptlets.** Instrucciones Java que se ejecutan en medio de la generación del HTML. Van entre `<%` y `%>`

### Algunas expresiones usadas en las JSPs

- `request.getParameter("cuadro")` devuelve el contenido del cuadro de texto llamado cuadro en el HTML.
- `Integer.parseInt(cadena)`, convierte una cadena en entero.
- `out.print(expresion)`. Evalúa una expresión e incluye su valor en el HTML que genera la JSP. Es equivalente a poner una expresión JSP del estilo `<%= expresion %>`
- `<%@ page import = "nombrepaquete.*"%>` Cuando usamos las clases de un paquete en una JSP, debemos poner esto como primera línea. Es equivalente del `import` de las clases.

### La JSP de crear un nuevo auto (1)

```
<%@ page import =
 "com.aurumsol.cursojava.auto.negocio.*" %>
<%
String nombre = request.getParameter("nombre");
String marca = request.getParameter("marca");
int anyoFabricacion = Integer.parseInt(
 request.getParameter("ano"));
int valor = Integer.parseInt(
 request.getParameter("valor"));

NgcAuto ngcAuto = new NgcAuto();
boolean exito = ngcAuto.crearNuevo(nombre, marca,
 anyoFabricacion, valor);
%>
```

**RECOGER ENTRADA**

**LLAMAR NEGOCIO**

### La JSP de crear un nuevo auto (2)

```
<HTML>
<HEAD><TITLE>Ingresar</TITLE></HEAD>
<BODY>
<%if (exito) {
 out.print("Guardado un nuevo auto.");
}else{
 out.print("Nombre repetido.");
}%>
</BODY>
</HTML>
```

**SALIDA**

### La JSP de obtener el valor de los autos de una marca en un año

```
<%@ page import =
 "com.aurumsol.cursojava.auto.negocio.*"%>
<%String marca = request.getParameter("marca");
int anyo = Integer.parseInt(
 request.getParameter("ano"));

NgcAuto ngcAuto = new NgcAuto();
int valor =
 ngcAuto.obtieneValorAutosMarcaAnyo(marca, anyo);
%>

<HTML><HEAD><TITLE>Ingresar</TITLE></HEAD>
<BODY><H1>El valor de los autos de la marca
 <%=marca%> en el año <%=anyo%> es
 <%=valor%> dólares</H1>
</BODY></HTML>
```

**RECOGER ENTRADA**

**LLAMAR NEGOCIO**

**SALIDA**



### Ejercicio

- Vamos a hacer un ejercicio de gestión de los clientes de la empresa. Un cliente puede comprar algo y pagar después (es decir, comprar a crédito). El crédito de los clientes es ilimitado. Por supuesto, no se permite que el cliente pague más de lo que ha comprado (somos honrados).
- Creen la clase de dominio Cliente con atributos privados de nombre, nombre de departamento del país, monto comprado y monto pagado. Los métodos serán iniciar un cliente, comprar una cierta cantidad de dólares, pagar una cierta cantidad de dólares y obtener la deuda del cliente con la empresa.
- Creen una aplicación que pida el nombre de un Departamento por página y escriba el total de monto adeudado que hay en ese Departamento.

### Ejercicio

- Lo haremos por partes.
- Primero creen la capa de dominio.
- Después la capa de datos.
- Después la capa de negocio.
- Después la capa de presentación.

### Programa del curso

1. Objetivos y metodología del curso.
2. Repaso del curso anterior.
- 3. Ampliación del lenguaje Java.
4. Cookies y rastreo de sesión.
5. Otros aspectos.
6. Un ejemplo práctico.

### Vamos a ampliar nuestro estudio del lenguaje Java

- En el curso anterior sólo habíamos dado lo más fundamental y básico de este lenguaje (clases, objetos, paquetes, **new**, acceso a atributos y métodos).
- Por falta de tiempo, ignoramos la mayor parte de Java y, en particular, las construcciones más potentes de ese lenguaje: herencia, interfaces, miembros estáticos.
- Ahora es el momento de ver todas estas construcciones y cuando se utilizan.

### Como ejemplo tomaremos las clases Cliente y Libreta

```
package com.aurumsol.banco.dominio;
public class Cliente {
 private String nombre;
 private int credito;
 public void iniciar(String nombre){
 this.nombre = nombre;
 this.credito = 0;
 }
 public void setCredito(int credito){
 this.credito = credito;
 }
 public String getNombre(){
 return this.nombre;
 }
 public int getCredito(){
 return this.credito;
 }
}
```

### Como ejemplo tomaremos las clases Cliente y Libreta

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 public void iniciar(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public int getNumero(){
 return this.numero;
 }
 public int getSaldo(){
 return this.saldo;
 }
}
```



### Como ejemplo tomaremos las clases **Cliente** y **Libreta**

```
public void depositar(int monto){
 this.saldo += monto;
}

public boolean retirar(int monto){
 if (monto > this.saldo){
 return false;
 } else {
 this.saldo -= monto;
 return true;
 }
}
```

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### El método `out.print` de las JSP

- Hasta ahora, lo hemos utilizado en muy diversas situaciones.

```
out.print("Hola a todos");
//Recibe un String como parámetro
out.print(5);
//Recibe un entero como parámetro
out.print(5.5);
//Recibe un double como parámetro
```

### El método `out.print` de las JSP

- Parece ser que este método puede recibir muchos tipos de parámetros diferentes.

```
out.print("Hola a todos");
//Recibe un String como parámetro
out.print(5);
//Recibe un entero como parámetro
out.print(5.5);
//Recibe un double como parámetro
```

### Esto es práctico

- Ya que, en caso contrario, tendríamos que tener un método para cada tipo de parámetro. Más difícil de recordar.

```
out.printString("Hola a todos");
out.printInt(5);
out.printDouble(5.5);
```



### A esto se le llama sobrecarga

- Es cuando un mismo método puede recibir:
  - parámetros de un tipo diferente
  - un número diferente de parámetros.

```
out.print("Hola a todos");
//Recibe un String como parámetro
out.print(5);
//Recibe un entero como parámetro
out.print(5.5);
//Recibe un double como parámetro
```

### La sobrecarga es útil en multitud de situaciones

- Recordemos el método para iniciar una Libreta. Se le pasa un número de libreta. El saldo se fija a cero.

```
public void iniciar(int numero){
 this.numero = numero;
 this.saldo = 0;
}
```

```
libreta1.iniciar(123);
libreta1.iniciar(355);
```

### La sobrecarga es útil en multitud de situaciones

- Podríamos tener un método que le pasáramos el saldo inicial.

```
public void iniciarConSaldo(int numero,
 int saldo){
 this.numero = numero;
 this.saldo = saldo;
}
```

```
libreta1.iniciarConSaldo(123,1000);
libreta1.iniciarConSaldo(355,2000);
```

### La sobrecarga es útil en multitud de situaciones

- Sería útil que **iniciar** e **iniciarConSaldo** se escribieran de la misma forma. Más fácil de recordar.
- Si no se le pasa un saldo, se asume que es saldo 0.

```
libreta1.iniciar(123);
libreta1.iniciar(123,2000);
```

- Es decir, la sobrecarga sirve para dar valores por defecto a los parámetros. Así:

### Cómo se programan métodos sobrecargados

- Los métodos sobrecargados, se declaran como **varios métodos diferentes con el mismo nombre y diferente lista de parámetros**.
- "Método sobrecargado" es un abuso de lenguaje. En realidad, son varios métodos con el mismo nombre.

### Cómo se declaran métodos sobrecargados

- Dentro de la clase **Libreta**, sería así

```
public void iniciar(int numero){
 this.numero = numero;
 this.saldo = 0;
}

public void iniciar(int numero,int saldo){
 this.numero = numero;
 this.saldo = saldo;
}
```



### Cómo se declaran métodos sobrecargados

- Otra opción, más fácil para el mantenimiento, pues evita la repetición de código, es llamar de un método a otro.

```
public void iniciar(int numero){
 this.iniciar(numero, 0);
}

public void iniciar(int numero, int saldo){
 this.numero = numero;
 this.saldo = saldo;
}
```

### Cómo se declaran métodos sobrecargados

- Sea como sea, se trata de métodos con el mismo nombre y diferentes tipos de los parámetros.

```
public void iniciar(int numero){
 this.iniciar(numero, 0);
}

public void iniciar(int numero, int saldo){
 this.numero = numero;
 this.saldo = saldo;
}
```

### Más posibilidades

- Podríamos tener un iniciar al que se le pasara una cadena. Así:

```
public void iniciar(String numero){
 this.iniciar(Integer.parseInt(numero), 0);
}

public void iniciar(int numero){
 this.iniciar(numero, 0);
}

public void iniciar(int numero, int saldo){
 this.numero = numero;
 this.saldo = saldo;
}
```

### ¿Cómo funciona esto?

Si ejecutamos `libreta1.iniciar("123");`  
Java detecta que el parámetro pasado es una cadena y por tanto ejecuta

```
public void iniciar(String numero)
```

Si ejecutamos `libreta1.iniciar(123);`  
Java detecta que el parámetro pasado es un entero y por tanto ejecuta

```
public void iniciar(int numero)
```

Si ejecutamos `libreta1.iniciar(123, 200);`, Java detecta que los parámetros pasados son dos enteros y, por lo tanto ejecuta

```
public void iniciar(int numero, int saldo)
```

### Resumen

- Los métodos sobrecargados tienen el mismo nombre pero diferentes tipos de parámetros.
- A partir de los tipos de los parámetros que le pasamos, Java identifica qué método debe ejecutar.

### Pregunta

- ¿Los siguientes métodos son sobrecargados?

```
public float duplicar(float num)
public double duplicar(float num)
```



### Respuesta

```
public float duplicar(float num)
public double duplicar(float num)
```

- No lo son, pues a pesar de tener el mismo nombre tienen el mismo tipo de parámetros.
- Más importante, Java no compilaría estos dos métodos si estuvieran en la misma clase

### ¿Por qué no compila?

```
public float duplicar(float num)
public double duplicar(float num)
```

- Si hiciéramos  
`objeto.duplicar(5.0f)`
- ¿cómo sabe Java cuál de los dos métodos usar?

### Regla de Java

- Dos métodos de una misma clase no pueden tener al mismo tiempo:
  - El mismo nombre
  - El mismo tipo de parámetros.

### Dicho de otra manera

- Los métodos de una misma clase deben tener:
  - O bien **distinto nombre** (métodos diferentes).
  - O bien **distintos tipos de parámetros** (métodos sobrecargados).

### Ejercicio

- Complementar la clase Libreta con un atributo booleano que indica si la libreta es corriente (es decir, puede emitir cheques) o de ahorro (es decir, no puede emitir cheques). Para la libreta corriente, el booleano será true y en la de ahorro será false.
- Crear un método sobrecargado para fijar el tipo de libreta: puede aceptar una cadena ("Corriente") o un booleano.

### Solución

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 private boolean tipo; //true es corriente
 public void setTipo(boolean tipo){
 this.tipo = tipo;
 }
 public void setTipo(String tipo){
 if (tipo.equals("Corriente")){
 this.setTipo(true);
 }else{
 this.setTipo(false);
 }
 }
}
```



### Ahora se puede hacer

```
libretal.setTipo(true);
```

O bien

```
libretal.setTipo("Corriente");
```

### Un método demasiado sobrecargado: un desastre

```
public void comodin(boolean tipo){
 this.tipo = tipo;
}
```

- Si se pasa un booleano, fija el tipo.

```
public void comodin(int saldo){
 this.saldo = saldo;
}
```

- Si se pasa un entero, fija el saldo.

### Esto plantea la pregunta

- ¿Qué criterios utilizaremos para decidir qué tareas se pueden agrupar en un método (aunque sea sobrecargado) y qué tareas deben incluirse en métodos diferentes?

### Unos cuantos criterios

- Un método debe representar un solo concepto bien distinto y detallado.
- Debe realizar una cosa bien definida.
- No debe ser muy largo (si es posible, menos de una pantalla).
- No debe tener muchos parámetros.

### Recordemos la clase Libreta

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 public void iniciar(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public int getNumero(){
 return this.numero;
 }
 public int getSaldo(){
 return this.saldo;
 }
}
```

### Recordemos la clase Libreta

```
public void depositar(int monto){
 this.saldo += monto;
}
public boolean retirar(int monto){
 if (monto > this.saldo){
 return false;
 } else {
 this.saldo -= monto;
 return true;
 }
}
```



### Problemas de la clase Libreta (1)

```
Libreta libreta = new Libreta(); //Crea
libreta.iniciar(134); //Inicializa
...
```

- Para crear la libreta hay que hacer dos instrucciones: crearla e inicializarla, lo que resulta poco práctico.

### Problemas de la clase Libreta (2)

```
Libreta libreta = new Libreta(); //Crea
libreta.iniciar(134); //Inicializa
libreta.depositar(1000);
libreta.iniciar(125);
...
```

- En cualquier momento, se puede inicializar la libreta de nuevo, lo cual es peligroso, pues se cambia el número de libreta y se asigna el saldo de una libreta a otra.

### Causas de estos problemas

- Tenemos dos operaciones:
  - Una para crear la libreta (**new**).
  - Una para darle su número (**iniciar**).
- No está bien modelado. En un banco, el número de una libreta sólo se determina al momento de la creación **y no se puede cambiar**.
- En nuestro caso, sí se puede cambiar y esto puede dar a errores de programación muy peligrosos.

### La solución a este problema

- Sería que el número de la libreta sólo se pudiera asignar en el momento de la creación.
- O sea, tener una única operación que fuera la combinación de **new+iniciar**
- Podría escribirse así:

```
Libreta libreta = new Libreta(134);
```

### Un poco de terminología

- A este método que ha surgido de la suma de **new+iniciar** y que se ejecuta así:

```
Libreta libreta = new Libreta(134);
```

se le llama **constructor**. 1ª definición:

- **Constructor** es un tipo especial de método que determina como se inicializa un objeto **cuando se crea**.

### Cómo se ejecutan los constructores en Java

- La llamada al constructor de **Libreta** sería así:

```
Libreta libreta = new Libreta(134);
```

- En general, la llamada es de este estilo:

```
new NombreClase(listaparametros)
```



### Observación

- Fíjense que si utilizamos el constructor (y no el método anterior **iniciar**) no podemos inicializar varias veces **Libreta**, ya que el método para inicializar ya que sólo se puede hacer **new** una vez (cuando se crea el objeto).
- Esto aumenta la robustez de nuestro programa y nos previene de posibles errores de programación.

### Como se declaran los constructores

Un método normal se declara así:

```
ámbito tipores nombre(listaparam) {
 sentencias
}
```

- Un constructor se declara así:

```
ámbito NombreClase(listaparam) {
 sentencias
}
```

- Los elementos en negro son opcionales.

### Como se declaran los constructores

```
ámbito NombreClase(listaparam) {
 sentencias
}
```

- En los constructores no se declara el tipo del resultado.
- En los constructores, el nombre del método es el nombre de la clase.

### Ejemplo: constructor de Libreta

Sin tipo de retorno      El nombre del constructor es el mismo que el de la clase

```
public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
}
```

### Ahora la clase Libreta queda así

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public int getNumero(){
 return this.numero;
 }
 public int getSaldo(){
 return this.saldo;
 }
}
```

### Ahora la clase Libreta queda así

```
public void depositar(int monto){
 this.saldo += monto;
}
public boolean retirar(int monto){
 if (monto > this.saldo){
 return false;
 } else {
 this.saldo -= monto;
 return true;
 }
}
```



### Ejercicio

- Se les proporcionará la clase Cliente. Cámbienla para que, en vez del método iniciar, tenga un constructor.

### Solución: la nueva clase Cliente

```
package com.aurumsol.banco.dominio;
public class Cliente {
 private String nombre;
 private int credito;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 }
 public void setCredito(int credito){
 this.credito = credito;
 }
 public String getNombre(){
 return this.nombre;
 }
 public int getCredito(){
 return this.credito;
 }
}
```

### Ejercicio

- Cambien la clase de negocio para usar la forma tradicional que usábamos hasta ahora, es decir, **new Cliente()**
- ¿Qué ven?

### Da un error de compilación

- Es uno de los errores de los que te indica Eclipse.
- Parece que ahora no podemos hacer **new Cliente()** sino que le tenemos que pasar un nombre.
- ¿Qué pasó con la opción **new Cliente()** que estaba disponible hasta ahora?

### Todas las clases tienen un constructor por defecto

- Es el constructor **NombreClase()** – sin parámetros. Este constructor se define por defecto cuando el programador no ha especificado constructores para una clase.
- Se le puede llamar constructor implícito, pues, aunque no está en el código de la clase, se encuentra allí y se puede utilizar. Es el que hemos usado hasta ahora.
- Si el programador define un (o varios) constructores para una clase, el constructor sin parámetros deja de estar disponible automáticamente (aunque puede ser programado explícitamente).

### Todas las clases tienen un constructor implícito si no se define otro

- El constructor implícito sólo crea el objeto y no hace nada más.
- No tiene parámetros y es el que hemos estado usando hasta ahora para definir nuevos objetos.

```
new NombreClase()
```



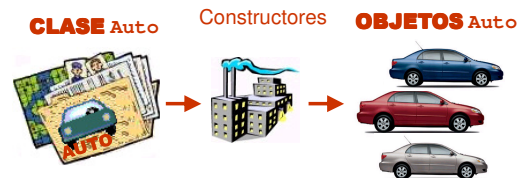
### Otra definición de constructor

- Todo esto nos lleva a una nueva definición de constructor más adecuada:

**Constructor** es aquel método que crea un objeto (no local) de la clase que lo contiene.

### Los constructores nos permiten crear objetos a partir de clases

- En el ejemplo, los constructores permiten crear objetos **Auto** a partir de la clase **Auto**.



### Introduciendo un constructor sin parámetros

- Si definimos otro constructor, el constructor sin parámetros (implícito) deja de estar por defecto.
- Sin embargo, si lo necesitamos, siempre podemos definirlo explícitamente de la forma  

```
public NombreClase() {
}
```
- También se puede poner algún código en el cuerpo de este constructor, aunque esto no es muy usual.
- Se le puede llamar constructor sin parámetros explícito, para distinguirlo del constructor implícito sin parámetros, que es el que está por defecto.

### Ejercicio

- En la clase **Libreta**, definir un constructor sin parámetros.

### Solución: un constructor con parámetros y otro sin parámetros

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero, saldo;
 public Libreta() {}
 public Libreta(int numero) {
 this.numero = numero;
 this.saldo = 0;
 }
 public int getNumero() {
 return this.numero;
 }
 public int getSaldo() {
 return this.saldo;
 }
}
```

Constructores sobrecargados

### Solución: un constructor con parámetros y otro sin parámetros

```
public void depositar(int monto) {
 this.saldo += monto;
}
public boolean retirar(int monto) {
 if (monto > this.saldo) {
 return false;
 } else {
 this.saldo -= monto;
 return true;
 }
}
```



### Los constructores también pueden estar sobrecargados

- Muy común. Por ejemplo: algunos constructores de la clase String (hay 11)

```
public String();
public String(String value);
public String(StringBuffer
 buffer);
public String(byte[] bytes);
public String(char[] value);
```

↑  
Arrays de bytes y de caracteres

### Los constructores también pueden estar sobrecargados

- Por ejemplo, se puede hacer:

```
cadena = new String();
 //Crea una cadena vacía
cadena = new String("Hola");
 //Crea la misma cadena
"Hola"
```

### Por ejemplo, en la clase Libreta se puede tener

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero, saldo;
 public Libreta(){}
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public Libreta(int numero, int saldo){
 this.numero = numero;
 this.saldo = saldo;
 }
}
```

Constructores sobrecargados

### Como llamar a un constructor desde dentro de una clase

- A veces, varios constructores comparten un mismo código. Por ello, un constructor puede llamar a otro. Esto es igual que los métodos con sobrecarga, que suelen llamarse unos a otros.
- Esto es un problema, pues los constructores no tienen nombre.
- La llamada debe ser la **primera línea** del constructor y debe tener la forma:

**this (listaparametros)**

### Por ejemplo, inicialización de libretas de banco teníamos

Se supone que el saldo es 0 si no se especifica.

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero, saldo;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public Libreta(int numero, int saldo){
 this.numero = numero;
 this.saldo = saldo;
 }
}
```

### Por ejemplo, inicialización de libretas de banco teníamos

La redundancia de código disminuye la legibilidad y dificulta el mantenimiento.

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero, saldo;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public Libreta(int numero, int saldo){
 this.numero = numero;
 this.saldo = saldo;
 }
}
```



### Una solución mejor

Si un constructor llama a otro, ya no hay redundancia de código y la clase es más fácil de mantener.

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero, saldo;
 public Libreta(int numero) {
 this(numero, 0);
 }
 public Libreta(int numero, int saldo){
 this.numero = numero;
 this.saldo = saldo;
 }
}
```

**Llama al otro constructor**

### Llamar a un constructor desde otro es una buena práctica

- Reduce la repetición de código.
- Mejora la legibilidad.
- Facilita el mantenimiento.

### Llamar a un constructor desde otro es una buena práctica

Por ejemplo, si quisiéramos cambiar el nombre de los atributos, sólo deberíamos cambiar el 2º constructor

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numeroLib, saldo;
 public Libreta(int numero) {
 this(numero, 0);
 }
 public Libreta(int numero, int saldo){
 this.numeroLib = numero;
 this.saldo = saldo;
 }
}
```

**Llama al otro constructor**

### Resumen:

Un método normal se declara así:

```
ámbito tipores nombre(listaparam) {
 sentencias
}
```

Y se invoca así: **objeto.nombre(params)**

- Un constructor se declara así:

```
ámbito NombreClase(listaparam) {
 sentencias
}
```

Y se invoca así: **new NombreClase(params)**

### Nota importante

- Las clases persistentes en Hibernate deben tener obligatoriamente un constructor sin parámetros (**sea éste implícito o explícito**).
- Si no queremos que una clase tenga un constructor sin parámetros pero queremos persistirla en Hibernate, la solución es declarar **un constructor sin parámetros que sea privado**.
- Como es privado, nadie lo va a utilizar fuera de la clase.
- Sólo lo pondremos allí para Hibernate y no lo utilizaremos nunca. En esto se parece al `private int id`.

### Por ejemplo, clase Libreta para persistirla en Hibernate

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int id;
 private int numero, saldo;
 private Libreta() {}
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public int getNumero(){
 return this.numero;
 }
 //Aquí los otros métodos
}
```

**Constructor sin parámetros que es privado, porque sólo lo queremos para Hibernate y nada más. (Es parecido al private int id).**



### Una observación

- Constructores: métodos que se ejecutan cuando se crea un objeto.
- En otros lenguajes hay destructores: métodos que se ejecutan cuando se destruye un objeto.
- En Java esto no se suele necesitar pues el “garbage collector” se encarga de liberar los recursos.
- Sólo se necesita cuando desde Java llamamos a código escrito en otros lenguajes.
- En ese caso, el destructor podemos implementarlo en un método normal (pero debemos de acordarnos de ejecutarlo antes de destruir el objeto)

### Ejercicios

- Queremos programar una clase de dominio para un estudiante universitario.
- La información que nos interesa de cada estudiante es su nombre, curso que estudia (de 0 a 5) y la nota media de las materias que ha cursado hasta ahora.
- Los procesos que nos interesa para cada estudiante es ingresar el estudiante en la Universidad (para el cual debemos dar su nombre y, opcionalmente, su curso), obtener su nombre, obtener y cambiar su curso y su nota media.
- Hacerlo con constructores.

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### Miembros estáticos

Queremos programar un sistema de matriculación de alumnos para una Universidad. Creamos una clase Alumno que aquí presentamos simplificada.

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private int numMatricula;
 public Alumno(int numMatr){
 this.numMatricula = numMatr;
 }
 public int getNumMatricula () {
 return this.numMatricula;
 }
}
```

### Miembros estáticos

El constructor asigna el número de matrícula que no puede ser modificado posteriormente.

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private int numMatricula;
 public Alumno(int numMatr){
 this.numMatricula = numMatr;
 }
 public int getNumMatricula () {
 return this.numMatricula;
 }
}
```

### Pregunta

- ¿Qué pasaría si deseáramos que el número de matrícula fuera asignado automáticamente por el sistema de forma correlativa?
- Así, el primer alumno que se matricule tendría el número 1, el segundo el número 2, etc.
- Esto quiere decir que el primer objeto que se crea tiene el número 1, etc.



### Normalmente

- Esto normalmente lo haríamos con la base de datos y Hibernate (parecido al atributo **id** que habíamos visto anteriormente).
- Pero aquí suponemos que queremos hacerlo en memoria.

### ¿Cómo podemos hacer esto?

Por ahora no lo podemos hacer. Para ello deberíamos mantener una variable "contador de alumnos". Esta variable debería ser global a todos los objetos.

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private int numMatricula;
 public Alumno(int numMatr){
 this.numMatricula = numMatr;
 }
 public int getNumMatricula () {
 return this.numMatricula;
 }
}
```

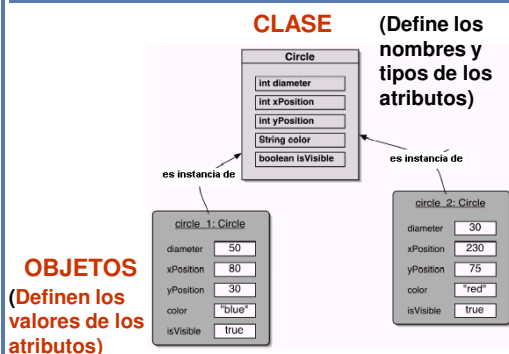
### La variable "contador de alumnos"

- Su valor debería ser global a todos los objetos de la clase **Alumno** e incrementarse cada vez que se crea un nuevo objeto.
- Debería ser accesible para todos los objetos de la clase **Alumno**.

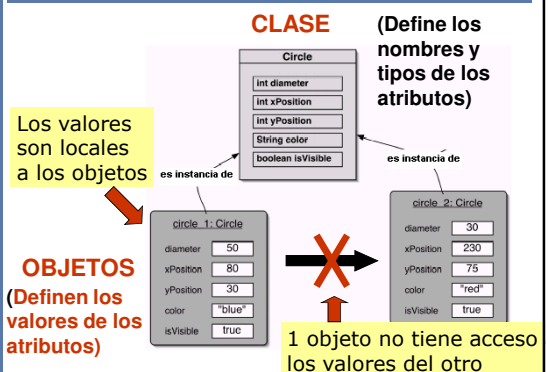
### Esto no es posible por ahora

- Los valores de atributos son locales a cada objeto.
- Un objeto no puede acceder a los valores de los atributos de otro objeto de la misma clase que se ha definido de forma independiente de él  
(**alumno1** no puede acceder a los atributos de **alumno2**)

### Recordemos: Una clase y sus objetos



### Recordemos: Una clase y sus objetos

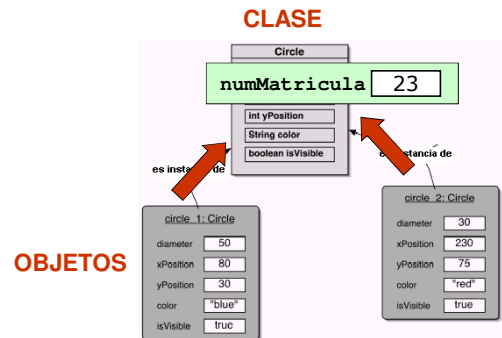




### Solución

- Crear un **tipo especial de atributo cuyo valor esté asociado a una clase**, en vez de a un objeto.
- Así:
  - sería visible por todos los objetos de esta clase.
  - podría incrementarse cada vez que se defina un objeto.

### Si el valor está en la clase, todos los objetos tienen acceso



### Atributos cuyo valor está asociado a una clase

- En Java se llaman atributos **static** o atributos de clase.
- Sólo existe uno de esos elementos para todos los objetos de una clase (compartido).
- El valor es común a todos los objetos.
- El atributo existe aunque no exista ningún objeto.

### Un poco de terminología

- Atributos cuyo valor está asociado a un objeto se llaman:  
**Atributos de instancia o simplemente atributos.**
- Son los que hemos visto hasta ahora.
- Atributos cuyo valor está asociado a una clase se llaman:  
**Atributos de clase, atributos static o atributos estáticos.**

### Declaración de un atributo estático

- Se declaran  
**ámbito static tipo nombre = valor inicial**  
(donde los elementos en rojo son obligatorios y en negro opcionales)
- La única diferencia con la sintaxis de una declaración de atributo normal es la palabra **static**.
- El valor inicial es el valor que tiene mientras no se modifique.

### Ejemplo de Declaración de un atributo static

```
private static int nAlumnos = 0
```

- Creamos una variable **nAlumnos** (que contiene el número de Alumnos, es decir, el último número de matrícula).
- Inicialmente (cuando se crea la clase) tiene el valor 0.
- Se podrá acceder por todos los objetos de la clase en que esté definida.
- Existe aunque no exista ningún objeto.



### Acceso a un atributo *static*

- |                                                        |                               |
|--------------------------------------------------------|-------------------------------|
| <b>Clase atributo</b>                                  | <code>Alumno.nAlumnos</code>  |
| • Siempre.                                             |                               |
| <b>objeto atributo</b>                                 | <code>alumno1.nAlumnos</code> |
| • En otra clase y siempre que haya un objeto definido. |                               |
| <b>atributo</b>                                        | <code>nAlumnos</code>         |
| • En la misma clase.                                   |                               |
| <b>this atributo</b>                                   | <code>this.nAlumnos</code>    |
| • En un método de instancia de la misma clase.         |                               |

### Acceso a un atributo *static*

- |                                                        |                               |
|--------------------------------------------------------|-------------------------------|
| <b>Clase atributo</b>                                  | <code>Alumno.nAlumnos</code>  |
| • Siempre.                                             |                               |
| <b>objeto atributo</b>                                 | <code>alumno1.nAlumnos</code> |
| • En otra clase y siempre que haya un objeto definido. |                               |
| <b>atributo</b>                                        | <code>nAlumnos</code>         |
| • En la misma clase.                                   |                               |
| <b>this atributo</b>                                   | <code>this.nAlumnos</code>    |
| • En un método de instancia de la misma clase.         |                               |

### Acceso a un atributo *static*

- |                       |                              |
|-----------------------|------------------------------|
| <b>Clase atributo</b> | <code>Alumno.nAlumnos</code> |
| • Siempre.            |                              |
- Se prefiere esta sintaxis:
- Está disponible siempre.
  - Es más legible y fácil de mantener.
  - Se ve enseguida que es un atributo estático (por la mayúscula de la clase).
  - No da lugar a malentendidos.

### La sintaxis preferida es más legible

```
Alumno.nAlumnos = 5;
Alumno.nAlumnos++;
out.print (Alumno.nAlumnos);
```

- Se ve que es un atributo **static** pues comienza por un nombre de clase (en mayúscula)

### Sintaxis diferentes a la preferida pueden dar malentendidos

```
<% Alumno alumno1 = new Alumno();
Alumno alumno2 = new Alumno();
alumno1.nAlumnos = 5;
alumno2.nAlumnos = 6;
out.println(alumno1.nAlumnos); %>
```

6

Esta es una de las razones por la que es mejor usar `Alumno.nAlumnos`

### Solución del problema del alumno

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public int getNumMatricula () {
 return this.numMatricula;
 }
}
```



### Solución del problema del alumno

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public int getNumMatricula () {
 return this.numMatricula;
 }
}
```

Cada vez que se crea un alumno se le asigna el correlativo y éste se incrementa

### Ejercicio

- \*Nota. Proporcionar el codi en Eclipse.
- Creen una JSP utilizando esta clase, creando alumnos, obteniendo su número de matrícula y mostrándolo por pantalla.
- Esto viola la arquitectura en n-capas, ya que haríamos una clase de negocio en vez de llamar la JSP directamente desde la clase de dominio, pero lo haremos así por rapidez.

### Ejercicio

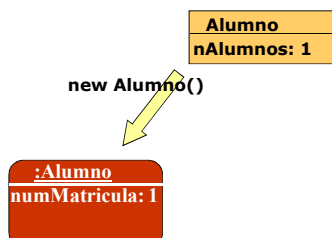
- ¿Qué pasaría si el atributo no fuera **static**? Compruébenlo
  - borrando la palabra **static**.
  - cambiando **Alumnos.nAlumnos** por **this.nAlumnos**.
- Observen los valores de la página Web ¿Por qué son así?

### Cuando usamos el atributo estático

<b>Alumno</b>
<b>nAlumnos: 0</b>

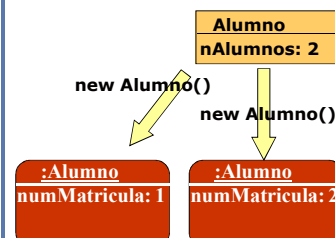
- Cuando se crea la clase, el atributo se inicializa a 0.

### Cuando usamos el atributo estático



- Si se crea una instancia se ejecuta el constructor, que incrementa el atributo estático y lo copia en numMatricula, que es un atributo de instancia.

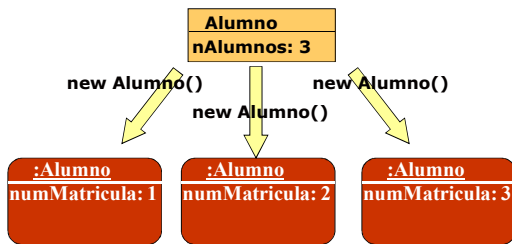
### Cuando usamos el atributo estático



- Lo mismo se repite para cada uno de los objetos que se crean



### Cuando usamos el atributo estático



- Como el atributo es estático, es compartido por todos los objetos de la clase, lo que permite tener un contador de instancias.

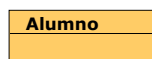
### Sin atributo estático

```

package com.aurumsol.universidad.dominio;
public class Alumno {
 private int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 this.nAlumnos++;
 this.numMatricula = this.nAlumnos;
 }
 public int getNumMatricula () {
 return this.numMatricula;
 }
}

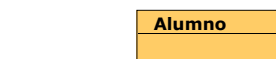
```

### Si hacemos no estático el atributo



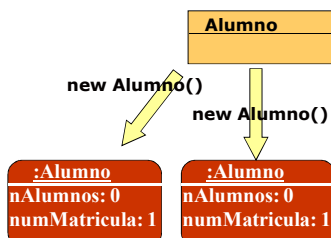
- Ahora se crea la clase, pero esta no tiene asociado un atributo.

### Si hacemos no estático el atributo



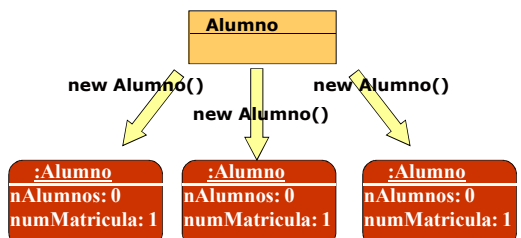
- Cuando se crea el objeto, el atributo nAlumnos se inicializa a 0, el constructor lo incrementa y lo asigna a numMatricula

### Si hacemos no estático el atributo



- Si se crea un nuevo objeto, se crea una nueva copia del atributo y se repite el proceso.

### Si hacemos no estático el atributo



- Como el atributo no se comparte, sino que cada objeto tiene una copia propia, no hay manera de llevar un contador de instancias.



### Una reflexión

- Un atributo **static** es accedido por todos los objetos.
- Un objeto puede encontrar un valor que no esperaba (porque ha sido cambiado por otro objeto).

### Por ejemplo

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public int getNumMatricula () {
 return this.numMatricula;
 }
}
```

### Efectos no esperados

```
<% Alumno alumno1 = new Alumno();
Alumno.nAlumnos = 5;
algunObjeto.algunMetodo();
out.print(Alumno.nAlumnos); %>
```

236

algunMetodo ha creado objetos **Alumno** que nos han cambiado el atributo

### Ejercicio

- Añadan a la clase anterior un método que retorne el valor del atributo **Alumno.nAlumnos**, ya que es **private**.

### Solución

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public int getTotalAlumnos () {
 return Alumno.nAlumnos;
 }
}
```

### Un problema

- Si queremos saber el total de alumnos que hay, debemos crear un objeto primero para poder invocar el método **getTotalAlumnos**.
- En cambio, el atributo **Alumno.nAlumnos** es de clase y no depende de ningún objeto.
- Cuando era público, lo podíamos acceder sin objeto. ¿Porque no podemos hacer lo mismo a través de un método?



### Solución: métodos que no necesiten objetos para ejecutarse

- Se llaman métodos **static** y están asociados a las clases en vez de a los objetos.
- Son similares a los atributos **static**.

### Un poco de terminología

- Métodos que están asociados a objetos y sólo se pueden ejecutar desde objetos

Métodos de instancia o simplemente métodos.

- Métodos que están asociados a una clase y se pueden ejecutar sin objetos:

Métodos de clase, métodos **static** o métodos estáticos.

### Definición de un método **static**

```
ambito static tipos nombre(listaparam) {
 sentencias
}
```

- Los elementos obligatorios van en rojo y los opcionales en negro.
- La única diferencia con un método normal (de instancia) es la palabra **static** antes del tipo del resultado.

### Ejemplo de declaración de un método **static**

```
public static int getTotalAlumnos () {
 //Aquí las sentencias del método
}
```

- La única cosa especial es la palabra reservada **static**.

### Ejercicio

- Ampliar la clase **Alumno** con un método **static** que devuelva el número de alumnos que se han creado. Comprobarla ejecutándola desde la JSP.

### Solución

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public static int getTotalAlumnos () {
 return Alumno.nAlumnos;
 }
}
```



### Ejecución de un método *static* en Java

- Clase.método()** `Alumno.devTotal()`
- Siempre.
- objeto.método()** `alumno1.devTotal()`
- En otra clase y siempre que haya un objeto definido.
- método()** `devTotal()`
- En la misma clase.
- this.método()** `this.devTotal()`
- En un método de instancia de la misma clase.

### Ejecución de un método *static* en Java

- Clase.método()** `Alumno.devTotal()`
- Siempre.
- objeto.método()** `alumno1.devTotal()`
- En otra clase y siempre que haya un objeto definido.
- método()** `devTotal()`
- En la misma clase.
- this.método()** `this.devTotal()`
- En un método de instancia de la misma clase.

### Ejecución de un método *static* en Java

- Clase.método()** `Alumno.devTotal()`
- Siempre.
- Se recomienda esta forma:
- Está disponible siempre.
  - Es más legible y fácil de mantener.
  - No vamos a ver otras.

### Ejercicio

- Cada alumno paga 500 dólares de matrícula. Crear un nuevo método en la clase **Alumno**, llamado **getMonto** que devuelva el monto total recaudado en la matrícula.
- Para programar **getMonto** no se debe acceder a ningún atributo: sólo debe implementarse ejecutando métodos.

### Solución

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno(){
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public static int getTotalAlumnos () {
 return Alumno.nAlumnos;
 }
 public static int getMonto(){
 return Alumno.getTotalAlumnos()*500;
 }
}
```

### Ejercicio

- ¿Qué pasa si el cuerpo del método **getMonto** es sustituido por `return this.getTotalAlumnos()*500;`?
- Pruébenlo en Eclipse.



### Solución

- Aparece el siguiente mensaje de error

```
Cannot use this in a static context
Press 'F2' for focus.
```

- Cannot use this in a static context

### Esto ya lo habíamos avisado

**NombreClase.nombreMétodo()**

- Siempre.
- **nombreObjeto.nombreMétodo()**
- En otra clase y siempre que haya un objeto definido.

**nombreMétodo()**

- En la misma clase

**this.nombreMétodo();**

- En un método de instancia de la misma clase.

### La regla general es

- Dentro de un método **static**, no pueden usarse atributos o métodos de instancia (**this** se considera una variable de instancia)
- Esto es lógico, ya que puede que no exista el objeto y, por lo tanto, tampoco los elementos de instancia.

### Entendiendo las cosas

- **Integer.parseInt** es un método **static** de la clase **Integer**.
- **String.valueOf** es un método **static** de la clase **String**.
- **Math.sin** es un método **static** de la clase **Math**.
- Todos ellos pueden ser invocados sin necesidad de crear instancias de sus clases respectivas.

### Orden recomendado de los atributos y métodos en una clase

```
public class NombreClase {
 atributos estáticos
 atributos de instancia
 métodos estáticos
 métodos de instancia
}
```

### Utilidad de los elementos (atributos y métodos) **static**

- ¿Qué ventajas presentan?
- ¿Qué desventajas presentan?
- ¿Cuándo usarlos?



### Utilidad de los elementos (atributos y métodos) *static*

- ¿Qué ventajas presentan?
- ¿Qué desventajas presentan?
- ¿Cuándo usarlos?

### Ventajas: Pocas y dudosas

- Se los puede usar sin instanciar la clase.
- Son compartidos por todos los miembros de la clase.
- Si son públicos, están accesibles globalmente en el programa.
- Aunque esto es una ventaja a veces, suele ser más desventaja, pues rompe con la programación orientada a objetos.

### Utilidad de los elementos (atributos y métodos) *static*

- ¿Qué ventajas presentan?
- ¿Qué desventajas presentan?
- ¿Cuándo usarlos?

### Desventajas: Muchas y grandes

- Rompen la filosofía de la programación orientada a objetos: una clase con sólo miembros estáticos es como un módulo tradicional.
  - No necesita crear objetos.
  - No es como un tipo de datos sino como una lista de funciones.
  - No puede usar herencia, interfaces (esto ya lo veremos).
- Por ello con miembros estáticos renunciamos a las ventajas de la orientación a objetos: flexibilidad, facilidad de mantenimiento.
- Pueden producir efectos no esperados, por ser compartidos: multithreading, etc.
- El orden en que Java inicializa los atributos estáticos puede producir errores de programación difíciles de localizar.
- Si tienen muchos en un programa, es indicio que el programa no está bien diseñado.
- Se debe limitar su uso.

### Los miembros estáticos son como una sierra eléctrica

- Son muy peligrosos.
- Si se usan bien son muy útiles.
- Si se usan mal pueden provocar destrozos.
- Hay que pensarlo tres veces antes de usarlos.

### Utilidad de los elementos (atributos y métodos) *static*

- ¿Qué ventajas presentan?
- ¿Qué desventajas presentan?
- ¿Cuándo usarlos?



### Por ejemplo, hasta ahora hemos visto como se usan los miembros estáticos

- Para contar los objetos que se crean.
- Esto lo hemos hecho por motivos didácticos.
- Pero no debería hacerse así.
- Por lo tanto, el uso que hemos hecho hasta ahora de los miembros estáticos es inadecuado.

### El problema del alumno no es bueno programarlo con elementos static

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public static int getTotalAlumnos () {
 return Alumno.nAlumnos;
 }
 public static int getMonto() {
 return Alumno.getTotalAlumnos()*500;
 }
}
```

### Sería mejor programarlo así

```
package com.aurumsol.universidad.dominio;
public class Universidad {
 private String nombre;
 private List listaAlumnos;
 public void ingresaAlumno(Alumno alumno) {
 listaAlumnos.add(alumno);
 }
 public int getTotalAlumnos () {
 return listaAlumnos.size();
 }
 public int getMonto () {
 return this.getTotalAlumnos()*500;
 }
}
```

### Si queremos contar los elementos que tenemos

- Es mucho mejor colocar todos los elementos dentro de otra clase que usar miembros estáticos.
- En nuestro caso, hemos puesto todos los alumnos dentro de la clase Universidad, en vez de usar miembros estáticos.
- No rompemos el paradigma de la orientación a objetos.
- Podemos recorrer la lista o hacer búsquedas sobre ella, lo cual es imposible con métodos estáticos.

### Por ejemplo, podemos recorrer la lista y sacar un reporte de todos los alumnos

```
package com.aurumsol.universidad.dominio;
public class Universidad {
 private String nombre;
 private List listaAlumnos;
 public void ingresaAlumno(Alumno alumno) {
 listaAlumnos.add(alumno);
 }
 public int getTotalAlumnos () {
 return listaAlumnos.size();
 }
 public int getMonto () {
 return this.getTotalAlumnos()*500;
 }
}
```

### Intenten sacar la lista de los nombres aquí y verán qué les pasa

```
package com.aurumsol.universidad.dominio;
public class Alumno {
 private static int nAlumnos = 0;
 private int numMatricula;
 public Alumno() {
 Alumno.nAlumnos++;
 this.numMatricula = Alumno.nAlumnos;
 }
 public static int getTotalAlumnos () {
 return Alumno.nAlumnos;
 }
 public static int getMonto() {
 return Alumno.getTotalAlumnos()*500;
 }
}
```



### Utilidad de los elementos (atributos y métodos) *static*

- ¿Qué desventajas presentan?
- ¿Cuándo usarlos?

Si no son buenos para contar, para qué utilizarlos

### Cuando usar los miembros estáticos: Los casos más usuales

- Para definir **valores constantes**. Lo veremos a continuación.
- Para implementar algunos patrones de diseño, como el **patrón Singleton**, que veremos en breve.
- Métodos que no requieren objetos pues **todos** sus parámetros y resultados son de tipos primitivos **Math.sin**. Aunque es mejor usar el patrón Singleton.
- El método **public static void main**, que se usa en programación de escritorio, pero que aquí no veremos.

### Ejercicio

- Hagan una clase Calculadora con miembros estáticos.
- La calculadora debe sumar, restar y sacar la media aritmética de dos números doubles.

### Constantes

- Valores que no cambian durante toda la ejecución del programa.
- En ello se diferencian de las variables.
- Por ejemplo: la constante matemática  $\pi$  o el tipo de IVA en El Salvador.

### Declaración de constantes

- Se declaran como cualquier atributo pero con la palabra reservada **final**.
- Suelen declararse **static**, ya que así están disponibles incluso si no hay objetos.
- Sintaxis:

**ámbito static final tipo nombre = valor;**

### Ejemplo de declaración de constantes

```
public static final double TIPO_IVA = 0.13;
```

- Es como una declaración normal de atributo. Lo único que difiere es la palabra reservada **final** y que el valor inicial es obligatorio.
- Los nombres de las constantes van siempre con todas sus letras en mayúsculas. Las palabras se separan por guiones por bajo (“underscore”).
- En Java las constantes son sólo un tipo especial de variables. En otros lenguajes son diferentes, pues son tratadas por el preprocesador.



### La importancia de llamarse **final**

- Cuando decimos que un atributo es **final**, Java no nos dejará modificarlo. Si lo intentamos, producirá un error de compilación. Por ejemplo, si hacemos **TIPO\_IVA = 100;**

The final field Factura.TIPO\_IVA cannot be assigned. It must be blank in this context, not qualified and not in compound assignment

- Por lo tanto, es la palabra **final** la que convierte al atributo en una constante.

### Ejemplo de uso de una constante

```
public class Factura {
 private int numero, monto;
 private static final double TIPO_IVA = 0.13;
 //Aquí otros métodos
 public double obtieneIva() {
 return this.monto*TIPO_IVA;
 }
}
```

- Es como cualquier atributo, sólo que ahora declararlo final nos asegura contra la posibilidad de que otro objeto o código lo modifique, lo cual llevaría a cálculos incorrectos.

• **final** es una medida de seguridad: una alarma.

### ¿Para qué usamos constantes? Primera parte

```
public class Factura {
 private int numero, monto;
 private static final double TIPO_IVA = 0.13;
 //Aquí otros métodos
 public double obtieneIva() {
 return this.monto*TIPO_IVA;
 }
}
```

- Si no las hiciéramos **final**, se podrían modificar y esto podría dar errores.
- Declarándolo **final** nos ahorramos estos errores.

### ¿Para qué usamos constantes? Segunda parte

```
public class Factura {
 private int numero, monto;
 private static final double TIPO_IVA = 0.13;
 //Aquí otros métodos
 public double obtieneIva() {
 return this.monto*TIPO_IVA;
 }
}
```

- Si son constantes ¿por qué no hacemos **this.monto\*0.13** y nos ahorramos la molestia de declarar un atributo?

### ¿Para qué usamos constantes? Segunda parte

```
public class Factura {
 private int numero, monto;
 private static final double TIPO_IVA = 0.13;
 //Aquí otros métodos
 public double obtieneIva() {
 return this.monto*TIPO_IVA;
 }
}
```

Por dos motivos:

1. Así es más legible. Distinguimos de otros 0.13.
2. Así es más fácil de mantener. Cuando cambie el tipo del IVA, sólo debemos hacer un cambio en todo el programa.

### Constantes para devolver error en métodos

- Otro uso de las constantes es devolver errores como resultados de un método.
- Lo vamos a ver a continuación con el ejemplo de auto con el que vimos la arquitectura en n-capas.



### Teníamos este método en la capa de negocio. Creaba un nuevo auto.

```
public boolean crearNuevo(String nombre, String
marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 Auto nuevoAuto = new Auto();
 nuevoAuto.setNombre(nombre);
 nuevoAuto.setAnyo(anyoFabricacion);
 nuevoAuto.setMarca(marca);
 nuevoAuto.setValor(valor);
 daoAuto.guardarNuevo(nuevoAuto);
 return true;
 }
}
```

### Vamos a simplificarlo para estudiarlo mejor

```
public class NgcAuto{
 public boolean crearNuevo(String nombre, String
marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 //Crea un nuevo auto y lo guarda
 return true;
 }
 }
 //Aquí otros métodos de la clase de negocio
}
```

### Teníamos una JSP de crear un nuevo auto que llama a esa clase de negocio (1)

```
<%@ page import =
 "com.aurumsol.cursojava.auto.negocio.*" %>
<%
String nombre = request.getParameter("nombre");
String marca = request.getParameter("marca");
int anyoFabricacion = Integer.parseInt(
 request.getParameter("ano"));
int valor = Integer.parseInt(
 request.getParameter("valor"));
NgcAuto ngcAuto = new NgcAuto();
boolean exito = ngcAuto.crearNuevo(nombre, marca,
anyoFabricacion, valor);
%>
```

**RECOGER ENTRADA**

**LLAMAR NEGOCIO**

### Teníamos una JSP de crear un nuevo auto que llama a esa clase de negocio (2)

```
<HTML>
<HEAD><TITLE>Ingresar</TITLE></HEAD>
<BODY>
<%if (exito) {
 out.print("Guardado un nuevo auto.");
}else{
 out.print("Nombre repetido.");
}%>
</BODY>
</HTML>
```

**SALIDA**

### La simplificamos para estudiarla mejor)

```
// Recoger los datos de entrada
NgcAuto ngcAuto = new NgcAuto();
boolean exito = ngcAuto.crearNuevo(nombre, marca,
anyoFabricacion, valor);
%>
<HTML>
<HEAD><TITLE>Ingresar</TITLE></HEAD>
<BODY>
<%if (exito) {
 out.print("Guardado un nuevo auto.");
}else{
 out.print("Nombre repetido.");
}%>
</BODY>
</HTML>
```

**RECOGER ENTRADA**

**LLAMAR NEGOCIO**

**SALIDA**

### El valor booleano que retornamos nos dice si ha habido error

```
public class NgcAuto{
 public boolean crearNuevo(String nombre, String
marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 //Crea un nuevo auto y lo guarda
 return true;
 }
 }
 //Aquí otros métodos de la clase de negocio
}
```



### La única causa que haya error es que ya exista el nombre


```
public class NgcAuto{
 public boolean crearNuevo(String nombre, String
 marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 //Crea un nuevo auto y lo guarda
 return true;
 }
 }
 //Aquí otros métodos de la clase de negocio
}
```

### Si el método devuelve false, sabemos que ya existía el nombre

```
public class NgcAuto{
 public boolean crearNuevo(String nombre, String
 marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 //Crea un nuevo auto y lo guarda
 return true;
 }
 }
 //Aquí otros métodos de la clase de negocio
}
```

### Nos permite dar un mensaje adecuado en la JSP cuando el método devuelve false

```
// Recoger los datos de entrada
NgcAuto ngcAuto = new NgcAuto();
boolean exito = ngcAuto.crearNuevo(nombre, marca,
 anyoFabricacion, valor);
%>
<HTML>
<HEAD><TITLE>Ingresar</TITLE></HEAD>
<BODY>
<%if (exito) {
 out.print("Guardado un nuevo auto.");
}else{
 out.print("Nombre repetido.");
}%>
</BODY>
</HTML>
```



### Ahora imaginemos que hay otra causa de error: que no exista la marca


```
public class NgcAuto{
 public boolean crearNuevo(String nombre, String
 marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 boolean exMarca = daoAuto.hayMarca(marca);
 if (!exMarca){
 return false;
 } else {
 //Crea un nuevo auto y lo guarda
 return true;
 }
 }
 }
}
```

### Ahora se devuelve false si ya existe el nombre y si no existe la marca

```
public class NgcAuto{
 public boolean crearNuevo(String nombre, String
 marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return false;
 }else{
 boolean exMarca = daoAuto.hayMarca(marca);
 if (!exMarca){
 return false;
 } else {
 //Crea un nuevo auto y lo guarda
 return true;
 }
 }
 }
}
```

### La JSP nos da un mensaje de error poco útil por poco concreto

```
// Recoger los datos de entrada
NgcAuto ngcAuto = new NgcAuto();
boolean exito = ngcAuto.crearNuevo(nombre, marca,
 anyoFabricacion, valor);
%>
<HTML>
<HEAD><TITLE>Ingresar</TITLE></HEAD>
<BODY>
<%if (exito) {
 out.print("Guardado un nuevo auto.");
}else{
 out.print("Nombre repetido o marca no existe");
}%>
</BODY>
</HTML>
```





### Sería mucho mejor

- Devolver un mensaje específico.
- Para ello, la clase de negocio debería devolverle a la JSP la información sobre el error concreto que se ha generado (si ha sido por el nombre o por la marca).
- El problema es que el tipo del método de negocio es boolean y sólo hay dos valores posibles de boolean: true (al que le hemos asignado el significado de "sin error") y false (al que le hemos asignado "con error").
- Debemos cambiar el método para que pueda devolver más valores, que se correspondan con los diferentes errores que pueden darse.

### Una solución sería usar enteros

- Devolver 0 si no ha habido error, 1 si el problema era el nombre, 2 si era la marca.
- Lo que pasa es que codificar valores de esta forma puede hacer confuso el código. Así:

```
if (error==1)
```

- Esto es poco legible.
- Para hacerlo más legible usaremos constantes.

### Definimos tres constantes enteras con los errores posibles que devolvemos.

```
public class NgcAuto{
 public static final int ERR_OK = 0;
 public static final int ERR_NOMBRE_EXISTE = 1;
 public static final int ERR_MARCA_NO_EXISTE = 2;
 public int crearNuevo(String nombre, String
 marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return NgcAuto.ERR_NOMBRE_EXISTE;
 }else{
 boolean exMarca = daoAuto.hayMarca(marca);
 if (!exMarca){
 return NgcAuto.ERR_MARCA_NO_EXISTE;
 } else {
 //Crea un nuevo auto y lo guarda
 return NgcAuto.ERR_OK;
 }
 }
 }
}
```

### Las hacemos públicas para que las pueda acceder la JSP

```
public class NgcAuto{
 public static final int ERR_OK = 0;
 public static final int ERR_NOMBRE_EXISTE = 1;
 public static final int ERR_MARCA_NO_EXISTE = 2;
 public int crearNuevo(String nombre, String
 marca, int anyoFabricacion, int valor){
 DAOAuto daoAuto = new DAOAuto();
 boolean yaExisteNombre =
 daoAuto.existeNombre(nombre);
 if (yaExisteNombre){
 return NgcAuto.ERR_NOMBRE_EXISTE;
 }else{
 boolean exMarca = daoAuto.hayMarca(marca);
 if (!exMarca){
 return NgcAuto.ERR_MARCA_NO_EXISTE;
 } else {
 //Crea un nuevo auto y lo guarda
 return NgcAuto.ERR_OK;
 }
 }
 }
}
```

### JSP con constantes (Utiliza las constantes públicas definidas en la clase de negocio)

```
// Recoger los datos de entrada
NgcAuto ngcAuto = new NgcAuto();
int error = ngcAuto.crearNuevo(nombre, marca,
 anyoFabricacion, valor);
%><HTML><HEAD><TITLE>Ingresar</TITLE></HEAD>
<BODY>
<%if (error == NgcAuto.ERR_OK) {
 out.print("Guardado un nuevo auto.");
}else if (error == NgcAuto.ERR_NOMBRE_EXISTE) {
 out.print("Nombre repetido");
}else if (error == NgcAuto.ERR_MARCA_NO_EXISTE) {
 out.print("Marca no existe");
}else {
 out.print("Error de programación");
}%></BODY></HTML>
```

### Ejercicio

- Modifiquen NgcAuto del ejercicio Autos y la JSP correspondiente para que la creación de un auto nuevo no sólo dé error si ya existe el nombre, sino si su valor es mayor que el valor amortizado de todos los autos de la marca nueva en el año de fabricación del auto nuevo (esto es una política de la empresa).
- No deben implementar ningún nuevo método, sino cambiar los métodos actuales.



### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- **3.3. El patrón Singleton.**
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### El patrón Singleton (“único”)

- Es un patrón de diseño que sirve para mejorar la eficiencia de un programa y hacer que utilice menos memoria.
- Hace uso de atributos y métodos estáticos, por eso ahora es el momento de estudiarlo.
- ¿Qué es un patrón de diseño?

### Patrones de diseño

- Soluciones a problemas comunes de la programación orientada a objetos.
- Cuando el programador encuentra un problema, se pregunta a que patrón de diseño pertenece y lo resuelve con la solución que tiene este patrón de diseño.
- Así los libros de patrones de diseño son como un libro de “recetas” con soluciones a problemas comunes.
- Son muy útiles a la hora de programar, aumentan la productividad, la calidad y reusabilidad del código.

### El movimiento de patrones de diseño

- Comienza con “**Design Patterns**” de la “banda de los cuatro” (“Gang of Four” o “GoF”: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides). Presenta 23 patrones básicos (“GoF patterns”). Esta escrita para C++ pero los patrones son aplicables a cualquier lenguaje.
- A partir de este texto, el movimiento de los patrones “explota”. Hay libros que definen patrones para J2EE, para EJB, antipatrones, etc.
- Incluso hay sitios web de programación que tienen secciones de patrones enviados por los usuarios.

### En concreto, el patrón “Singleton”

- Soluciona el problema común de las clases de las cuales sólo queremos que haya una única instancia.
- Se basa en programar un método estático que devuelva la única instancia.
- La instancia se guarda en un atributo estático.

### Un código para el patrón “Singleton”

```
public class Clase {
 private static Clase instancia = new Clase();
 //Aquí los otros atributos
 public static Clase getInstancia() {
 return Clase.instancia;
 }
 //Aquí los otros métodos de la clase
}
```

- Este es el código más sencillo para el patrón. Después, lo mejoraremos. Lo vamos a examinar a continuación.



### Un código para el patrón “Singleton”

```
public class Clase {
 private static Clase instancia = new Clase();
 //Aquí los otros atributos
 public static Clase getInstancia() {
 return Clase.instancia;
 }
 //Aquí los otros métodos de la clase
}
```

- Se crea una instancia de la clase y se coloca en un atributo estático privado.
- Se ve que este atributo tiene una única instancia (objeto) de la clase.

### Un código para el patrón “Singleton”

```
public class Clase {
 private static Clase instancia = new Clase();
 //Aquí los otros atributos
 public static Clase getInstancia() {
 return Clase.instancia;
 }
 //Aquí los otros métodos de la clase
}
```

- El método estático devuelve ese único objeto de la clase.
- Por lo tanto ejecutando `getInstancia()` obtenemos siempre el mismo objeto de la clase.

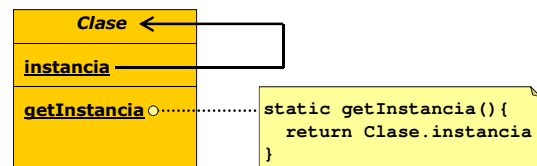
### Dicho de otra manera

```
public class Clase {
 private static Clase instancia = new Clase();
 //Aquí los otros atributos
 public static Clase getInstancia() {
 return Clase.instancia;
 }
 //Aquí los otros métodos de la clase
}
```

- Siempre que usemos `getInstancia()`, de esta clase sólo **se obtendrá un único objeto**.

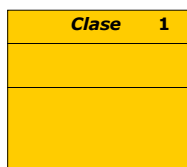
### El patron Singleton

- En UML, un miembro subrayado es un miembro estático. **La flecha es una referencia, es decir, que el atributo `instancia` de la clase `Clase`.**



### El patron Singleton

- En UML, podemos representar el patrón Singleton de forma abreviada con un 1, que indica que sólo se creará un objeto.



### Usando el patrón “Singleton”

- La clase que hemos programado se usa de la siguiente manera.

**`Clase.getInstancia()`**

Nos devuelve la instancia de esta clase.

- **¡¡Nunca se utiliza `new` sólo `getInstancia()` !!**
- Siempre que se usa (con `getInstancia()`) se obtiene la misma y única instancia de la clase.



### Veamos un ejemplo

- Supongamos que estamos programando un programa de gestión para una Universidad.
- Tendremos una clase que modelará una Universidad.
- De esta clase sólo necesitamos un único objeto ya que programamos para una única Universidad.
- Por lo tanto, podemos usar el patrón Singleton.

### La capa de dominio sería así (omitimos la instrucción package por falta de espacio)

```
public class Universidad {
 private static Universidad instancia =
 new Universidad();
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 public void setNombre(String nombre) {
 this.nombre = nombre;
 }
 public String getNombre() {
 return this.nombre;
 }
}
```

### Todo esto es de una clase normal

```
public class Universidad {
 private static Universidad instancia =
 new Universidad();
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 public void setNombre(String nombre) {
 this.nombre = nombre;
 }
 public String getNombre() {
 return this.nombre;
 }
}
```

### El resto es el patrón Singleton

```
public class Universidad {
 private static Universidad instancia =
 new Universidad();
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 public void setNombre(String nombre) {
 this.nombre = nombre;
 }
 public String getNombre() {
 return this.nombre;
 }
}
```

### Usando la clase Universidad

- La clase que hemos programado se usa de la siguiente manera.

```
Universidad.getInstancia()
```

Nos devuelve la instancia de esta clase.

- Sea como sea, siempre que se usa (con **getInstancia()**) se obtiene la misma y única instancia de la clase.

### Usando la clase Universidad

- Por ejemplo si queremos saber el nombre de la Universidad podemos hacer:

```
Universidad universidad =
 Universidad.getInstancia();
String nombreUni =
 universidad.getNombre();
```

- O de forma más abreviada.

```
String nombreUni =
 Universidad.getInstancia().getNombre();
```

- Recordemos, nunca usaremos **new**, siempre que se usa (con **getInstancia()**) se obtiene la misma y única instancia de la clase.



### Aclarando conceptos

- Como vemos, **getInstancia** es la alternativa a **new** para las clases que implementan el patrón "Singleton".
- Así, en una clase normal, para obtener un objeto con el cual poder llamar a sus métodos utilizábamos

```
new Clase()
```

- Mientras que en una clase que implementa el patrón "Singleton", la forma con la que obtenemos un objeto es

```
Clase.getInstancia()
```

### Pero hay una diferencia importante

- Usando **new** obtiene cada vez un objeto diferente.
- Usando **getInstancia** se obtiene siempre el mismo objeto.
- Esto constituye una gran ventaja, sobre todo porque disminuye el uso de recursos de forma espectacular si se usa el patrón "singleton".
- Crear un nuevo objeto requiere memoria y es una operación relativamente costosa en tiempo.
- Si sólo tenemos un objeto es un desperdicio de tiempo y memoria crear un objeto cada vez.
- Un "singleton" nos garantiza que sólo se crea un único objeto.

### Creen un proyecto en Eclipse llamado singleton e incluyan esta clase

```
package com.aurumsol.singleton.negocio;
public class ClaseNormal {
 private static int numInstancias = 0;
 public ClaseNormal() {
 ClaseNormal.numInstancias++;
 }
 public int getNumeroInstancias() {
 return ClaseNormal.numInstancias;
 }
}
```

- Como se ve es una clase de negocio normal que lo único que hace es contar el número de instancias que tiene (en el constructor).

### Creen esta clase que es la misma, pero ahora con patrón Singleton

```
package com.aurumsol.singleton.negocio;
public class ClaseSingleton {
 private static int numInstancias = 0;
 private static ClaseSingleton instancia = new ClaseSingleton();
 public static ClaseSingleton getInstancia() {
 return ClaseSingleton.instancia;
 }
 public ClaseSingleton() {
 ClaseSingleton.numInstancias++;
 }
 public int getNumeroInstancias() {
 return ClaseSingleton.numInstancias;
 }
}
```

### Incluyan esta JSP y llámenla normal.jsp

```
<%@ page
import="com.aurumsol.singleton.negocio.*" %>
<% ClaseNormal objNegocio = new ClaseNormal();
int instancias =
objNegocio.getNumeroInstancias(); %>
<html><head><title>Clase normal</title></head>
<body>
He llamado al metodo de negocio. La clase de
negocio tiene <%= instancias %> instancias
</body>
</html>
```

- Como ven, se llama a **new** cada JSP y se le pregunta el número de instancias.

### Incluyan esta JSP y llámenla singleton.jsp

```
<%@ page import =
"com.aurumsol.singleton.negocio.*" %>
<% ClaseSingleton objNegocio =
ClaseSingleton.getInstancia();
int instancias =
objNegocio.getNumeroInstancias(); %>
<html><head><title>Singleton</title></head>
<body>
He llamado al metodo de negocio. La clase de
negocio tiene <%= instancias %> instancias
</body>
</html>
```

- Como ven, se llama a **getInstancia** cada JSP y se le pregunta el número de instancias.



### Crean dos formularios HTML

- Llamados **normal.html** y **singleton.html**, cada uno de ellos con un botón que llama respectivamente a **normal.jsp** y **singleton.jsp**.
- Desplieguen.
- Ejecuten **normal.html** y hagan clic varias veces en el botón (deberán pulsar el botón de “Atrás” en el navegador).
- Ejecuten **singleton.html** y hagan clic varias veces en el botón (deberán pulsar el botón de “Atrás” en el navegador).
- ¿Qué diferencia ven entre los dos casos?

### Diferencia

- En el primer caso, se crea una instancia cada vez que se hace clic en el botón, es decir, cada vez que se hace **new**.

He llamado al metodo de negocio. La clase de negocio tiene 7 instancias

- En el segundo caso, se crea una única instancia que es la que se recupera cada vez que se se hace clic en el botón, es decir, cada vez que se hace **getInstancia()**.

He llamado al metodo de negocio. La clase de negocio tiene 1 instancias

### Hasta aquí todo bien

- El singleton se comporta como nosotros pensábamos.
- Ahora bien, cambien la segunda línea de **singleton.jsp** para que sea:

```
ClaseSingleton objNegocio =
 new ClaseSingleton();
```

- Desplieguen y ejecutan.
- ¿Qué es lo que ven?

### ;;Se crea un objeto diferente cada vez!!

- Vemos algo así.

He llamado al metodo de negocio. La clase de negocio tiene 8 instancias

- ¡¡Pero estábamos hablando de la **ClaseSingleton**, que sólo debería tener una instancia!!
- En realidad, como vemos, las clases “singleton” sólo producen un único objeto si se usa el método **getInstancia** y no el método **new**.
- ¿Quiere decir esto que tenemos que ir con cuidado para no usar **new** sino sólo **getInstancia** cuando usemos una clase “singleton”?

### Eso sería una mala política

- Dejar que sea el programador el que tenga que acordarse de no usar **new** cada vez que usa una clase Singleton, sería una fuente de errores de programación.
- Sería mucho mejor que con un Singleton NO SE PUDIERA usar **new**.
- ¿Cómo hacemos esto?

### El nuevo código del patrón Singleton

```
public class Clase {
 private static Clase instancia = new Clase();
 //Aquí los otros atributos
 public static Clase getInstancia() {
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```

- El constructor lo hacemos privado. No se puede acceder desde fuera de la clase.



### El nuevo código del patrón Singleton

```
public class Clase {
 private static Clase instancia = new Clase();
 //Aquí los otros atributos
 public static Clase getInstancia() {
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```

- No se puede hacer **new** de esta clase desde fuera de ella. La única forma de crear un objeto es usar **getInstancia**.

### En el caso de Universidad, sería así

```
public class Universidad {
 private static Universidad instancia =
 new Universidad();
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 private Universidad() {}
 public void setNombre(String nombre) {
 this.nombre = nombre;
 }
 public String getNombre() {
 return this.nombre;
 }
}
```

### Hagan privado el constructor de ClaseSingleton

```
public class ClaseSingleton {
 private static int numInstancias = 0;
 private static ClaseSingleton instancia =
 new ClaseSingleton();
 public static ClaseSingleton getInstancia() {
 return ClaseSingleton.instancia;
 }
 private ClaseSingleton() {
 ClaseSingleton.numInstancias++;
 }
 public int getNumeroInstancias() {
 return ClaseSingleton.numInstancias;
 }
}
```

### Hagan privado el constructor de ClaseSingleton

- Así aplicaremos el nuevo código.
- Vuelvan a desplegar.
- ¿Qué pasa?

### Da un mensaje de error

#### Estado HTTP 500 -

**type** Informe de Excepción

**mensaje**

**descripcion** El servidor encontró un error interno () que hizo que no pudiera rellenar este requerimiento.

**excepción**

org.apache.jasper.JasperException: No se puede compilar la clase para JSP  
Ha tenido lugar un error en la línea: 2 en el archivo jsp: /singleton.jsp  
Error de servlet generado:  
The constructor ClaseSingleton() is not visible

- No se puede compilar la clase para JSP.
- Dice "The constructor ClaseSingleton() is not visible".

### Ahora ya no tenemos que preocuparnos de que algún programador haga new

- De nuestro **singleton**, estropeando todo el diseño que habíamos preparado.
- Ya no se puede hacer **new**.
- Nuestra clase es un **singleton** y no puede ser otra cosa.



### El código que acabamos de presentar es el recomendado para el singleton

- Es el mejor para la inmensa mayoría de los casos.
- Sin embargo, hay casos especiales en los que puede ser más conveniente otro código.
- Por ejemplo, imaginemos que la creación de un objeto de la clase "singleton" tarda mucho tiempo (por ejemplo, debe acceder a archivos, etc.).
- En este código que hemos visto siempre se crea un objeto, aunque no se llame a `getInstancia` nunca.

### En ese caso, se puede usar el siguiente código

```
public class Clase {
 private static Clase instancia;
 //Aquí los otros atributos
 public static synchronized Clase getInstancia() {
 if (Clase.instancia == null) {
 Clase.instancia = new Clase();
 }
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```

### La 1ª vez que se ejecuta `getInstancia` se inicializa y se devuelve el atributo

```
public class Clase {
 private static Clase instancia;
 //Aquí los otros atributos
 public static synchronized Clase getInstancia() {
 if (Clase.instancia == null) {
 Clase.instancia = new Clase();
 }
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```

### Las otras veces sólo se devuelve el atributo, que se inicializó la primera vez

```
public class Clase {
 private static Clase instancia;
 //Aquí los otros atributos
 public static synchronized Clase getInstancia() {
 if (Clase.instancia == null) {
 Clase.instancia = new Clase();
 }
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```

### La palabra `synchronized` se usa para evitar problemas de multithreading

```
public class Clase {
 private static Clase instancia;
 //Aquí los otros atributos
 public static synchronized Clase getInstancia() {
 if (Clase.instancia == null) {
 Clase.instancia = new Clase();
 }
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```

Sin embargo, hace lento el programa si se usa mucho `getInstancia`

### Para solucionar este último problema se puede usar este código, que no explicaremos

```
public class Clase {
 private volatile static Clase instancia;
 //Aquí los otros atributos
 public static Clase getInstancia() {
 if (Clase.instancia == null) {
 synchronized (Clase.class) {
 if (Clase.instancia == null) {
 Clase.instancia = new Clase();
 }
 }
 }
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```



### Sin embargo, esto es complicarse mucho la vida

- Normalmente, nosotros usaremos la primera versión del patrón Singleton, que es buena la mayoría de las veces.

### Este es el código del patrón que usaremos

```
public class Clase {
 private static Clase instancia = new Clase();
 //Aquí los otros atributos
 public static Clase getInstancia() {
 return Clase.instancia;
 }
 private Clase() {
 //Aquí el código del constructor, si lo hay
 }
 //Aquí los otros métodos de la clase
}
```

### ¿Cómo inicializar la clase Singleton?

- La clase Singleton, a parte de tener un solo objeto, es una clase.
- Por ello, su inicialización se debe hacer como en cualquier clase: **en los constructores**.

### Por ejemplo, tenemos la clase Universidad donde setNombre no es adecuado

```
public class Universidad {
 private static Universidad instancia =
 new Universidad();
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 private Universidad() {}
 public void setNombre(String nombre) {
 this.nombre = nombre;
 }
 public String getNombre() {
 return this.nombre;
 }
}
```

**Podríamos cambiar el nombre de la Universidad más de 1 vez**

### Solución: poner la inicialización del nombre en el constructor

```
public class Universidad {
 private static Universidad instancia =
 new Universidad("Francisco Gavidia");
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 private Universidad(String nombre) {
 this.nombre = nombre;
 }
 public String getNombre() {
 return this.nombre;
 }
}
```

### Fíjense que el constructor sólo se ejecuta una vez

```
public class Universidad {
 private static Universidad instancia =
 new Universidad("Francisco Gavidia");
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 private Universidad(String nombre) {
 this.nombre = nombre;
 }
 public String getNombre() {
 return this.nombre;
 }
}
```



### La inicialización podría ser incluso más complicada

```
public class Universidad {
 private static Universidad instancia =
 new Universidad();
 private String nombre;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 private Universidad() {
 //Lee de un archivo de configuración
 //los valores iniciales de nombre
 //y otros atributos.
 }
 public String getNombre() {
 return this.nombre
 }
}
```

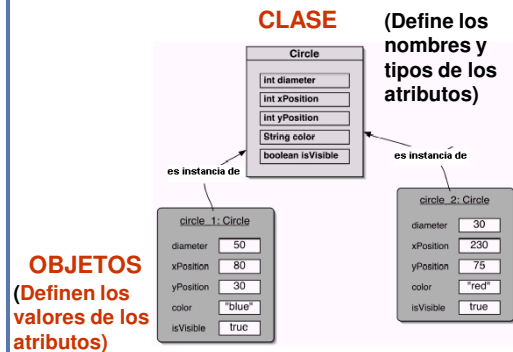
### ¿Cuándo usar el patrón Singleton?

- El patrón Singleton se usa cuando sólo hay una instancia (objeto) de una determinada clase.
- Entonces, podemos preguntarnos, ¿cuándo sólo hace falta una única instancia de una determinada clase?

### ¿Cuándo sólo debemos tener un objeto de la clase en un programa?

- Algunos casos:
- Cuando sólo hay un objeto de esa clase en la vida real (el mismo caso que hemos visto con la Universidad).
- Clases de infraestructura de las cuales sólo debe haber una: pools de threads o de conexiones, cachés, clases que manejan preferencias y configuraciones, drivers de dispositivos, etc.
- Cuando la clase no tiene atributos (de instancia): sólo es una colección de métodos. Como no tiene atributos (de instancia), no tiene sentido mantener más de un objeto de ella.

### Recordemos: Una clase y sus objetos



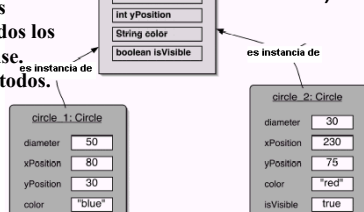
### Lo único que se diferencian las diferentes instancias de una clase

Es el valor de los atributos.

El nombre y tipo de los atributos son los mismos para todos los objetos de la clase. También los métodos.

**OBJETOS**  
(Definen los valores de los atributos)

**CLASE** (Define los nombres y tipos de los atributos)

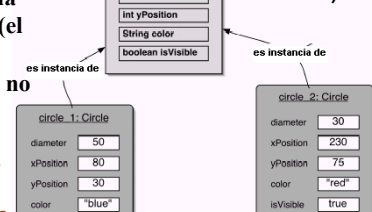


### Si no hay atributos, todos los objetos de una clase son iguales

¡Ya que la única cosa que se diferencian los objetos de una misma clase (el valor de los atributos) ya no existe!!

**OBJETOS**  
(Definen los valores de los atributos)

**CLASE** (Define los nombres y tipos de los atributos)



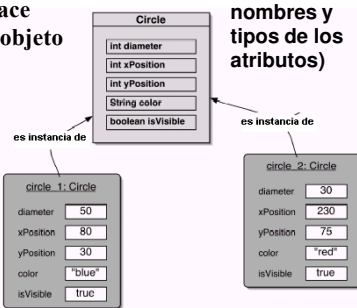


### Si no hay atributos, todos los objetos de una clase son iguales

Como todos son iguales, sólo hace falta tener un objeto por clase.

#### CLASE

(Define los nombres y tipos de los atributos)



**OBJETOS**  
(Definen los valores de los atributos)

### En estos casos, debemos tener un solo objeto

- Cuando sólo hay un objeto de esa clase en la vida real (el mismo caso que hemos visto con la Universidad).
- Clases de infraestructura de las cuales sólo debe haber una: pools de threads o de conexiones, cachés, clases que manejan preferencias y configuraciones, drivers de dispositivos, etc.
- Cuando la clase no tiene atributos (de instancia): sólo es una colección de métodos. Como no tiene atributos (de instancia), no tiene sentido mantener más de un objeto de ella.

### ¿Debemos aplicar el patrón Singleton en esos casos?

- Depende. Hay otros patrones que también nos permiten tener un objeto de cada clase.
- Ya veremos que, la mayoría de las veces, hay patrones más adecuados que el Singleton para lograr este efecto.
- Los singleton deberían usarse de forma esporádica, pues tienen problemas importantes (que veremos más adelante).
- Un error común de los programadores novatos es usar mucho el Singleton.
- Cuando tenemos muchos singleton en nuestro programa, es señal de que algo anda mal.

### Como se diseña un Singleton

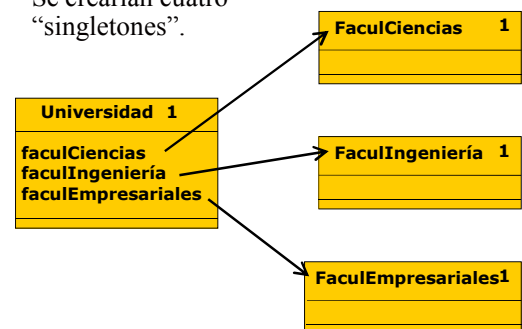
- La forma más sencilla consta de dos partes.
- 1. Se diseña la clase como si fuera normal, como si no fuera un Singleton.
- 2. Se añade el código del Singleton
  - Se hace privado el constructor.
  - Se añaden el atributo y método estáticos relacionados con instancia.

### Ejemplo

- Queremos modelar una única Universidad. Esta tiene una única facultad de ingeniería, una única facultad de ciencias y una única facultad de empresariales.

### Un programador novato puede hacer algo así

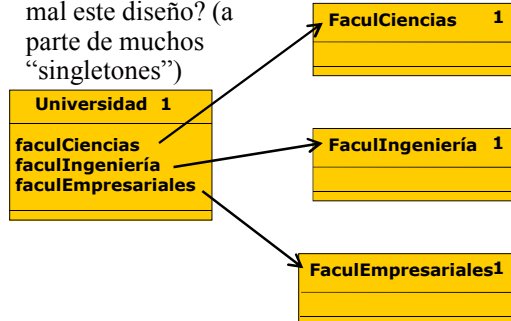
- Se crearían cuatro “singletons”.





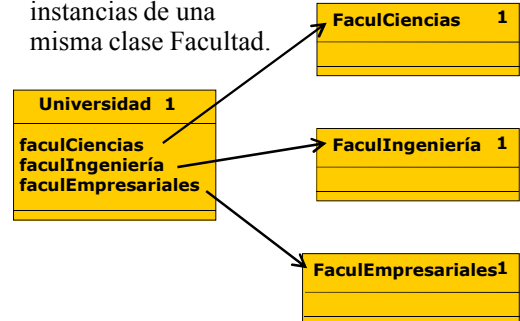
### Cuando hay muchos “singletons” es indicio de que el diseño puede estar mal

- Pregunta: ¿En qué está mal este diseño? (a parte de muchos “singletons”)



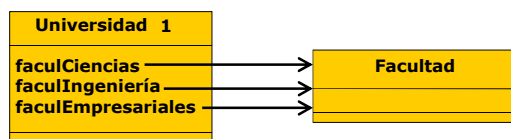
### Respuesta: FaculCiencias y similares no parecen clases

- Parece mejor que sean instancias de una misma clase Facultad.



### Un mejor diseño

- Es mejor que las tres facultades sean instancias de una misma clase Facultad.



### El código sería así

```

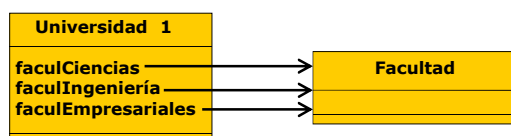
public class Universidad {
 private static Universidad instancia =
 new Universidad();
 private Facultad faculCiencias;
 private Facultad faculIngeniería;
 private Facultad faculEmpresariales;
 public static Universidad getInstancia() {
 return Universidad.instancia;
 }
 //Aquí más métodos
}

```

Fíjense que sólo hay una Facultad de Ciencias, una de Ingeniería y una de Empresariales.  
Pregunta: ¿Por qué?

### Solución

- Sólo hay una Universidad (por el Singleton). Esta Universidad tiene sólo una facultad de cada clase, por los atributos.
- Tenemos una única facultad de cada clase con un solo Singleton.



- Conclusión: hay que limitar el uso de singleton.

### Recordemos: en estos casos, debemos tener un solo objeto

- Cuando sólo hay un objeto de esa clase en la vida real (el mismo caso que hemos visto con la Universidad).
- Clases de infraestructura de las cuales sólo debe haber una: pools de threads o de conexiones, cachés, clases que manejan preferencias y configuraciones, drivers de dispositivos, etc.
- Cuando la clase no tiene atributos (de instancia): sólo es una colección de métodos. Como no tiene atributos (de instancia), no tiene sentido mantener más de un objeto de ella.



### Pregunta

- ¿Conocen alguna clase de nuestro programa que no tenga atributos (sólo sea una colección de métodos) y que, por lo tanto, sólo deba tener un objeto de esa clase?

### Las clases de negocio y datos (aquí están abreviadas) no tienen atributos

```
public class NgcAuto {
 public boolean crearNuevo(String nombre, String
 marca, int anyoFabricacion, int valor){
 }
 public int obtieneValorAutosMarcaAnyo (String
 marca, int anyo){
 }
}
public class DAOAuto {
 public boolean existeNombre(String nombre){
 }
 public void guardarNuevo (Auto auto) {
 }
 public List obtenerAutosMarca (String marca) {
 }
}
```

### Solución

- Todas las clases de las capas de negocio y de datos carecen de atributos.
- En realidad, sólo necesitamos un objeto de cada una.
- ¿Debemos implementar un patrón Singleton para cada una?

### Realmente, no

- Esto llenaría de “Singletones” nuestra aplicación y sería síntoma de mal diseño.
- Hay otras formas más adecuadas de hacer que las clases de negocio y de datos sólo tengan un objeto de cada clase.
- Las veremos más adelante.

### Una aplicación legítima del patrón Singleton

- Hay otros patrones que usan o se combinan con el patrón Singleton.
- En este caso, es bueno aplicarlo.

### Ejercicio

- Queremos hacer una aplicación para un banco. Sobre el mismo banco nos interesa su nombre, los activos, el número de acciones y el valor de cada acción. Queremos calcular la “ratio” valor de las acciones/activos, lo que es un importante indicador bursátil. Queremos que sólo haya un banco en nuestra aplicación.



### \*Escriure solució de l'exercici de banc

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- **3.4. Composición.**
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### Tipos de datos en Java

Se pueden dividir en dos clases:

- **Tipos de valor.** Sus miembros son valores (es decir, simples datos).  
Por ejemplo, una variable `int` contiene 5, que es un valor (un simple dato).
- **Tipos de entidad.** Sus miembros son entidades (es decir, conjuntos de datos que suelen modelar objetos reales).  
Por ejemplo, una variable de tipo `Auto` es una entidad, un conjunto de datos que modela un objeto real.

### Tipos de valor y tipos de entidad

- **Tipos de valor.** Los tipos primitivos, **String**, **BigDecimal**, **BigInteger** y muchos más.
- **Tipos de entidad.** Todos los otros tipos de datos en Java.

### ¿Cómo distinguirlos?

- **Tipos de valor.** A los miembros de un tipo de valor no se les pueden asociar propiedades (atributos).  
Así, un entero 5 es un entero 5 y no hay ninguna propiedad que asociarle. Lo mismo con el **String** "Hola a todos".
- **Tipos de entidad.** A los miembros de un tipo de entidad se les puede asociar propiedades (atributos).  
Así, a un objeto **Auto** se le puede asociar una infinidad de propiedades: velocidad, etc.

### ¿Cómo distinguirlos?

- **Tipos de valor.** Los miembros de un tipo de valor con los mismos datos son intercambiables, pues no tienen una identidad propia.  
Así, un entero 5 se puede intercambiar por cualquier otro entero 5. Un **String** "Hola a todos" es intercambiable por cualquier otro **String** "Hola a todos".
- **Tipos de entidad.** Los miembros de un tipo de entidad con los mismos datos NO son intercambiables, pues tienen identidad propia.  
Así, dos objetos **Auto** con los mismos atributos no son intercambiables, pues podrían corresponder a dos autos diferentes con las mismas características.



### Composición

- Es cuando hay atributos de una clase que no son de tipo de valor.
- Dicho de otra manera, es cuando una clase tiene como atributo una entidad (es decir, un objeto de tipo de entidad).

### Vamos a verlo con un ejemplo

- Queremos una clase que represente una Empresa. De una empresa sólo nos interesa su razón social y su facturación anual (en dólares sin centavos).

### La clase sería así

```
package com.aurumsol.planilla.dominio;
public class Empresa {
 public String razonSocial;
 public int facturacionAnual;
 public int getFacturacionMensual() {
 return (this.facturacionAnual/12)
 }
}
```

- Esta clase NO tiene composición, pues todos sus atributos son de tipo de valor.
- Nota: Los atributos deberían ser privados y tener métodos que los accedan. Pero aquí los haremos públicos para hacer la explicación más sencilla.

### Ahora queremos una clase Empleado

- Queremos programar una clase que recoja toda la información de un empleado es decir, su nombre, sueldo y empresa. Escribir esta clase especificando sólo los atributos.
- Díctenlo en la pizarra.

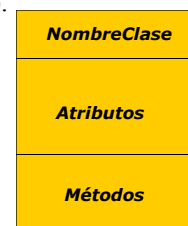
### Solución

```
package com.aurumsol.planilla;
public class Empleado {
 public String nombre;
 public int sueldo;
 public Empresa empresa;
}
```

- Esta clase SÍ tiene composición, pues hay un atributo que es una entidad.
- Recuerden que los atributos deberían ser privados.

### Notación UML

- Recordemos que una clase en UML se representa así (la parte de métodos es opcional).





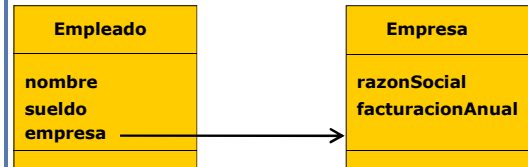
### Notación UML

- Un objeto se representa de forma similar, pero tiene la variable del objeto y además, ésta y el nombre de la clase están subrayados.



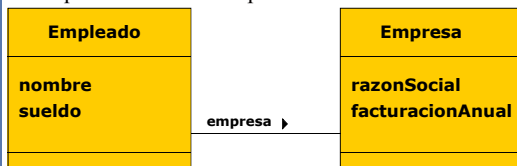
### Vamos a representar la composición así

- Esto es lógico: el atributo empresa contiene una referencia a un objeto de la clase Empresa.



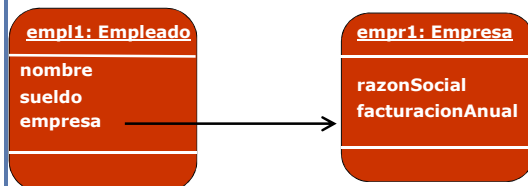
### Sin embargo, en UML se representa así

- La línea representa la composición y el atributo se pone en la línea en vez de estar con los atributos primitivos.
- Esto puede ser confuso para los principiantes, por lo que no lo usaremos por ahora.



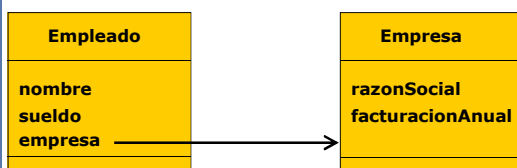
### Entre dos objetos, la notación que usaremos será esta

- El atributo empresa del objeto **empl1** contiene el objeto empl1.
- En UML también sería una línea, como entre dos clases.



### Terminología

- En una relación “muchos a uno” o “uno a muchos”, la clase que está en el uno se puede llamar “**padre**” y la que está en el muchos se puede llamar “**hijo**”.
- Así, en nuestro caso, Empresa sería el padre y Empleado sería el hijo (para cada empresa hay muchos empleados y, por simplicidad, suponemos que un empleado sólo trabaja en una empresa).



### Si empl1 es de tipo Empleado

Para acceder a su empresa empleamos

```
empl1.empresa
```

Para acceder a los atributos de la empresa del empleado **empl1**

```
empl1.empresa.facturacionAnual
```

o bien

```
Empresa emprEmpleado1 = empl1.empresa
emprEmpleado1.facturacionAnual
```

Si no sabemos si el **empl1** tiene empresa también podemos hacer

```
Empresa emprEmpleado1 = empl1.empresa
if (emprEmpleado1 != null){
 emprEmpleado1.facturacionAnual
}
```



### Lo mismo es con los métodos

Para acceder a su empresa empleamos

```
empl1.empresa
```

Para acceder a los atributos de la empresa del empleado empl1

```
empl1.empresa.getFacturacionMensual()
```

o bien

```
Empresa emprEmpleado1 = empl1.empresa
emprEmpleado1.getFacturacionMensual()
```

Si no sabemos si el empl1 tiene empresa también podemos hacer

```
Empresa emprEmpleado1 = empl1.empresa
if (emprEmpleado1 != null){
 emprEmpleado1.getFacturacionMensual()
}
```

### Uso de estas clases (1)

```
Empresa empresa1= new Empresa();
```

empresa1:Empresa

### Uso de estas clases (2)

```
Empresa empresa1= new Empresa();
empresa1.razonSocial = "Aurum";
empresa1.facturacionAnual = 100000;
```

empresa1:Empresa

razonSocial: "Aurum"  
facturacionAnual: 100000

### Uso de estas clases (3)

```
Empresa empresa1= new Empresa();
empresa1.razonSocial = "Aurum";
empresa1.facturacionAnual = 100000;
Empleado empleado1 = new Empleado();
```

empleado1:Empleado

empresa1:Empresa

razonSocial: "Aurum"  
facturacionAnual: 100000

### Uso de estas clases (4)

```
Empresa empresa1= new Empresa();
empresa1.razonSocial = "Aurum";
empresa1.facturacionAnual = 100000;
Empleado empleado1 = new Empleado();
empleado1.nombre = "Juan"
empleado1.sueldo = 1000;
```

empleado1:Empleado

nombre: "Juan"  
sueldo:1000

empresa1:Empresa

razonSocial: "Aurum"  
facturacionAnual: 100000

### Uso de estas clases (5)

```
Empresa empresa1= new Empresa();
empresa1.razonSocial = "Aurum";
empresa1.facturacionAnual = 100000;
Empleado empleado1 = new Empleado();
empleado1.nombre = "Juan"
empleado1.sueldo = 1000;
empleado1.empresa = empresa1;
```

empleado1:Empleado

nombre: "Juan"  
sueldo:1000  
empresa:

empresa1:Empresa

razonSocial: "Aurum"  
facturacionAnual: 100000



### Recordemos que la clase Libreta era

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public int getNumero(){
 return this.numero;
 }
 public int getSaldo(){
 return this.saldo;
 }
}
```

### Recordemos que la clase Libreta era

```
public void depositar(int monto){
 this.saldo += monto;
}
public boolean retirar(int monto){
 if (monto > this.saldo){
 return false;
 } else {
 this.saldo -= monto;
 return true;
 }
}
```

### Recordemos que la clase Cliente era

```
package com.aurumsol.banco.dominio;
public class Cliente {
 private String nombre;
 private int credito;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 }
 public void setCredito(int credito){
 this.credito = credito;
 }
 public String getNombre(){
 return this.nombre;
 }
 public int getCredito(){
 return this.credito;
 }
}
```

### Ejercicio

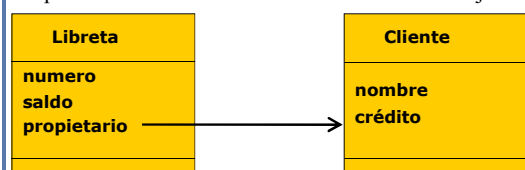
- Complementen la clase Libreta que se les ha dado para que contenga información sobre el propietario de la libreta, que es un Cliente del banco.

### Solución

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 private Cliente propietario;
 public Libreta(int numero, Cliente propie){
 this.numero = numero;
 this.saldo = 0;
 this.propietario = propie;
 }
 public Cliente getPropietario(){
 return this.propietario;
 }
}
// Todos los otros métodos igual que antes.
```

### Fijémonos que el padre es Cliente y el hijo es Libreta

- Un cliente tiene varias libretas.
- Una libreta es de sólo un cliente.
- Fijémonos que en una relación muchos a uno, quien tiene el atributo referencia es la clase Hijo.

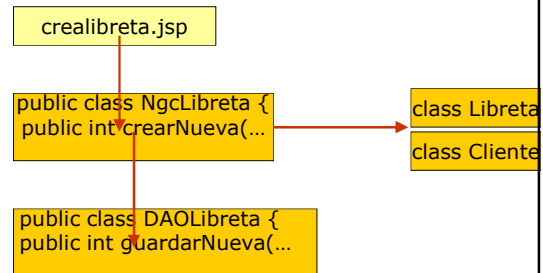




### Ejercicios obligatorios

- Preparación:
  - Creen un nuevo proyecto, llamado “ejerciciocomposicion”.
  - Importen el archivo “ejerciciocomposicion.zip” que se les proporcionará.
  - Organicenlo todo antes de encontrar
- Ejercicio:
  - Escriban el código de los métodos obtenerNombrePropietario y crearNueva de la clase NgcLibreta (primero **obtenerNombrePropietario**).
  - (Todos los otros archivos del proyecto ya están programados, incluyendo la capa de presentación).

### La estructura es la siguiente



- Importante: revisen estas clases y estos métodos antes de comenzar a programar.

### Solución

### Pero tenemos un problema

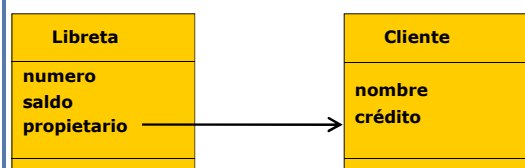
- Estos objetos con composición se deberán guardar en la base de datos y utilizar con Hibernate.
- Debemos ver cómo se implementa la composición con Hibernate y en la base de datos.
- Comenzaremos con la base de datos.

### Guardando la composición en la base de datos

- Sin embargo, debemos ver como se refleja la composición entre **Cliente** y **Libreta** en la base de datos.
- Recordemos que la base de datos es relacional y no orientada a objetos.

### Tenemos que en nuestra capa de dominio

- Tenemos que una **Libreta** tiene un atributo propietario, que es un objeto cliente.





### En las tablas, esto se traducirá en una clave foránea

libreta			
5	123	0	566

↑ Id    ↑ Número    ↑ Saldo    ↑ Propietario

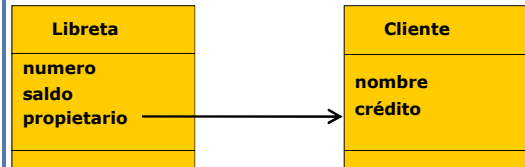
- El campo propietario de la tabla libreta tiene la clave primaria del cliente que es propietario de la libreta.

cliente		
566	Juan	1000
	...	

↑ Id    ↑ Nombre    ↑ Crédito

### Nota: Fíjense que en la programación O-O las relaciones son unidireccionales

- Van en una dirección y no en la contraria.
- Esto contrasta con la base de datos: las relaciones en el modelo relacional son bidireccionales.



### Es por eso

- Que en Java hablamos de relaciones muchos a uno y uno a muchos.
- Cuando en la base de datos simplemente son un único tipo de relación: uno a muchos.

### Simplemente tendremos que crear una clave foránea

- Esto es fácil: simplemente creen un campo **entero** en la tabla libreta que se llame propietario.
- (Es entero pues es una referencia a la clave primaria de la tabla cliente que es un entero)
- Y ya está. Con esto Hibernate, ya sabe que es una clave foránea.
- Fíjense que estamos diciendo “una clave foránea”, no “una restricción de integridad”.

### Ya hemos acabado con la base de datos

- Ahora veremos como se indica la composición en Hibernate.
- Lo único que tenemos que hacer es indicar la composición en el archivo de correspondencia de la clase hijo.

### Incluimos lo siguiente en la clase hijo (la que tiene el atributo referencia)

```
<many-to-one name="nombreAtributo"
class="nombreClaseALaQueReferencia"
column="nombreCampoClaveForánea"
access="field" lazy="false"/>
```

- many-to-one** indica que ese atributo contiene una referencia a otra clase.
- nombreClaseALaQueReferencia** debe tener un formato **nombrePaquete.nombreClase** (es la clase padre).
- nombreCampo** es el campo de la tabla en el que guardamos el atributo. Si el campo se llama igual que el atributo no hace falta.



### Teníamos (fragmento principal de Libreta.hbm.xml)

```
<hibernate-mapping>
 <class name="com.aurumsol.cursojava.banco.dominio.Libreta" table="libreta">
 <id name="id" type="int" access="field">
 <column name="id"/>
 <generator class="identity"/>
 </id>
 <property name="numero" access="field"/>
 <property name="saldo" access="field"/>
 </class>
</hibernate-mapping>
```

- Pero ahora tenemos un atributo más (**propietario**), que es una referencia a otra clase (**Cliente**). ¿Cómo lo indicamos?

### En nuestro caso

```
<many-to-one name="propietario"
 class="com.aurumsol.cursojava.banco.dominio.Cliente" column="propietario" access="field"
 lazy="false"/>
```

- El nombre del atributo es **propietario** y es una referencia a la clase **Cliente**.
- **column="propietario"** indica el campo de la tabla en el que se guardará este atributo.
- Como tiene el mismo nombre que el atributo, podríamos prescindir de él.

### Libreta.hbm.xml queda así

```
<hibernate-mapping>
 <class name="com.aurumsol.cursojava.banco.dominio.Libreta" table="libreta">
 <id name="id" type="int" access="field">
 <column name="id"/>
 <generator class="identity"/>
 </id>
 <property name="numero" access="field"/>
 <property name="saldo" access="field"/>
 <many-to-one name="propietario"
 class="com.aurumsol.cursojava.banco.dominio.Cliente" column="propietario" access="field"
 lazy="false"/>
 </class>
</hibernate-mapping>
```

- 2 atributos normales (**property**) y uno que es una referencia (**many-to-one**).

### ¿Por qué many-to-one?

- La relación entre Libreta y Cliente es de “muchos a uno”: un cliente puede tener muchas libretas. Esto es lo más usual.
- Si fuera una relación “uno a uno” (por ejemplo, si un Cliente sólo pudiera tener una sola libreta) se usaría el atributo **unique="true"** dentro del **many-to-one**. Esto generará una restricción de integridad en la base de datos. Es poco común.
- Las relaciones “uno a muchos” y “muchos a muchos” se estudiarán más adelante.

### Compruébenlo

1. Reciban el proyecto completo “composicion”.
2. En MySQL hagan que “prueba” sea el esquema por defecto (“Make default schema”).
3. Creen la estructura en la base de datos (pueden usar el código SQL que hay en el archivo createtable).
4. Revisen Libreta.hbm.xml.
5. Desplieguen el proyecto.
6. Creen una nueva libreta y obtengan el número del propietario.

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento



### Concepto de arreglo

- Un arreglo es una variable que permite guardar una colección ordenada de datos del mismo tipo.
- Puntos importantes:
  - como es una variable, tiene un **nombre**
  - la colección es **ordenada**.
  - los datos deben ser todos **del mismo tipo**.
  - el número de datos a guardar (la **longitud**) es **fijo**

30 12 57 99 -2 83 91 9

Variable arreglo **tabla** (con enteros)

### Un arreglo es similar a una sucesión de variables

- Cada dato se guarda en algo análogo a una variable que se llama elemento
- Como es parecido a una variable, podemos leer y escribir su valor.

Elemento Elemento Elemento  
30 12 57 99 -2 83 91 9

Variable arreglo **tabla**

### Escribiendo o leyendo un valor en un elemento de un arreglo

- Debemos decir qué elemento de todos queremos escribir o leer.
- Para ello, se utiliza la notación **nombreArreglo[numeroElemento]**.
- Podemos denominar:

tabla[0] tabla[1] tabla[2]  
30 12 57 99 -2 83 91 9

Variable arreglo **tabla**

### Como ven

- En Java, los elementos del arreglo **comienzan a contarse desde cero**
- Así:

tabla[0] tabla[1] tabla[2]  
30 12 57 99 -2 83 91 9

Variable arreglo **tabla**

### Escribiendo o leyendo un valor en un elemento de un arreglo

- Es como una variable.
- Para leer (recuperar) el valor **nombreArreglo[numeroElemento]**
- Para escribir (asignar) el valor **nombreArreglo[numeroElemento] = expresión**

tabla[0] tabla[1] tabla[2]  
30 12 57 99 -2 83 91 9

Variable arreglo **tabla**

### Escribiendo o leyendo un valor en un elemento de un arreglo

- Para leer (recuperar) el valor **tabla[1]**, por ejemplo, **out.println(tabla[1])**
- Para escribir (asignar) el valor **tabla[1] = 30**

tabla[0] tabla[1] tabla[2]  
30 12 57 99 -2 83 91 9

Variable arreglo **tabla**



### Índice

- Es el número de elemento (o de posición) con el que accedemos a los datos.

- Aparece entre corchetes. `tabla[5]`

<code>tabla[0]</code>	<code>tabla[2]</code>							
<code>tabla[1]</code>								
0	1	2	3	4	5	6	7	
30	12	57	99	-2	83	91	9	

Variable arreglo `tabla`

### La longitud de un arreglo

- Es el número de elementos que tiene.
- Es fija. Una vez definida no se puede cambiar.

El arreglo `tabla` tiene una longitud 8.  
Tiene 8 elementos.  
Cabén 8 datos **y ni uno más**.

30	12	57	99	-2	83	91	9
----	----	----	----	----	----	----	---

Variable arreglo `tabla`

### Tipos de arreglo en Java

- Como hemos dicho, un arreglo tiene todos sus elementos de un mismo tipo.
- Así, hay arreglos de enteros, de objetos **Empleado**, de objetos **Auto**, de objetos de cualquier tipo.

### Tipos de arreglo en Java

- Un arreglo tiene todos sus elementos de un mismo tipo.
- El tipo del arreglo depende del tipo de datos de sus elementos, con la notación **TipoElementos[]**
- El tipo de un arreglo de enteros es `int[]`
- El tipo de un arreglo de Strings es `String[]`
- El tipo de un arreglo de empleados es `Empleado[]`
- El tipo de un arreglo de autos es `Auto[]`

### Definición de un arreglo (1)

- En 2 Pasos (como cualquier otro objeto)
- 1. Se **declara una variable referencia** de la clase de arreglo que queremos definir.

```
TipoElemento[] varArreglo;
```

```
int[] tabla;
String[] nombres;
Empleado[] fabrica;
```

### Definición de un arreglo (2)

- 2. Se **crea un objeto de clase arreglo** con el constructor **new**.

```
varArreglo = new TipoElemento[longitud];
```

```
tabla = new int[10];
nombres = new String[20];
fabrica = new Empleado[100];
```



### Los dos pasos se pueden hacer en una sola línea

- Como en cualquier otro objeto.

```
int[] tabla = new int[10];
String[] nombres = new String[20];
Empleado[] fabrica = new Empleado[100];
```

Recordemos que la longitud se fija cuando se define el arreglo y **NO SE PUEDE CAMBIAR**

### Buena práctica: declarar tamaño del arreglo como constante

- Así, declararemos los atributos

```
private static final MAX_TABLA = 10;
private static final MAX_NOMBRES = 10;
private static final MAX_EMPLEADOS = 100;
```

Y crearemos los arreglos:

```
int[] tabla = new int[MAX_TABLA];
String[] nombres = new String[MAX_NOMBRES];
Empleado[] fabrica = new Empleado[MAX_EMPLEADOS];
```

Así mejoramos la legibilidad y la mantenibilidad.

### Otra forma de declaración de arreglo

- Se puede declarar el arreglo sin **new** si se especifican los elementos que contiene con la sintaxis:

```
nombreArreglo = {Lista valores}
```

- Los valores se separan por comas. Por ejemplo:

```
String[] continentes = {"América",
"Europa", "Asia", "Africa",
"Oceanía"};
```

### El atributo length

- Indica la longitud del arreglo (el número de elementos) del arreglo, que no puede cambiarse (es de sólo lectura).

- Así

```
tabla.length devuelve 10
nombres.length devuelve 20
fabrica.length devuelve 100
```

### El atributo length

- Como el arreglo empieza desde cero, las posiciones (índice) del arreglo van desde 0 a **length-1**

- Así

```
tabla[0] a tabla[9]
nombres[0] a nombres[19]
fabrica[0] a fabrica[99]
```

### Ejemplo de uso de un arreglo

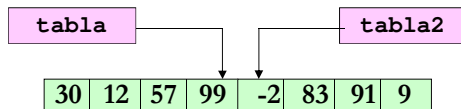
```
public class Fabrica {
 private static final int MAX_EMP = 1000;
 private Empleado[] staff = new Empleado[MAX_EMP];
 private int numEmpleados = 0;
 public void agregarEmpleado(Empleado empleado) {
 this.staff[numEmpleados] = empleado;
 numEmpleados++;
 }
 public int totalCostoPlanilla() {
 int totalPlanilla = 0;
 for (i=0; i<this.numEmpleados; i++) {
 totalPlanilla += staff[i].getSueldo()
 }
 return totalPlanilla;
 }
}
```



### Los arreglos son unos objetos

- Por ello, para copiarlos no basta con copiar sus variables referencia.

```
tabla2 = tabla; // Copia la referencia pero sigue siendo el mismo arreglo.
```



### Para copiar dos arreglos, se utiliza el método **arraycopy**

- Método **static** de la clase **System**. Sirve para copiar arreglos o fragmentos de arreglo.

- Sintaxis:

```
System.arraycopy(a1, pos1, a2, pos2, nc)
```

- Lee **nc** elementos del arreglo **a1**, comenzando en la posición **pos1**.
- Copia estos elementos a la posición **pos2** del arreglo **a2**.

### Recorriendo un arreglo

- Hay una forma obvia, que hemos usado hasta ahora.

```
for (int i =1; i<arreglo.length;i++){
 //Tratar el elemento arreglo[i]
}
```

Hay otra forma alternativa ("para cada elemento del arreglo", donde *Tipo* es el tipo del elemento).

```
for (Tipo elemento: arreglo){
 //Tratar el elemento elemento
}
```

### Ejemplo, imprimir todos los nombres de un arreglo de empleados (empleados)

- Hay una forma obvia, que hemos usado hasta ahora.

```
for (int i =1; i<empleados.length;i++){
 out.print(empleados[i].getNombre());
}
```

Hay otra forma alternativa "para cada elemento del arreglo".

```
for (Empleado empActual: empleados){
 out.print(empActual.getNombre());
}
```

### Mi gozo en un pozo

- La forma alternativa (que es la más legible) sólo está disponible en Java 1.5 y posteriores.
- La versión que usamos de Eclipse (la 3.0) no soporta las novedades de Java 1.5. Para ello, debemos esperar a la versión 3.1. que está en beta (y que es mucho más rápida).

### Arreglos: unos objetos extraños

- Los objetos arreglo tienen **sintaxis diferente** de los restantes objetos Java para inicializarse, acceder a ellos, etc.
- Esta sintaxis es reminiscencia de otros lenguajes.
- Son objetos, pero muy extraños.
- Podemos verlos como objetos o como algo diferente, según nos convenga.



### Arreglos de varias dimensiones

- A veces, necesitamos arreglos de varias dimensiones que pueden accederse por varios índices.

Variable arreglo **coord**

	0	1	2	3	4	5	6	7
0	30	12	57	99	-2	83	91	9
1	65	79	40	64	36	81	19	7
2	30	12	57	99	-2	83	91	9
3	30	12	57	99	-2	83	91	9
4	30	12	57	99	-2	83	91	9

### Arreglos de varias dimensiones

- Se escriben como arreglos de arreglos.
- Con 2 dimensiones. 2 pares de corchetes
- Con n dimensiones. N pares de corchetes

Variable arreglo **coord**

	0	1	2	3	4	5	6	7
0	30	12	57	99	-2	83	91	9
1	65	79	40	64	36	81	19	7
2	30	12	57	99	-2	83	91	9
3	30	12	57	99	-2	83	91	9
4	30	12	57	99	-2	83	91	9

### Arreglos de varias dimensiones

```
int[][] coord = new int[5][8];
coord[2][3] = 40;
out.print(coord[4][5]);
```

Variable arreglo **coord**

	0	1	2	3	4	5	6	7
0	30	12	57	99	-2	83	91	9
1	65	79	40	64	36	81	19	7
2	30	12	57	99	-2	83	91	9
3	30	12	57	99	-2	83	91	9
4	30	12	57	99	-2	83	91	9

### Ejercicio

- Queremos modelar un curso de Universidad. De cada curso nos interesa su nombre, su número de alumnos y sus propios alumnos.
- Queremos añadir alumnos, calcular la nota media y devolver cuantos alumnos tenemos.
- Se supone que un curso no puede tener más de 200 alumnos.

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### Un ejemplo

- Somos “Renta-un-Carro”, una empresa de alquiler de vehículos y queremos manejar las compras de nuestra empresa.
- Compramos muchos productos pero nuestras principales compras son autos y tanques (depósitos) de gasolina.
- Tenemos la clase **Auto** y la clase **TanqueGasolina**.



### La clase Auto (1)

```
package com.aurumsol.cursojava.auto.dominio;
public class Auto {
 private String marca;
 private int anyo, valor;
 public Auto (String marca, int anyo, int valor){
 this.marca = marca; this.anyo = anyo;
 this.valor = valor;
 }
 public String getMarca(){
 return this.marca;
 }
 public int getValor(){
 return this.valor;
 }
 public double getIVAAPagar(){
 return this.valor*0.13;
 }
}
```

### La clase Auto (2)

```
public double getValorAmortizadoEnAnyo
(int anyoCalc){
 double valorAmortizado;
 if (anyoCalc < this.anyo){
 valorAmortizado = 0.0;
 } else {
 valorAmortizado =
 this.valor-0.05*this.valor *
 (anyoCalc - this.anyo);
 if (valorAmortizado < 0.0){
 valorAmortizado = 0.0;
 }
 }
 return valorAmortizado;
}
```

### Nota

- En la clase Auto, los valores 0.13 y 0.05 deberian haber sido constantes para facilitar modificaciones futuras.
- Sin embargo, por brevedad, se ha prescindido de esto.
- Lo mismo se hará con la clase TanqueGasolina, que estamos a punto de ver.

### La clase TanqueGasolina

```
public class TanqueGasolina {
 private int litros;
 private int precioLitro;
 public TanqueGasolina (int litros,int pr){
 this.litros = litros;
 this.precioLitro = pr;
 }
 public int getValor(){
 return this.litros * this.precioLitro;
 }
 public int getIVAAPagar{
 return this.litros*this.precioLitro*0.13;
 }
}
```

### La clase TanqueGasolina

```
public boolean vaciar (int litrosVaciar){
 if (litrosVaciar>litros){
 return false;
 } else {
 this.litros -= litrosVaciar;
 return true;
 }
}
```

### Las clases resumidas

- Aquí tenemos las clases resumidas en formato UML. Fijense que ahora no me interesan los atributos ni los constructores.

Auto	TanqueGasolina
getMarca getValor getIVAAPagar getValorAmortizadoAnyo	getValor getIVAAPagar vaciar



### Estas clases tienen métodos comunes

- Estos métodos comunes devuelven el valor del objeto y el IVA que debemos pagar por él.

Auto	TanqueGasolina
getMarca getValor getIVAAPagar getValorAmortizadoAnyo	getValor getIVAAPagar vaciar

### Estos métodos tienen el mismo nombre y hacen lo mismo

- Aunque su implementación sea diferente. Por ejemplo,

```
public int getValor() {
 return this.valor;
}
```

**AUTO**

```
public int getValor() {
 return this.litros * this.precioLitro;
}
```

**TANQUEGASOLINA**

Auto	TanqueGasolina
getMarca getValor getIVAAPagar getValorAmortizadoAnyo	getValor getIVAAPagar vaciar

### Estos métodos tienen el mismo nombre y hacen lo mismo

- Java tiene una construcción para estos casos en los que diferentes clases comparten los mismos métodos.
- Esta construcción se llama interfaz.

Auto	TanqueGasolina
getMarca getValor getIVAAPagar getValorAmortizadoAnyo	getValor getIVAAPagar vaciar

### La construcción interface o interfaz

```
public interface NombreInterface{
 Definiciones de métodos de instancia
}
```

- Donde la definición de métodos es la cabecera (la primera línea) de métodos de instancia que contiene:
  - El nombre del método.
  - Los parámetros y resultados.
- Y nada más.

### ¿Qué no tiene una interfaz?

```
public interface NombreInterface{
 Definiciones de métodos de instancia
}
```

- Una interfaz no tiene atributos de ningún tipo.
- Una interfaz no tiene métodos estáticos.
- Una interfaz no tiene la implementación de los métodos de instancia. Sólo la definición: la cabecera (el nombre del método, los parámetros y resultados).

### Dicho de otra manera

```
public interface NombreInterface{
 Definiciones de métodos de instancia
}
```

- Una interfaz es una colección de definiciones de métodos de instancia.
- Una interfaz no tiene constructores (ni el implícito) por lo tanto no se puede hacer **new**.
- Una interfaz no es una clase sino una colección de definiciones.**



### Veamos un ejemplo

```
public interface ProductoComprado {
 public int getValor();
 public int getIVAAPagar();
}
```

- Una interfaz es una colección de definiciones de métodos de instancia.
- Fijense que no hay atributos, métodos estáticos, ni implementación de estos métodos.

### Auto y TanqueGasolina tienen los mismos métodos que la interfaz.

```
public interface ProductoComprado {
 public int getValor();
 public int getIVAAPagar();
}
```

Auto	TanqueGasolina
getMarca getValor getIVAAPagar getValorAmortizadoAnyo	getValor getIVAAPagar vaciar

### Auto y TanqueGasolina tienen la implementación de los métodos que define la interfaz

```
public interface ProductoComprado {
 public int getValor();
 public int getIVAAPagar();
}
```

```
public int getValor(){
 return this.valor;
}
```

**AUTO**

```
public int getValor(){
 return this.litros *this.precioLitro;
}
```

**TANQUEGASOLINA**

### Decimos que Auto y TanqueGasolina implementan la interfaz ProductoComprado

```
public interface ProductoComprado {
 public int getValor();
 public int getIVAAPagar();
}
```

```
public int getValor(){
 return this.valor;
}
```

**AUTO**

```
public int getValor(){
 return this.litros *this.precioLitro;
}
```

**TANQUEGASOLINA**

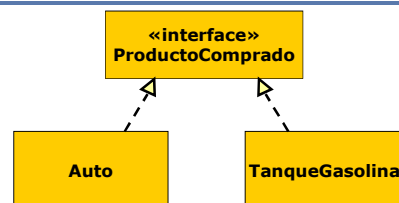
### Esto se escribe así

```
public interface ProductoComprado {
 public int getValor();
 public int getIVAAPagar();
}
```

```
public class Auto implements ProductoComprado {
 //Aquí todos los atributos y métodos
}

public class TanqueGasolina implements ProductoComprado{
 //Aquí todos los atributos y métodos
}
```

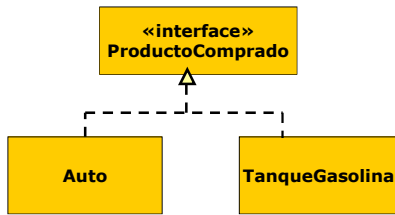
### En UML se representa así



- La interfaz se marca con el calificador «interface» para marcar que no es una clase.
- Si una clase implementa una interfaz tiene una flecha para ella con trazo discontinuo y punta triangular.

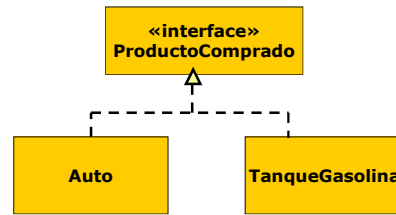


**También se puede representar así,  
por comodidad**



- Son dos flechas, no sólo una flecha.

**Terminología**



- Se dice que las clases implementan la interfaz.
- Se dice que las clases son implementaciones (de la interfaz).
- Se puede decir que la interfaz es supertipo de las clases.

**¿Y para que nos sirve todo esto?  
¿Para escribir más?**

```
public interface ProductoComprado {
 public int getValor();
 public int getIVAAPagar();
}
```

```
public class Auto implements ProductoComprado {
 //Aquí todos los atributos y métodos
}
```

```
public class TanqueGasolina implements ProductoComprado{
 //Aquí todos los atributos y métodos
}
```

**Un hecho importante**

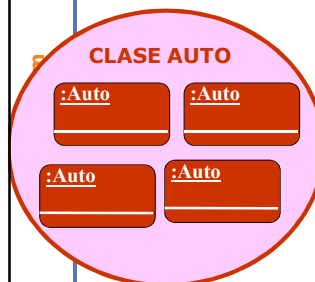
- **Todas las interfaces son tipos de datos.**
- Los tipos de datos en Java pueden ser:
  - Tipos primitivos.
  - Clases.
  - Interfaces.
- Por ejemplo, podemos tener variables que tengan como tipo la interfaz. Así:

```
ProductoComprado producto1;
```

**Otro hecho importante**

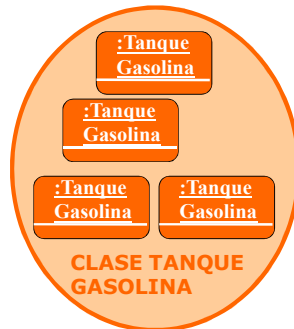
- Los valores de este tipo de datos son los objetos de las clases que implementan la interfaz.
- Así, en nuestro caso, los valores del tipo **ProductoComprado** son objetos de las clases **Auto** y **TanqueGasolina**.

**La clase `Auto` contiene todos los objetos `auto`**





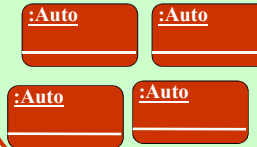
La clase **TanqueGasolina** contiene todos los objetos tanque de gasolina



La interfaz **ProductoComprado** tiene tanto los objetos auto como los objeto tanques

#### INTERFAZ PRODUCTOCOMPRADO

##### CLASE AUTO



##### CLASE TANQUE GASOLINA



#### Dicho de otra manera

- Ahora, los objetos de la clase **Auto**:
  - Son del tipo de datos **Auto**.
  - Son del tipo de datos **ProductoComprado**.
- De la misma manera, los objetos de la clase **TanqueGasolina**:
  - Son del tipo de datos **TanqueGasolina**.
  - Son del tipo de datos **ProductoComprado**.
- Los objetos de una clase:
  - Son del tipo de datos de una clase.
  - Son del tipo de datos de las interfaces que implementa la clase.

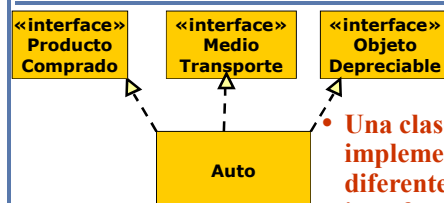
#### A esto se le llama polimorfismo

- Polimorfismo** (del griego “muchas formas”) quiere decir que un mismo objeto puede ser de varios tipos de datos diferentes.
- En nuestro caso, un objeto **Auto** puede ser de los tipos **Auto** (su clase) y **ProductoComprado** (la interfaz que implementa su clase).
- Las interfaces sirven para implementar el polimorfismo.

#### Dicho de otra manera

- Polimorfismo** quiere decir que un mismo objeto puede tener varios roles diferentes en una aplicación.
- En nuestro caso, un objeto **Auto** puede tener el rol de un auto o el rol de un producto comprado.
- Las clases e interfaces son roles diferentes que pueden tomar los objetos.

#### No hay límites al polimorfismo



- Una clase puede implementar diferentes interfaces.
- En código sería:
 

```

public class Auto implements
 ProductoComprado, MedioTransporte,
 ObjetoDepreciable {
 //Aquí el código de la Clase
}

```



### El polimorfismo es necesario

- En la vida real, un mismo objeto puede considerarse con varias perspectivas y finalidades.
- Así, un auto es un producto comprado para el Departamento de Compras, es un medio de transporte para el Departamento de Logística y es un objeto depreciable para el Departamento de Contabilidad.
- Si un objeto sólo pudiera contemplarse de una forma, nuestro programa no sería lo suficientemente flexible.
- Vamos a verlo con ejemplos concretos.

### Uso de una interfaz

```
Auto auto1 = new Auto("Toyota", 1999, 2000);
ProductoComprado autoComoProducto = auto1;
```

- Aunque se puede hacer en una sola línea.

```
ProductoComprado autoComoProducto =
 new Auto("Toyota", 1999, 2000);
```

- Como vemos, los objetos **Auto** se pueden tratar como **ProductoComprado**, sin ningún problema.

### Lo mismo pasa con un método

- Supongamos que tenemos el método

```
public void agregaCompra(ProductoComprado p1)
```

- Se puede hacer lo siguiente.

```
Auto auto1 = new Auto("Toyota", 1999, 2000);
agregaCompra(auto1);
```

- Como vemos, un método que acepta un **ProductoComprado**, como parámetro puede aceptar un **auto** sin ningún problema, ya que un **auto** se puede ver como **ProductoComprado** (pues **Auto** implementa **ProductoComprado**).

### Supongamos una clase así

```
public class LibroIVACompras {
 private static final MAX_COMPRAS = 100;
 ProductoComprado[] compras =
 new ProductoComprado[MAX_COMPRAS];
 int numCompras = 0;
 public void agregaCompra (ProductoComprado pr){
 compras[numCompras] = pr;
 numCompras++;
 }
 public int getTotalIVAAPagar(){
 int totalIVA = 0; ProductoComprado producto;
 for (int i=0,i<this.numCompras, i++){
 producto = compras[i];
 totalIVA += producto.getIVAAPagar();
 }
 return totalIVA;
 }
}
```

### Ejemplo (1)

```
Auto auto1 = new Auto("Toyota", 1999, 2000);
TanqueGasolina tanquel = new Auto(500, 0.50);
LibroIVACompras libroIVA = new LibroIVACompras();
libroIVA.agregaCompra(auto1);
libroIVA.agregaCompra(tanquel);
out.print(libroIVA.getTotalIVAAPagar());
out.print(auto1.getValorAmortizadoEnAnyo());
boolean exito = tanquel.vaciar(200);
```

- Se crean tres objetos: un auto (**auto1**), un tanque (**tanquel**) y un libro de IVA.

### Ejemplo (2)

```
Auto auto1 = new Auto("Toyota", 1999, 2000);
TanqueGasolina tanquel = new Auto(500, 0.50);
LibroIVACompras libroIVA = new LibroIVACompras();
libroIVA.agregaCompra(auto1);
libroIVA.agregaCompra(tanquel);
out.print(libroIVA.getTotalIVAAPagar());
out.print(auto1.getValorAmortizadoEnAnyo());
boolean exito = tanquel.vaciar(200);
```

- El auto y el tanque se agregan al libro de compras.
- Aquí los dos se tratan como **ProductoComprado**.



### agregaCompra los trata como ProductoComprado

```
public class LibroIVACompras {
 private static final MAX_COMPRAS = 100;
 ProductoComprado[] compras =
 new ProductoComprado[MAX_COMPRAS];
 int numCompras = 0;
 public void agregaCompra (ProductoComprado pr){
 compras[numCompras] = pr;
 numCompras++;
 }
 public int getTotalIVAAPagar(){
 int totalIVA = 0; ProductoComprado producto;
 for (int i=0,i<this.numCompras, i++){
 producto = compras[i];
 totalIVA += producto.getIVAAPagar();
 }
 return totalIVA;
 }
}
```

**El parámetro del método es ProductoComprado y el array es de ProductoComprado.**

### Toda la clase trata ProductoComprado

```
public class LibroIVACompras {
 private static final MAX_COMPRAS = 100;
 ProductoComprado[] compras =
 new ProductoComprado[MAX_COMPRAS];
 int numCompras = 0;
 public void agregaCompra (ProductoComprado pr){
 compras[numCompras] = pr;
 numCompras++;
 }
 public int getTotalIVAAPagar(){
 int totalIVA = 0; ProductoComprado producto;
 for (int i=0,i<this.numCompras, i++){
 producto = compras[i];
 totalIVA += producto.getIVAAPagar();
 }
 return totalIVA;
 }
}
```

**Esta clase no sabe nada de autos ni de tanques sólo de ProductoComprado.**

### Ejemplo (3)

```
Auto auto1 = new Auto("Toyota",1999,2000);
TanqueGasolina tanque1 = new Auto(500,0.50);
LibroIVACompras libroIVA = new LibroIVACompras();
libroIVA.agregaCompra(auto1);
libroIVA.agregaCompra(tanque1);
out.print(libroIVA.getTotalIVAAPagar());
out.print(auto1.getValorAmortizadoEnAnyo());
boolean exito = tanque1.vaciar(200);
```

- También cuando calculamos el total de IVA estamos tratando al auto y al tanque como ProductoComprado.
- Los dos están en el arreglo como productos comprados y son tratados así por el método.

### Los 2 se tratan como ProductoComprado

```
public class LibroIVACompras {
 private static final MAX_COMPRAS = 100;
 ProductoComprado[] compras =
 new ProductoComprado[MAX_COMPRAS];
 int numCompras = 0;
 public void agregaCompra (ProductoComprado pr){
 compras[numCompras] = pr;
 numCompras++;
 }
 public int getTotalIVAAPagar(){
 int totalIVA = 0; ProductoComprado producto;
 for (int i=0,i<this.numCompras, i++){
 producto = compras[i];
 totalIVA += producto.getIVAAPagar();
 }
 return totalIVA;
 }
}
```

**Esta clase no sabe nada de autos ni de tanques sólo de ProductoComprado.**

### Ejemplo (4)

```
Auto auto1 = new Auto("Toyota",1999,2000);
TanqueGasolina tanque1 = new Auto(500,0.50);
LibroIVACompras libroIVA = new LibroIVACompras();
libroIVA.agregaCompra(auto1);
libroIVA.agregaCompra(tanque1);
out.print(libroIVA.getTotalIVAAPagar());
out.print(auto1.getValorAmortizadoEnAnyo());
boolean exito = tanque1.vaciar(200);
```

- Sin embargo, aunque los hemos tratado como ProductoComprado, ahora los volvemos a tratar respectivamente como auto y tanque.

### Esta es la ventaja de las interfaces

- Permiten que un mismo objeto se pueda tratar de varias formas según lo necesitemos.
- O dicho en griego, permiten el polimorfismo.



### Las interfaces simplifican la programación

- Imaginemos que no tenemos la interfaz.
- El libro del IVA debería tener dos arreglos: uno de autos y uno de tanques.
- Todo el código se debería duplicar.

### Nueva clase sin interfaces (1)

```
public class LibroIVACompras {
 private static final MAX_AUTOS = 50;
 private static final MAX_TANQUES = 50;
 Auto[] autos = new Auto[MAX_AUTOS];
 TanqueGasolina[] tanques =
 new TanqueGasolina[MAX_TANQUE];
 int numAutos = 0; int numTanques = 0;
```

- Duplicamos los atributos, ahora deberemos tener un juego de constante, arreglo y entero para auto y para tanques.

### Nueva clase sin interfaces (2)

```
public void agregaCompra (Auto auto) {
 autos[numAutos] = auto;
 numAutos++;
}
public void agregaCompra (TanqueGasolina tanque) {
 tanques[numTanques] = tanque;
 numTanques++;
}
```

- Debemos tener dos métodos **agregaCompra** pues ahora el parámetro es de dos tipos diferentes. Además, lo ponemos en arreglos diferentes.
- Para más simplicidad, utilizamos la sobrecarga.

### Nueva clase sin interfaces (3)

```
public int getTotalIVAAPagar() {
 int totalIVA = 0; Auto auto;
 for (int i=0, i<this.numAutos, i++){
 auto = autos[i];
 totalIVA += auto.getIVAAPagar();
 }
 for (int i=0, i<this.numTanques, i++){
 tanques = tanques[i];
 totalIVA += tanques.getIVAAPagar();
 }
 return totalIVA;
}
```

- Ahora tenemos dos bucles para sumar el IVA

### Se ve claramente

- Sin interfaces hemos hecho el doble de trabajo que con interfaces.
- Las interfaces simplifican la programación.

### Pregunta

- ¿Y no hubiera sido mejor definir una sola clase **ProductoComprado** y olvidarnos de **Auto**, **TanqueGasolina** y las interfaces?



### Solución

- No. Esto no permitiría tener métodos específicos, como **vaciar** (que sólo se aplica a un tanque) o **getValorAmortizadoAnyo** (que sólo se aplica a un auto, pues la gasolina no se deprecia).
- Lo bonito de las interfaces es que podemos decidir el nivel de detalle que queremos.
  - Si un auto lo queremos ver específicamente como auto, lo podemos hacer.
  - Si queremos verlo generalmente como una compra, también lo podemos hacer.

### Se ve claramente

- Sin interfaces hemos hecho el doble de trabajo que con interfaces.
- Las interfaces simplifican la programación, como hemos visto.
- Pero esta no es su principal virtud.
- La principal virtud es que las interfaces simplifican el mantenimiento.

### Las interfaces simplifican el mantenimiento

- Después de un año, nuestra empresa de alquiler de vehículos, ha decidido ampliar operaciones.
- Puede alquilar buses, camiones y bicicletas. Para eso necesita comprar esos tres tipos de vehículos así como tanques de diesel.
- Tenemos que modificar el programa para que introduzca esos cambios ("mantenimiento"). Debemos hacer dos cosas:
  - Programar las clases **Bus**, **Camion**, **Bicicleta** y **TanqueDiesel**. Suponemos que ya están programadas.
  - Adaptar la clase **LibroIVCompras**. Esto es lo que haremos ahora.

### Primera opción: Hacerlo sin interfaces



- Cada una de las clases está separada.
- Vamos a adaptar la clase **LibroIva** para estas nuevas compras.

### Nueva clase sin interfaces (1)

```
public class LibroIVCompras {
 private static final MAX_AUTOS = 50;
 private static final MAX_BICICLETAS = 50;
 private static final MAX_BUSES = 50;
 private static final MAX_CAMIONES = 50;
 private static final MAX_TANQUES = 50;
 private static final MAX_TANDIESEL = 50;
 Auto[] autos = new Auto[MAX_AUTOS];
 Bicicleta[] bicis = new
 Bicicleta[MAX_BICICLETAS];
 Bus[] buses = new Bus[MAX_BUSES];
 Camion[] camiones = new Camion[MAX_AUTOS];
 TanqueGasolina[] tanques =
 new TanqueGasolina[MAX_TANQUES];
 TanqueDiesel[] tanDiesels =
 new TanqueDiesel[MAX_TANDIESEL];
 int numAutos = 0; int numTanques = 0;
 int numBicis = 0; int numBuses = 0;
 int numCamiones = 0; int numTanDiesels = 0;
```

### Nueva clase sin interfaces (2)

```
public void agregaCompra (Auto auto) {
 autos[numAutos] = auto;
 numAutos++;
}
public void agregaCompra (Bicicleta bici) {
 bicis[numBicis] = bici;
 numBicis++;
}
public void agregaCompra (Bus bus) {
 buses[numBuses] = bus;
 numBuses++;
}
```

- Todo el código se multiplica y se hace inmanejable.



### Nueva clase sin interfaces (3)

```
public void agregaCompra (Camion camion){
 camiones[numCamiones] = camion;
 numCamiones++;
}

public void agregaCompra (TanqueGasolina tanque){
 tanques[numTanques] = tanque;
 numTanques++;
}

public void agregaCompra (TanqueDiesel tanDiesel){
 tanDiesels[numTanDiesels] = tanDiesel;
 numTanDiesels++;
}
```

### Nueva clase sin interfaces (4)

```
public int getTotalIVAAPagar(){
 int totalIVA = 0; Auto auto;
 for (int i=0,i<this.numAutos, i++){
 auto = autos[i];
 totalIVA += auto.getIVAAPagar();
 }
 for (int i=0,i<this.numBicis, i++){
 bici = bicis[i];
 totalIVA += bici.getIVAAPagar();
 }
 for (int i=0,i<this.numBuses, i++){
 bus = buses[i];
 totalIVA += bus.getIVAAPagar();
 }
}
```

- Ahora tenemos que recorrer todos los arreglos.

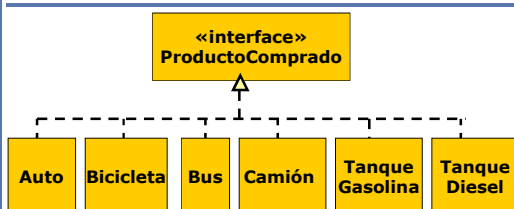
### Nueva clase sin interfaces (5)

```
for (int i=0,i<this.numCamiones, i++){
 camion = camiones[i];
 totalIVA += camion.getIVAAPagar();
}
for (int i=0,i<this.numTanques, i++){
 tanque = tanques[i];
 totalIVA += tanque.getIVAAPagar();
}
for (int i=0,i<this.numTanDiesels, i++){
 tanDiesel = tanques[i];
 totalIVA += tanDiesel.getIVAAPagar();
}
return totalIVA;
}
```

### ¿Y cada vez que tenemos que hacer mantenimiento debemos tener esta agonía?

- Veamos la segunda opción.

### Segunda opción: Hacerlo con interfaces



- Ahora simplemente todas las clases implementan la interfaz **ProductoComprado**.
- Ahora vamos a adaptar la clase **LibroIva** para las nuevas clases.

### Ya está.

- Ya hemos acabado, pues no debemos hacer nada.
- La clase **LibroIvaCompras** tal como estaba antes ya sirve para las nuevas clases, sin cambiar una coma.
- No debemos hacer ningún cambio en esta clase.



### Esta es la clase original y ya nos sirve

```
public class LibroIVACompras {
 private static final MAX_COMPRAS = 100;
 ProductoComprado[] compras =
 new ProductoComprado[MAX_COMPRAS];
 int numCompras = 0;
 public void agregaCompra (ProductoComprado pr){
 compras[numCompras] = pr;
 numCompras++;
 }
 public int getTotalIVAAPagar(){
 int totalIVA = 0; ProductoComprado producto;
 for (int i=0, i<this.numCompras, i++){
 producto = compras[i];
 totalIVA += producto.getIVAAPagar();
 }
 return totalIVA;
 }
}
```

### Hemos visto

- Las interfaces simplifican la programación.
- Las interfaces simplifican el mantenimiento.
- Las interfaces simplifican el código, lo hacen más legible y sencillo.
- Esto nos lleva a una conclusión, que es un mandamiento del mantenimiento.

### Mandamiento del mantenimiento

*Intentarás programar con supertipos.*

- Es decir, con interfaces.



### Intentarás programar con supertipos

- Cuando estemos programando con clases, preguntémonos “¿No estaría mejor poner una interfaz aquí? ¿No sería más flexible?”.
- Esto no quiere decir que no usemos clases. Hay que ver en cada punto cuál es la mejor opción.



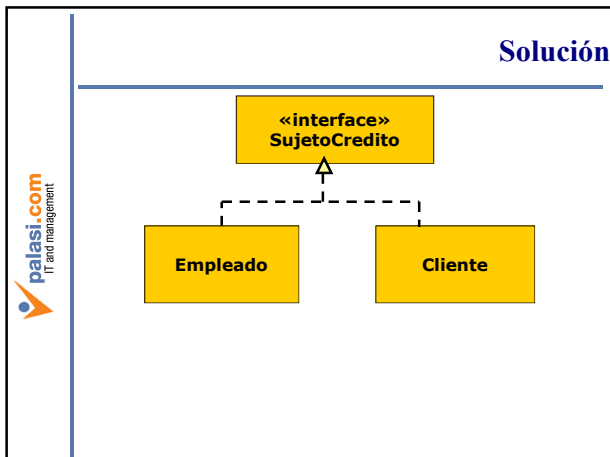
### Una guía para saber cuando una interfaz es mejor

- Si se necesita polimorfismo, es decir, si se necesita:
  - Tratar un mismo objeto de varias formas diferentes (**Auto** como auto o como producto comprado).
  - Tratar instancias de diferentes clases con la misma manera (**Auto** y **TanqueGasolina** como producto comprado).
- Si se cree que esa parte puede variar en el futuro.
  - Por ejemplo, en **LibroIva**, los productos comprados pueden variar en el futuro. Por eso, es bueno usar una interfaz.

### Ejercicio

- Queremos modelar los clientes y los empleados de un banco. De los clientes, nos interesa su nombre y el monto de sus activos. De los empleados nos interesa su nombre y su sueldo.
- Además, tanto empleados como clientes son sujetos de crédito. El límite de crédito para los clientes es el 5% de sus activos y para los empleados es el 10% de su sueldo. La tasa de interés varía para cada persona.
- Para cada sujeto de crédito queremos obtener su límite de crédito, fijar su tasa de interés y recuperar su tasa de interés.





**Solución (1)**

```

public interface SujetoCredito {
 public void setTipoInteres(int tipo);
 public int getTipoInteres();
 public int getLimiteCredito();
}

```

- La interfaz tendrá estos métodos

**Solución (2)**

```

public class Empleado implements SujetoCredito {
 private static final int TASA_CREDITO = 10;
 private String nombre; private int sueldo;
 private int tipoInteres;
 public Empleado(String nombre, int sueldo){
 this.nombre = nombre;this.sueldo = sueldo;
 this.tipoInteres = 0;
 }// Aquí set y get de sueldo y get de nombre
 public void setTipoInteres(int tipo){
 this.tipoInteres = tipo;
 }
 public int getTipoInteres(){
 return this.tipoInteres;
 }
 public int getLimiteCredito(){
 return (Empleado.TASA_CREDITO *
 this.sueldo)/100;}
}

```

**Solución (3)**

```

public class Cliente implements SujetoCredito {
 private static final int TASA_CREDITO = 5;
 private String razonSocial;
 private int activo; private int tipoInteres;
 public Cliente(String razon, int activo){
 this.razonSocial = razon;
 this.activo = activo;this.tipoInteres = 0;
 }// Aquí set y get de activo y get de nombre
 public void setTipoInteres(int tipo){
 this.tipoInteres = tipo;
 }
 public int getTipoInteres(){
 return this.tipoInteres;
 }
 public int getLimiteCredito(){
 return (Cliente.TASA_CREDITO *
 this.activo)/100; }
}

```

**Ejercicio**

- Crear una clase **InformacionSujetosCredito** que contenga todos los sujetos de crédito de nuestro banco. Queremos un método para añadir un sujeto de crédito y otro para recuperar aquel sujeto que tenga mayor límite de crédito.

**Solucion**

```

public class InformacionSujetosCredito {
 private static final MAX_SUJETOS = 100000;
 SujetoCredito[] sujetos =
 new SujetoCredito[MAX_SUJETOS];
 int numSujetos = 0;
 public void agregaSujetos(SujetoCredito suj){
 sujetos[numSujetos] = suj;
 numSumjetos++;
 }
 public SujetoCredito getMaximoSujeto(){
 int maxLimite = 0;
 SujetoCredito maxSujeto = null;
 for (int i=0,i<this.numSujetos, i++){
 if (sujetos[i].getLimiteCredito()> maxLimite){
 maxSujeto = sujetos[i];
 maxLimite = maxSujeto.getLimiteCredito();
 }
 }
 return maxSujeto
 }
}

```



**Nota**

- Fíjense que, al devolver el sujeto con más límite de crédito, también estamos devolviendo al mismo tiempo el mayor límite de crédito. Basta con hacer:

```
SujetoCredito maxSujCredito =
informacion.getMaximoSujeto();
int maxLimiteCredito =
maxSujCredito.getLimiteCredito();
```

Objeto de la clase  
InformacionSujetosCredito

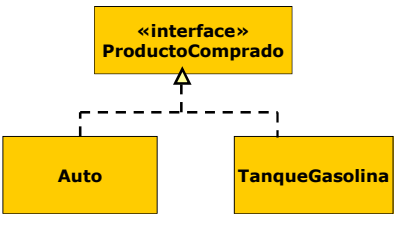
**Nota**

- Fíjense que, en este fragmento de crédito, yo no me preocupo si el sujeto de crédito es un empleado o un cliente. No me interesa. Eso es lo bueno del polimorfismo, que puedo ver las cosas con el detalle que me interesa.

```
SujetoCredito maxSujCredito =
informacion.getMaximoSujeto();
int maxLimiteCredito =
maxSujCredito.getLimiteCredito();
```

Objeto de la clase  
InformacionSujetosCredito

**Conversión de tipos entre interfaces**



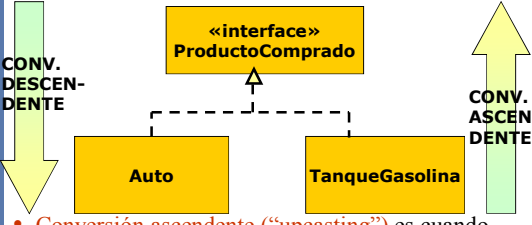
```

graph BT
 Auto --|> ProductoComprado
 TanqueGasolina --|> ProductoComprado
 style ProductoComprado fill:#fff,stroke:#333,stroke-width:1px
 style Auto fill:#fff,stroke:#333,stroke-width:1px
 style TanqueGasolina fill:#fff,stroke:#333,stroke-width:1px

```

- En inglés, se llama "Casting".

**Conversión de tipos entre interfaces**



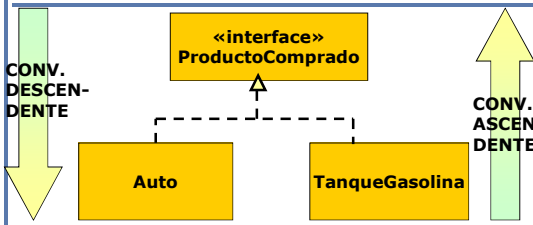
```

graph BT
 Auto --|> ProductoComprado
 TanqueGasolina --|> ProductoComprado
 style ProductoComprado fill:#fff,stroke:#333,stroke-width:1px
 style Auto fill:#fff,stroke:#333,stroke-width:1px
 style TanqueGasolina fill:#fff,stroke:#333,stroke-width:1px
 subgraph Conversion
 direction TB
 D[CONV. DESCENDENTE] --> Auto
 A[CONV. ASCENDENTE] --> TanqueGasolina
 end

```

- Conversión ascendente ("upcasting") es cuando transformamos un objeto de tipo clase a tipo interfaz.
- Conversión descendente ("downcasting") es cuando transformamos un objeto de tipo interfaz a tipo clase.

**La conversión ascendente Java la realiza automáticamente**



```

graph BT
 Auto --|> ProductoComprado
 TanqueGasolina --|> ProductoComprado
 style ProductoComprado fill:#fff,stroke:#333,stroke-width:1px
 style Auto fill:#fff,stroke:#333,stroke-width:1px
 style TanqueGasolina fill:#fff,stroke:#333,stroke-width:1px
 subgraph Conversion
 direction TB
 D[CONV. DESCENDENTE] --> Auto
 A[CONV. ASCENDENTE] --> TanqueGasolina
 end

```

- No hace falta escribir ningún código especial.
- Veamos unos ejemplos.

**Conversión ascendente**

```
Auto auto1 = new Auto("Toyota", 1999, 2000);
ProductoComprado autoComoProducto = auto1;
```

- Aunque se puede hacer en una sola línea.

```
ProductoComprado autoComoProducto =
new Auto("Toyota", 1999, 2000);
```

- Como vemos, los objetos **Auto** se pueden tratar como **ProductoComprado**, sin ningún problema.
- Es decir, la conversión entre **Auto** y **ProductoComprado** (ascendente) se realiza automáticamente.



### Lo mismo pasa con un método

- Supongamos que tenemos el método

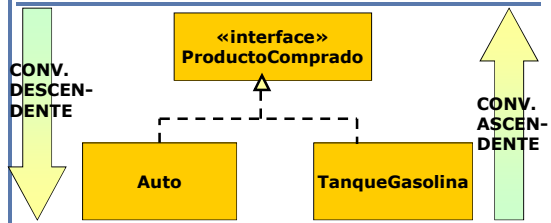
```
public void agregaCompra(ProductoComprado p1)
```

- Se puede hacer lo siguiente.

```
Auto auto1 = new Auto("Toyota", 1999, 2000);
agregaCompra(auto1);
```

- Como vemos, un método que acepta un **ProductoComprado**, como parámetro puede aceptar un auto sin ningún problema.
- La conversión entre **Auto** y **ProductoComprado** se realiza automáticamente, pues es ascendente.

### La conversión descendente no se realiza automáticamente



- Se necesita el operador de “casting”, que se usa así:

```
(TipoClase)expresionDeTipoInterfaz
```

### Conversión descendente

```
Auto auto1 = (Auto) autoComoProducto
```

- Donde **autoComoProducto** es de tipo **ProductoComprado**.
- Es decir, la conversión entre **Auto** y **ProductoComprado** (descendente) necesita un operador de casting.
- En el caso de no poner el operador de casting, se produce un error de compilación.

### Lo mismo pasa con un método

- Supongamos que tenemos el método

```
public void agregaAuto(Auto a1)
```

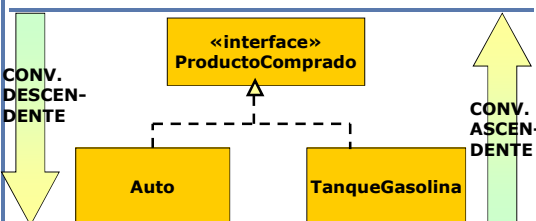
- Se puede hacer lo siguiente.

```
Auto auto1 = (Auto) autoComoProducto
agregaAuto(auto1);
```

```
agregaAuto((Auto) autoComoProducto);
```

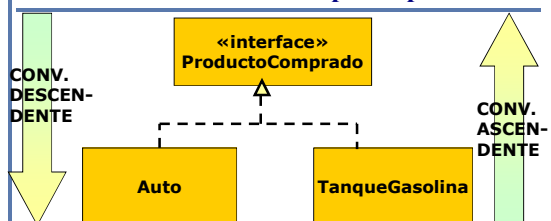
- La conversión descendente siempre necesita del operador de casting.

### Conversión de tipos entre interfaces



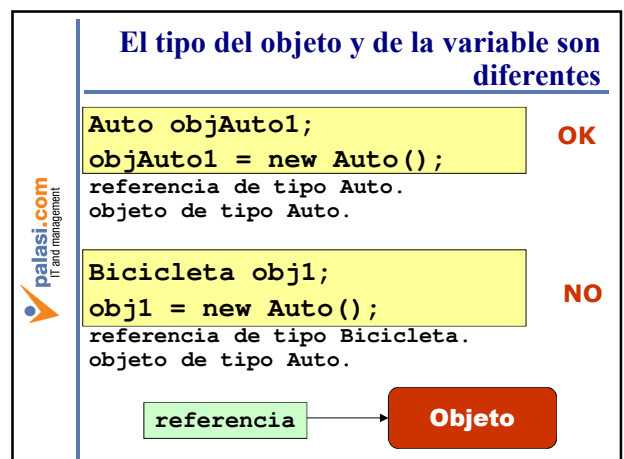
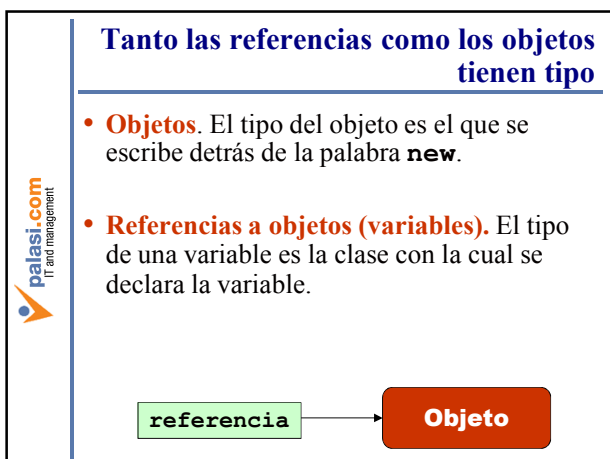
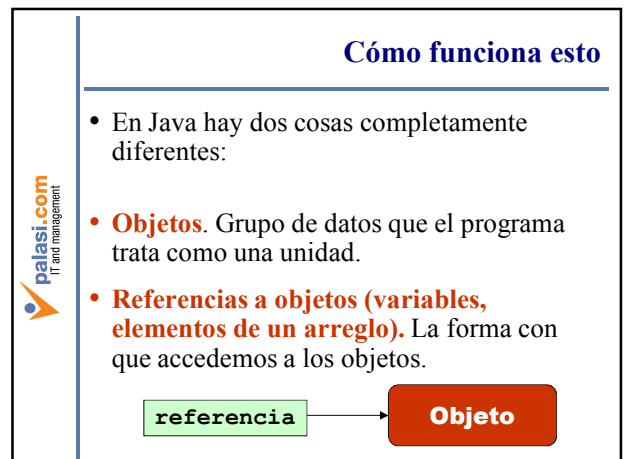
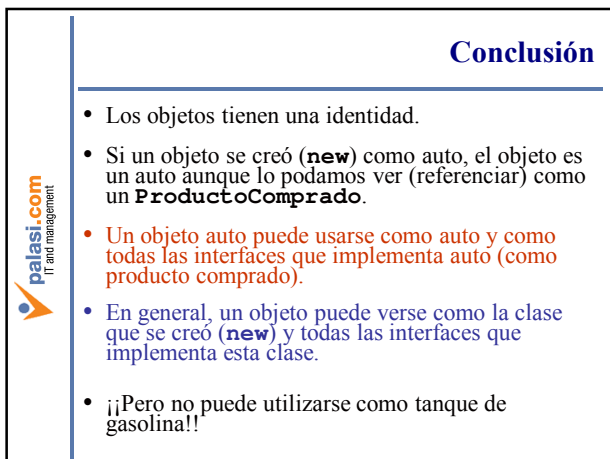
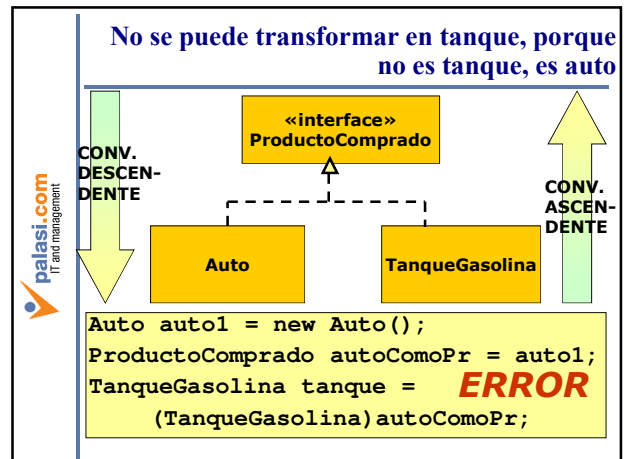
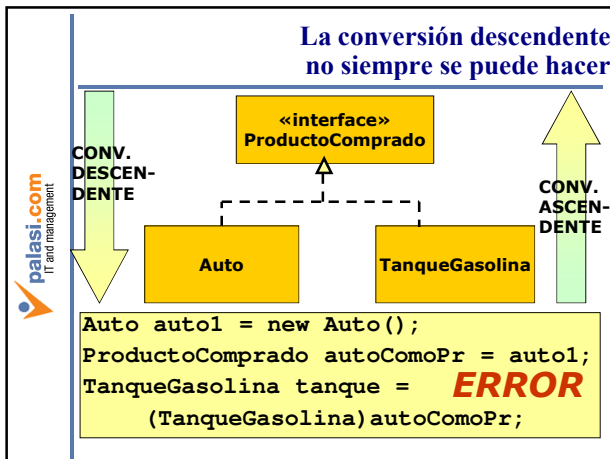
- La conversión ascendente (“upcasting”) siempre se puede hacer. Siempre podemos transformar un **Auto** en un **ProductoComprado**, pues **Auto** implementa esta interfaz.
- Conversión descendente (“downcasting”) no siempre se puede hacer.

### La conversión descendente no siempre se puede hacer



```
Auto auto1 = new Auto();
ProductoComprado autoComoPr = auto1;
Auto auto2 = (Auto) autoComoPr; OK
```



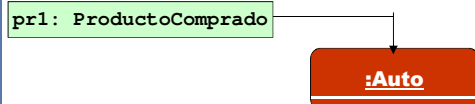




### En un fragmento de código así

```
ProductoComprado pr1 =
new Auto("Toyota", 1999, 2000);
referencia de tipo ProductoComprado.
objeto de tipo Auto.
```

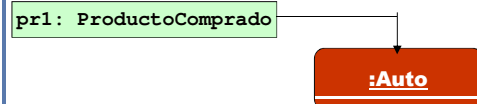
Los objetos siempre son de tipo clase. No pueden ser de tipo interfaz porque no se puede hacer **new** de la interfaz. Las referencias pueden ser de tipo clase o interfaz.



### En un fragmento de código así

```
ProductoComprado pr1 =
new Auto("Toyota", 1999, 2000);
referencia de tipo ProductoComprado.
objeto de tipo Auto.
```

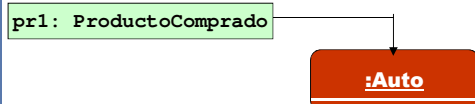
Como los objetos son de tipo clase, sabemos siempre que implementación de un método vamos a ejecutar. Así, si hacemos **pr1.getValor** sabemos que es de tipo **Auto** y no de tipo **TanqueGasolina**.



### En un fragmento de código así

```
ProductoComprado pr1 =
new Auto("Toyota", 1999, 2000);
referencia de tipo ProductoComprado.
objeto de tipo Auto.
```

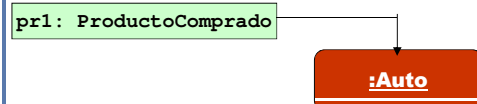
Las referencias pueden ser del tipo del objeto o del tipo de las interfaces que implementa el objeto.



### En un fragmento de código así

```
ProductoComprado pr1 =
new Auto("Toyota", 1999, 2000);
referencia de tipo ProductoComprado.
objeto de tipo Auto.
```

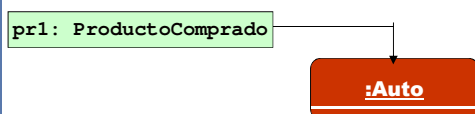
En este caso, está bien, pues la referencia es de tipo **ProductoComprado**, que es una interfaz que implementa el tipo del objeto (**Auto**).



### La conversión descendente no siempre se puede hacer

```
ProductoComprado pr1 = new
Auto();
TanqueGasolina tanque =
(TanqueGasolina) autoComoPr;
```

**Error: El objeto no puede convertirse en tanque de gasolina pues TanqueGasolina no es una interfaz que implementa Auto**



### Resumen

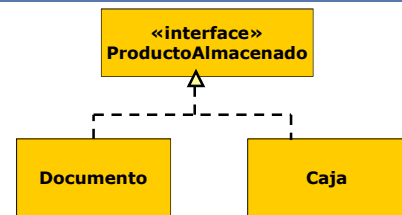
- Si un objeto se creó (**new**) como auto, el objeto es un auto aunque lo podamos ver (referenciar) como un **ProductoComprado**.
- Un objeto auto puede usarse como auto y como todas las interfaces que implementa auto (como producto comprado).
- En general, un objeto puede verse como la clase que se creó (**new**) y todas las interfaces que implementa esta clase.



### Ejercicio

- Somos una bodega de material. Almacenamos tanto documentos como cajas cúbicas. De cada documento nos interesa su nombre, número de páginas y espacio que ocupa cada página (en metros cúbicos). De cada caja, nos interesa su identificador (entero), la longitud de su lado (el volumen de un cubo es el cubo de su lado).
- Queremos definir una interfaz ProductoAlmacenado que nos indique el espacio que ocupa tanto las cajas como los documentos.

### Solución



### Solución

```

public interface ProductoAlmacenado {
 public double getEspacioQueOcupa();
}

public class Caja implements ProductoAlmacenado{
 private String iden; private double lado;
 public Caja(String iden, double lado){
 this.iden = iden; this.lado = lado;
 }
 public double getEspacioQueOcupa(){
 return this.lado * this.lado * this.lado;
 // o bien Math.pow(lado,3)
 }
}

```

### Solución

```

public class Documento implements
 ProductoAlmacenado{
 private String nombre;
 private int paginas;
 private double espacioPagina;
 public Documento(String nombre, int paginas,
 double espacioPagina){
 this.nombre = nombre; this.paginas =paginas;
 this.espacioPagina = espacioPagina;
 }
 public double getEspacioQueOcupa(){
 return this.paginas * this.espacioPagina;
 }
}

```

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### Colección

- Es un objeto que agrupa múltiples **elementos** (objetos) en una sola unidad.
- Así, colecciones pueden ser listas, conjuntos, etc.
- Recordemos, por ejemplo, las listas que agrupaban diferentes elementos en una sola lista.
  - La lista sería la colección.
  - Los elementos son los objetos que contiene.
- Las colecciones son estructuras de datos.



### Las colecciones en Java se dividen en diferentes tipos

- **Listas (lists).** Son colecciones ordenadas que pueden contener elementos duplicados. Cada elemento tiene una posición dentro de la lista. Por ejemplo, la lista de los diez primeros países del mundo.
- **Conjuntos (sets).** Son colecciones que no contienen elementos duplicados. Es decir, no se puede repetir el mismo objeto en un conjunto. Por ejemplo, los empleados de una empresa.
- **Correspondencias (maps).** Son colecciones que hacen corresponder claves y valores. Cada clave se corresponde a un valor como máximo. Por ejemplo, una tabla que hace corresponder una empresa con su registro fiscal.

### Cada uno de estos tipos tiene una interfaz

- A las listas se les asocia la interfaz **List**.
- A los conjuntos se les asocia la interfaz **Set**.
- A las correspondencias se les asocia la interfaz **Map**.

### Las listas

- Clases que implementan la interfaz **List**. Son colecciones ordenadas en las que pueden haber elementos repetidos.
- No tienen límite de elementos (al contrario de los arreglos). Siempre se puede añadir un elemento más.
- Las listas comienzan desde cero.

#### Posiciones

0	1	2	3	4	5
Micro- soft	Dell	Oracle	Sun	BEA	IBM

### Las listas

- Nota: en el gráfico anterior se veía una lista con **Strings** porque es mucho más gráfico. Pero sería mejor una lista con objetos (en nuestro caso, objetos **Empresa**).

#### Posiciones

0	1	2	3	4	5
<b>:Empre sa</b> Microsoft	<b>:Empre sa</b> Dell	<b>:Empre sa</b> Oracle	<b>:Empre sa</b> Sun	<b>:Empre sa</b> BEA	<b>:Empre sa</b> IBM

### Las listas

- Hay muchas clases que implementan la interfaz **List**. Veamos las más usuales:
- **ArrayList.** Lista implementada en un arreglo. Es la adecuada en la mayoría de los casos por ser la más rápida casi siempre.
- **LinkedList.** Se implementa como una lista enlazada. Podría ser adecuada en listas en las que se añaden muchos elementos al principio de la lista o se borran muchos elementos en el interior.

### Algunos métodos (no todos) de la interfaz List

- Si **l** es de tipo lista, **e** es 1 elemento y **p** es 1 posición
- **l.add(e)** Añade el elemento **e** al final de la lista
- **l.add(p,e)** Añade el elemen. **e** en la posición **p**
- **l.isEmpty()** Dice si la lista está vacía.
- **(Clase) l.get(p)** Obtiene el objeto que está en la posición **p** (comienzan de 0) y con la clase **Clase**
- **l.size()** Obtiene el número de elementos de la lista.
- **l.clear()** Vacía la lista.
- **l.contains(e)** Da **true** si la lista **l** contiene el elemento **e**.
- **l.indexOf(e)** Da la primera posición en la que aparece el elemento **e** en la lista **l** (-1 si no está).
- **l.remove(p)** Borra el elemento en la posición **p**.
- **l.remove(e)** Borra la primera aparición de **e**.



### A parte de los métodos de la interfaz **List**

- **ArrayList** y **LinkedList** tienen algunos métodos específicos cada uno de ellos, pero aquí no nos interesan.
- Sólo nos interesan los métodos de la interfaz **List**.

### Un uso de la lista

```
List lista = new ArrayList();
lista.add(objeto1);
lista.add(objeto2);
out.print(lista.size());
```

- Fijéense que no se puede hacer **new List()** porque **List** es 1 interfaz, no 1 clase.
- Fijémonos que el **ArrayList** sólo lo usamos para el **new**.
- En lo demás, sólo usamos la interfaz **List** y, por lo tanto, los métodos de **List**.

### Un uso de la lista

```
List lista = new ArrayList();
lista.add(objeto1);
lista.add(objeto2);
out.print(lista.size());
```

- La verdad es que el hecho de que sea **ArrayList()** sólo nos interesa para que sea más rápida.
- En el resto del programa, sólo nos interesa que es una lista. Y no me importa para programarla que esté implementada en un arreglo.

### Este es un uso muy adecuado de las interfaces

```
List lista = new ArrayList();
lista.add(objeto1);
lista.add(objeto2);
out.print(lista.size());
```

- Todos los detalles de implementación se guardan en la clase.
- La interfaz no se ocupa de los aspectos de implementación sino sólo de cómo se usa el objeto.
- Siempre usamos las interfaces en nuestro programa, así hacemos el programa independiente de la implementación.
- Así, si vemos que el **ArrayList** no es lo suficientemente rápido, cambiamos el **new** por un **LinkedList** (sólo una línea) y el resto del programa sigue igual.

### Las interfaces son algo muy conveniente

- Imaginemos que creo una clase **PalasiList** propia, que implementa la interfaz **List**.
- Ahora puedo usar mi clase con millones de clases de todo el mundo que trabajan con la interfaz **List**.

### Es por eso que el mandamiento del mantenimiento dice

*Intentarás programar con supertipos.*

- Es decir, con interfaces.
- Pues la interfaz nos permite cambiar la implementación (si lo necesitamos) sin necesidad de cambiar el programa (excepto el **new**).





### Recordemos una guía para saber cuando una interfaz es mejor

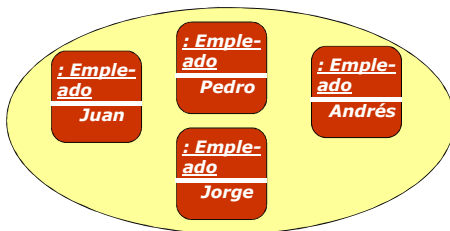
- Si se necesita polimorfismo, es decir, si se necesita:
  - Tratar instancias de diferentes clases con la misma manera (**Auto** y **TanqueGasolina** como producto comprado).
  - En nuestro caso **LinkedList** y **ArrayList**.
- Si se cree que esa parte puede variar en el futuro.
  - Por ejemplo, en nuestro programa, puede cambiar **ArrayList** por **LinkedList**.

### Conclusión

- La interfaz **List** es un uso modélico de las interfaces.
- Lo mismo con las interfaces **Set** y **Map**, pero no lo veremos con tanto detalle.

### Los conjuntos

- Clases que implementan la interfaz **Set**. Son colecciones en las que no puede haber elementos repetidos. No tienen límite de elementos.



### Los conjuntos

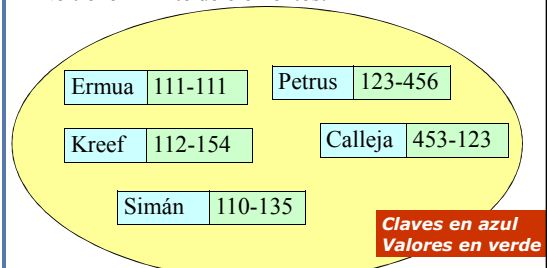
- Hay muchas clases que implementan la interfaz **Set**. Veamos las más usuales:
- **HashSet**. Conjunto implementado en una tabla hash. Es la adecuada en la mayoría de los casos por ser la más rápida casi siempre.
- **TreeSet**. Conjunto implementado como un árbol. Mucho más lento, pero contiene los elementos ordenados por su valor.
- **LinkedHashSet**. Casi tan rápido como HashSet y tiene los elementos ordenados por orden de inserción.

### Algunos métodos (no todos) de la interfaz Set

- Si **s** es de tipo Set, **e** es 1 elemento.
- **s.add(e)** Añade el elemento **e** al conjunto.
- **s.isEmpty()** Dice si el conjunto está vacío.
- **s.size()** Obtiene el número de elementos del conjunto.
- **s.clear()** Vacía el conjunto.
- **s.contains(e)** Da **true** si el conjunto **s** contiene el elemento **e**.
- **s.remove(e)** Borra el elemento **e** del conjunto.

### Las correspondencias

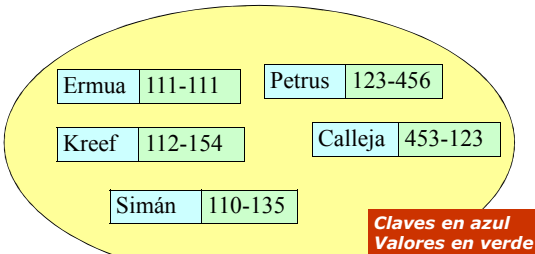
- Clases que implementan la interfaz **Map**. Son colecciones que hacen corresponder claves con valores. Para cada clave hay como máximo un valor. No tienen límite de elementos.





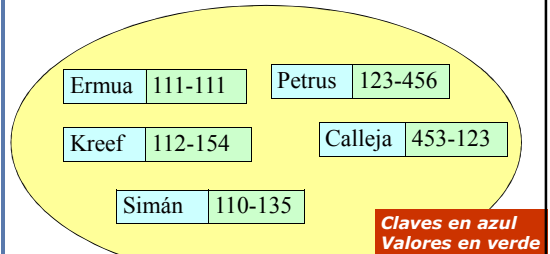
### Las correspondencias

- La idea es poder recuperar un valor a partir de una clave (al estilo de un listín telefónico).



### Las correspondencias

- Muchas veces tanto las claves como los valores son objetos y no tipos primitivos como aquí, pero lo hacemos así por una mejor gráfica.



### Las correspondencias

- Hay muchas clases que implementan la interfaz **Map**. Veamos las más usuales:
- HashMap**. Correspondencia implementada en una tabla hash. Es la adecuada en la mayoría de los casos por ser la más rápida casi siempre.
- TreeMap**. Correspondencia implementada como un árbol. Mucho más lenta, pero contiene los elementos ordenados por su valor.
- LinkedHashMap**. Casi tan rápido como HashMap y tiene los elementos ordenados por orden de inserción.

### Algunos métodos (no todos) de la interfaz Map

- Si **m** es de tipo **Map**, **c** es un objeto que sirve de clave y **v** es un objeto que sirve de posición.
- m.put(c,v)** Asocia la clave **c** con el valor **v** en la correspondencia **m** (Devuelve el anterior valor).
- m.isEmpty()** Dice si la correspondencia está vacía.
- (Clase)m.get(c)** Obtiene el valor que corresponde a la clave **c** y con la clase **Clase**
- m.size()** Obtiene el número de pares clave-valor de la correspondencia.
- m.clear()** Vacía la correspondencia.
- m.containsKey(c)** Da **true** si la correspondencia **m** contiene la clave **c**.
- m.containsValue(v)** Da **true** si la correspondencia **m** contiene el valor **v**.
- m.remove(c)** Borra la clave **c** y su valor asociado de la correspondencia.

### Nota

- Vemos que algunos de los métodos que aparecen son comunes a las interfaces **List**, **Set** y **Map** (y otros son específicos).
- Parecería buena idea tener una interfaz con esos métodos comunes.
- Esta interfaz ya existe.

### La interfaz Collection

- Todas las clases que son colecciones implementan la interfaz **Collection**.
- Todas las clases que hemos visto
- ArrayList**, **LinkedList**, **HashSet**, **TreeSet**, **LinkedHashSet**, **HashMap**, **TreeMap**, **LinkedHashMap**.
- Y muchas más implementan la interfaz **Collection**.
- (Recordemos que una misma clase puede implementar más de una interfaz).



### Algunos métodos (no todos) de la interfaz Collection

- Si **c** es de tipo **Collection**, **e** es 1 elemento.
- **c.add(e)** Añade el elemento **e** a la colección.
- **c.isEmpty()** Dice si la colección está vacía.
- **c.size()** Obtiene el número de elementos de la colección.
- **c.clear()** Vacía el conjunto.
- **c.contains(e)** Da **true** si la colección **c** contiene el elemento **e**.
- **c.remove(e)** Borra el elemento **e** de la colección.

### Nota: ¿Qué interfaz usamos?

- ¿La interfaz general **Collection** o las más específicos de **List**, **Set** y **Map**?
- Si queremos usar algunos métodos que sólo tengan las interfaces específicas, usaremos éstas.
- En todos los otros casos, usaremos **Collection**. Así, nuestro programa es más fácil de cambiar, pues si queremos cambiar la lista por un conjunto, nuestro programa no cambiará.

### Ejemplo:

Interfaz Collection	Interfaz específica	Clase concreta
Collection lista = new ArrayList(); ... out.print(list a.size());	List lista = new ArrayList(); ... out.print(lista .size()); out.print(lista .get(0));	ArrayList lista = new ArrayList(); ... out.print(lista. size()); out.print(lista. get(0)); lista.ensureCapa city(1000);
Se pueden hacer más cosas		Es más general

### Si queremos cambiar a otro tipo de lista

Interfaz Collection	Interfaz específica	Clase concreta
Collection lista = new ArrayList(); ... out.print(list a.size());	List lista = new ArrayList(); ... out.print(lista .size()); out.print(lista .get(0));	ArrayList lista = new ArrayList(); ... out.print(lista. size()); out.print(lista. get(0)); lista.ensureCapa city(1000);
		NO SE PUEDE

### Si queremos cambiar a un conjunto

Interfaz Collection	Interfaz específica	Clase concreta
Collection lista = new ArrayList(); ... out.print(list a.size());	List lista = new ArrayList(); ... out.print(lista .size()); out.print(lista .get(0));	ArrayList lista = new ArrayList(); ... out.print(lista. size()); out.print(lista. get(0)); lista.ensureCapa city(1000);
		NO SE PUEDE

### Ejemplo:

Interfaz Collection	Interfaz específica	Clase concreta
Collection lista = new ArrayList(); ... out.print(list a.size());	List lista = new ArrayList(); ... out.print(lista .size()); out.print(lista .get(0));	ArrayList lista = new ArrayList(); ... out.print(lista. size()); out.print(lista. get(0)); lista.ensureCapa city(1000);
Se pueden hacer más cosas		Es más flexible



### La idea es usar lo más general que funcione

- Miremos cuáles son las opciones que funcionan y elegimos la más general y flexible.
- Si sólo necesitamos una colección, sin características especiales, usamos la interfaz **Collection**.
- Si necesitamos características especiales, usamos las interfaces específicas (**Set**, **List**, **Map**). Así,
  - si necesitamos que los elementos de la colección estén ordenados usamos **List**.
  - si queremos que no haya repetidos **Set**.
  - si necesitamos acceso por una clave **Map**.
- Sólo usamos las clases concretas (**HashSet**, **ArrayList**, etc.) si tenemos necesidades muy específicas que no podemos satisfacer con interfaces.

### Recorriendo una colección

- Como ven, no todas las colecciones tienen el método **get** que nos permite acceder a los elementos.
- Queremos tener un método para recorrer los elementos de una colección (y aplicarle un determinado tratamiento).
- Queremos que el método se pueda aplicar a todas las colecciones, es decir, a todas las clases que implementen **Collection**.

### Recorriendo una colección

- Una forma es usando una variante del bucle **for**.

```
for (Object elemento: colección) {
}
```

Se lee “para cada objeto de la colección”.

- Dentro del código, para obtener el objeto de la colección se usa (**Clase**) **elemento**

### Suponemos que queremos imprimir el nombre de todos los empleados

- Tenemos una colección de empleados.

```
Empleado empleadoActual;
for (Object o: empleados) {
 empleadoActual = (Empleado)o;
 out.print(empleadoActual.getNombre());
}
```

O de forma más abreviada.

```
for (Object o: empleados) {
 out.print(((Empleado)o).getNombre());
}
```

### Mi gozo en un pozo

- Esta forma sólo está disponible en Java 1.5 y posteriores.
- La versión que usamos de Eclipse (la 3.0) no soporta las novedades de Java 1.5. Para ello, debemos esperar a la versión 3.1. que está en beta (y que es mucho más rápida).

### Iterator

- Una interfaz que permite recorrer los elementos de una colección.
- Cada colección tiene su **Iterator** que se obtiene con **colección.iterator()**
- Tiene dos métodos:
  - **hasNext()** devuelve un booleano que dice si hay un elemento siguiente.
  - (**Clase**) **next()** nos devuelve el elemento siguiente y avanza el recorrido.



### Suponemos que queremos imprimir el nombre de todos los empleados

- Tenemos una colección de empleados.

```
Empleado empleadoActual;
Iterator iterador = empleados.iterator();
while (iterador.hasNext()) {
 empleadoActual = (Empleado)next();
 out.print(empleadoActual.getNombre());
}
```

O con bucle **for**.

```
Empleado empleadoActual;
for (Iterator iterador=empleados.iterator();
 iterador.hasNext();){
 out.print(((Empleado)next()).getNombre());
}
```

### Iterator es más complicado que **for**

- Sin embargo, con **Iterator** podemos parar de recorrer la colección cuando queramos.
- Con el **for** siempre la recorremos toda.

## 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- **3.8. Composición con colecciones.**
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

## 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”

## 3.8. Composición con colecciones

- **3.8.1. Composición “uno a muchos”.**
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”

## Recordemos

- **Tipos de valor.** Sus miembros son valores. Los tipos primitivos, **String**, **BigDecimal**, **BigInteger** y muchos más.
- **Tipos de entidad.** Sus miembros son entidades. Todos los otros tipos de datos en Java.
- **Composición.** Es cuando hay atributos de una clase que no son de tipo de valor.



### Habíamos estudiado cierto tipos de composición

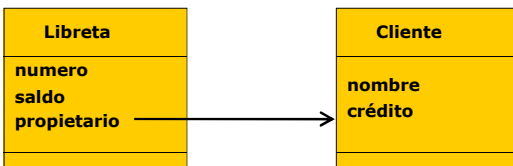
- Habíamos estudiado:
  - Uno a uno.
  - Muchos a uno.
- Ahora vamos a estudiar los dos tipos que aún no habíamos visto:
  - Uno a muchos.
  - Muchos a muchos.

### Vamos a ver estos casos

- Partiendo del mismo ejemplo que habíamos visto antes: el ejemplo de **Cliente** y **Libreta**.

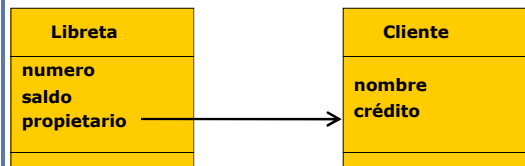
### Teníamos dos clases **Cliente** y **Libreta**

- La libreta tenía un atributo propietario que es de tipo Cliente.
- Esta es una relación “muchos a uno”: muchas libretas son propiedad de un solo cliente.



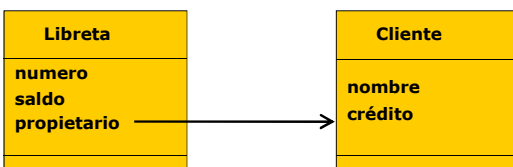
### A **Cliente** la llamábamos “Padre” y a **Libreta**, “Hijo”

- “Padre” era el extremo uno e “Hijo” el extremo “muchos”.
- Esta es una relación “muchos a uno”: muchas libretas son propiedad de un solo cliente.



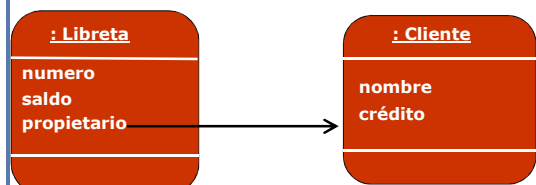
### **Nota:** Fíjense que en la programación O-O las relaciones son unidireccionales

- Van en una dirección y no en la contraria.
- Esto contrasta con la base de datos: las relaciones en el modelo relacional son bidireccionales.



### Podemos dibujarlo con objetos

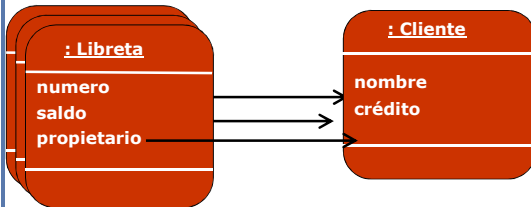
- Ya habíamos visto que la composición con objetos se ve de esta manera





### En realidad sería así

- Ya que hay varias libretas por cada cliente.



### En código, la clase Cliente era

```

package com.aurumsol.banco.dominio;
public class Cliente {
 private String nombre;
 private int credito;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 }
 public void setCredito(int credito){
 this.credito = credito;
 }
 public String getNombre(){
 return this.nombre;
 }
 public int getCredito(){
 return this.credito;
 }
}

```

### En código, la clase Libreta era (1)

```

package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 private Cliente propietario;
 public Libreta(int numero, Cliente propie){
 this.numero = numero;
 this.saldo = 0;
 this.propietario = propie;
 }
 public Cliente getPropietario(){
 return this.propietario;
 }
 public int getNumero(){
 return this.numero;
 }
}

```

### En código, la clase Libreta era (2)

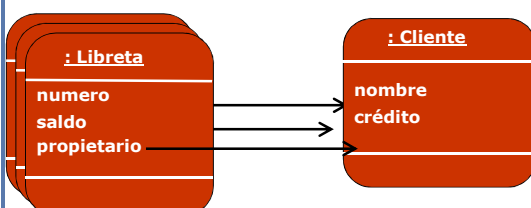
```

 public int getSaldo(){
 return this.saldo;
 }
 public void depositar(int monto){
 this.saldo += monto;
 }
 public boolean retirar(int monto){
 if (monto > this.saldo){
 return false;
 } else {
 this.saldo -= monto;
 return true;
 }
 }
}

```

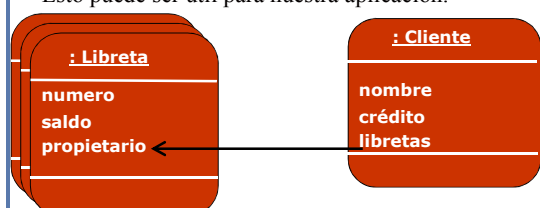
### Vemos que Cliente es accesible desde Libreta

- A partir de una libreta podemos saber a su propietario como **libreta.getPropietario()**
- Es decir, se sigue el sentido de la flecha.



### ¿Qué pasa si queremos seguir el sentido contrario?

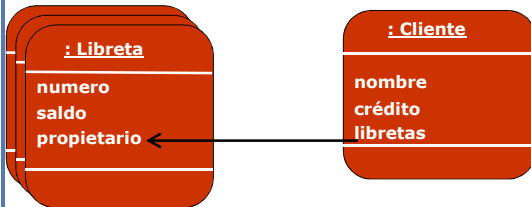
- Queremos, a partir de un cliente, acceder a las libretas de las que es propietario.
- Esto es una relación uno a muchos: un cliente tiene varias libretas.
- Esto puede ser útil para nuestra aplicación.





### Lo primero que vemos es que hay varias libretas por cada cliente

- Por lo tanto, debemos guardarlas como una colección o un arreglo.
- Veamos mejor el código.



### La clase Libreta ahora no tiene un atributo propietario

```

public class Libreta {
 private int numero;
 private int saldo;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public int getNumero(){
 return this.numero;
 }
 public int getSaldo(){
 return this.saldo;
 }
}

```

Es porque la composición va al revés. De Cliente se accede a las libretas y no al revés

### Por eso, la clase Libreta sólo tiene numero y saldo

```

public void depositar(int monto){
 this.saldo += monto;
}
public boolean retirar(int monto){
 if (monto > this.saldo){
 return false;
 } else {
 this.saldo -= monto;
 return true;
 }
}

```

### La clase Cliente (1)

```

public class Cliente {
 private int id;
 private String nombre;
 private int credito;
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new HashSet();
 }
 public void setCredito(int credito){
 this.credito = credito;
 }
}

```

Ahora tengo una colección de libretas cuyo propietario es el cliente.

### La clase Cliente (2)

```

public String getNombre(){
 return this.nombre;
}
public int getCredito(){
 return this.credito;
}
public void agregaLibreta(Libreta lib){
 this.libretas.add(lib);
}
public Collection getLibretas(){
 return this.libretas;
}

```

### Fijémonos en este trozo de código

```

private Collection libretas;
public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new HashSet();
}

```

- Utilizamos la interfaz **Collection** pues es la más general que funciona. Sólo queremos una colección, sin características.
- Para instanciarla, necesitamos una clase. Escogemos un **HashSet** pues no queremos repetidos.



### Fijémonos en este trozo de código

```
public Collection getLibretas(){
 return this.libretas;
}
```

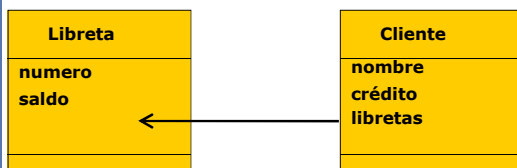
- Fijémonos que se devuelve una interfaz **Collection**.
- Así, el exterior de la clase **Cliente** no sabe cómo hemos implementado la colección. Si queremos cambiar la implementación en un futuro y pasar de un **HashSet** a un **ArrayList** sólo debemos cambiar la clase **Cliente** y el resto del programa seguirá igual.

### La clase **Cliente** resumida

```
public class Cliente {
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new HashSet();
 }
 public void agregaLibreta(Libreta lib){
 this.libretas.add(lib);
 }
 public Collection getLibretas(){
 return this.libretas;
 }
}
```

### Lo que tenemos es una asociación uno a muchos

- **libretas** es un conjunto de referencias a objetos de la clase **Libreta**



### Un uso posible de la clase

```
Libreta libretal=new Libreta(123);
Libreta libreta2=new Libreta(234);
Cliente cliente1=new Cliente("pepe");

cliente1.agregaLibreta(libretal);
cliente1.agregaLibreta(libreta2);
Collection libCliente1 =
 cliente1.getLibretas();

Libreta libActual;int dineroCliente1 =0;
for (iterador = libCliente1.iterator();
 iterador.hasNext();){
 libActual = (Libreta)iterador.next();
 dineroCliente1 += libActual.getSaldo();
}
```

### Esto está muy bien, pero ¿cómo guardamos esto en la base de datos?

- Tenemos cuatro clases de tipos relacionados con las colecciones:
  - Las listas, los conjuntos y los mapas, cada cual de ellas, tenía interfaces específicas **List**, **Set** y **Map**.
  - Los arreglos, que no eran colecciones, sino un caso parecido.
- Hibernate guarda de forma diferente los cuatro tipos de datos.

### Importante: cuando recuperemos colecciones de la base de datos

- Las podemos tratar como la interfaz **Collection** o como las interfaces **List**, **Set** y **Map**.
- No podemos tratarlas como las clases concretas **ArrayList**, **HashSet**, etc.
- Sin embargo, antes de guardarlas por primera vez si podemos usar las clases concretas.

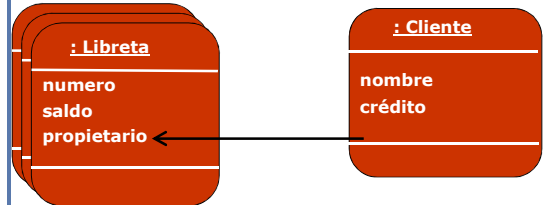


### En nuestro caso, guardaremos libretas como Set, pues es como lo implementamos

```
public class Cliente {
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new HashSet();
 }
 public void agregaLibreta(Libreta lib) {
 this.libretas.add(lib);
 }
 public Collection getLibretas() {
 return this.libretas;
 }
}
```

### Recordemos que la clase Cliente se llama padre y Libreta se llama hijo

- Padre es el extremo "uno" de una relación uno a muchos.
- Hijo es el extremo "muchos" de una relación uno a muchos.



### Sólo debemos añadir esto en el archivo de correspondencia de la clase Padre

```
<set name="atributoColeccion" lazy="false"
 access="field" cascade="all,delete-orphan">
 <key column="claveForáneaDeTablaHijo" />
 <one-to-many class ="paquete.ClaseHijo" />
</set>
```

- Esto va debajo de los **<property name** en el archivo de correspondencia.

### Para implementar el atributo libretas, usamos lo siguiente en Cliente.hbm.xml

```
<set name="libretas" lazy="false" access="field"
 cascade="all,delete-orphan">
 <key column="propietario" />
 <one-to-many class ="com.aurumsol.cursojava.
 banco.dominio.Libreta" />
</set>
```

### Cliente.hbm.xml

```
<hibernate-mapping>
<class name="com.aurumsol.cursojava.banco.
 dominio.Cliente" table="cliente">
 <id name="id" type="int" access="field">
 <column name="id" />
 <generator class="identity" />
 </id>
 <property name="nombre" access="field" />
 <property name="credito" access="field" />
 <set name="libretas" lazy="false"
 access="field" cascade="all,delete-orphan">
 <key column="propietario" />
 <one-to-many class ="com.aurumsol.cursojava.
 banco.dominio.Libreta" />
 </set>
</class>
</hibernate-mapping>
```

### Recordemos que la estructura de las tablas era así

libreta			
5	123	0	566

↑ Id    ↑ Número    ↑ Saldo    ↑ Propietario

cliente		
566	Juan	1000
	...	

↑ Id    ↑ Nombre    ↑ Crédito

- El campo propietario de la tabla libreta es la clave foránea que relaciona las dos tablas.



### Fijémonos que la estructura de las tablas es la misma que antes

libreta			
5	123	0	566

↑ Id    ↑ Número    ↑ Saldo    ↑ Propietario

cliente		
566	Juan	1000
	...	

↑ Id    ↑ Nombre    ↑ Crédito

- Cuando tenemos una relación muchos-a-uno (la que vimos antes) se tiene la misma BD que la relación uno-a-muchos (la que vemos ahora).

### Y ya está. Para implementar un conjunto.

- Crear una clave foránea de la tabla hijo a la tabla padre.
- Se añade el trozo de texto siguiente al archivo de correspondencia de la clase padre.

```
<set name="atributoColeccion" lazy="false"
access="field" cascade="all,delete-orphan">
<key column="claveForáneaDeTablaHijo" />
<one-to-many class ="paquete.ClaseHijo" />
</set>
```

### Ejercicio

- Se le distribuirá el **ejercicioComposicion1aN**.
- Deben crear el archivo de correspondencia **Cliente.hbm.xml**.
- Deben programar **obtenerLibretasClientes** de **NgcCliente**.
- Completar **obtienelibretascliente.jsp** con las partes que están comentadas.
- Desplegar y ejecutar.

### 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”

### Listas

- Una lista es parecida a un conjunto, pero los elementos de una lista tienen orden o posición (hay un primero, un segundo, etc.).
- Esta posición es un entero que comienza en cero.

### Podemos tener una clase Cliente así

```
public class Cliente {
 private List libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new ArrayList();
 }
 public void agregaLibreta(Libreta lib,
int posicion){
 this.libretas.add(posicion, lib);
 }
 public Libreta getLibreta(int posicion){
 return
 (Libreta) this.libretas.get (posicion);
 }
}
```



### ¿Cómo lo implementamos esto en la base de datos?

- Necesitamos un campo (columna) en la tabla hijo que contenga la posición del elemento en la lista.
- Este campo sólo debemos crearlo. Hibernate se encargará de actualizarlo.

### Tendremos un campo más en la tabla Hijo

libreta				
5	123	0	0	566
6				123
7	124	1	0	566

Id Número Posic. Saldo Propietario

cliente		
566	Juan	1000
	...	

Id Nombre Crédito

- Esta columna la tenemos que crear, pero no nos tenemos que ocupar de ella, ya que será actualizada automáticamente por Hibernate.

### Ahora debemos incluir en el archivo de correspondencia de la clase Padre

```
<list name="atributoColeccion" lazy="false"
access="field" cascade="all,delete-orphan">
 <key column="claveForáneaDeTablaHijo" />
 <list-index column="columnaPosicion" />
 <one-to-many class ="paquete.ClaseHijo" />
</list>
```

- Esto va debajo de los **<property name** en el archivo de correspondencia.
- Lo que es diferente del conjunto se pone en color rojo.

### Para implementar el atributo libretas, usamos lo siguiente en Cliente.hbm.xml

```
<list name="libretas" lazy="false" access="field"
cascade="all,delete-orphan">
 <key column="propietario" />
 <list-index column="posicion" />
 <one-to-many class ="com.aurumsol.cursojava.
banco.dominio.Libreta" />
</list>
```

### Y ya está. Para implementar una lista.

- Crear una clave foránea de la tabla hijo a la tabla padre (como en el conjunto).
- Crear una columna en la tabla hijo para que tenga la posición.
- Se añade el trozo de texto siguiente al archivo de correspondencia de la clase padre.

```
<list name="atributoColeccion" lazy="false"
access="field" cascade="all,delete-orphan">
 <key column="claveForáneaDeTablaHijo" />
 <list-index column="columnaPosicion" />
 <one-to-many class ="paquete.ClaseHijo" />
</list>
```

### ¿Conjuntos o listas?

- Las listas si necesitamos que haya un orden entre los elementos de la colección o si necesitamos que haya repetidos.
- Conjuntos en todos los otros casos, pues es más sencillo.



### 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”

### Arreglos

- Para guardar arreglos en la base de datos, se hace como si fueran listas.
- Para declararlo en el archivo de correspondencia, se usa como las listas, pero con la palabra array

```
<array name="atributoColeccion" lazy="false"
access="field" cascade="all,delete-orphan">
 <key column="claveForáneaDeTablaHijo" />
 <list-index column="columnaPosicion" />
 <one-to-many class ="paquete.ClaseHijo" />
</array>
```

### Podemos tener una clase Cliente así

```
public class Cliente {
 private static final MAX_LIBRETAS = 10000;
 private Libreta[] libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new Libreta[MAX_LIBRETAS]
 }
 public void agregaLibreta(Libreta lib, int
posicion){
 this.libretas[posicion]=lib;
 }
 public Libreta getLibreta(int posicion){
 return this.libretas[posicion];
 }
}
```

### ¿Arreglos o listas?

- Normalmente, listas pues tienen ventajas.
- Son extensibles (no tienen límite de elementos).
- Implementan la interfaz **Collection**.

### 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”

### Correspondencias

- Distinguimos entre correspondencias cuya clave es un tipo de valor o un tipo de entidad.
- Las correspondencias cuya clave es un tipo de valor son como las listas, pero en vez de tener un campo que es la posición, hay un campo que es la clave de la correspondencia.



### En una correspondencia en el que la clave es un tipo de valor

libreta				
5	123	"prin"	0	566
6				123
7	123	"sec"	0	566

Id Número Clave Saldo Propietario

- Ahora la clave se pone en una columna de la tabla hijo.

cliente		
566	Juan	1000
	...	

Id Nombre Crédito

### En el archivo de correspondencia de la clase hijo

- Se pone el siguiente texto.

```
<map name="atributoColeccion" lazy="false"
access="field" cascade="all,delete-orphan">
 <key column="claveForáneaDeTablaHijo" />
 <map-key column="colClave"
 type="tipoClave" />
 <one-to-many class ="paquete.ClaseHijo" />
</map>
```

### Para una correspondencia en el que la clave es un tipo de entidad

- La clave tendrá su propia clase en el programa.
- Esa clase tendrá una tabla propia en la base de datos.

### En el caso de correspondencia en el que la clave es una entidad

libreta				
5	123	2	0	566
6				123
7	123	1	0	566

Id Número Clave Saldo Propietario

tipolibreta	
1	LibretaPrincipal
2	LibretaAuxiliar
3	LibretaComplem.

Id Nombre

- La entidad de clave tiene su propia tabla.
- La clave del **Map** tiene una cl. foránea a esa tabla.

cliente		
566	Juan	1000
	...	

Id Nombre Crédito

### En el archivo de correspondencia de la clase hijo

- Se pone el siguiente texto.

```
<map name="atributoColeccion" lazy="false"
access="field" cascade="all,delete-orphan">
 <key column="claveForáneaDeTablaHijo" />
 <map-key-many-to-many column="colClave"
 class="claseClave" />
 <one-to-many class ="paquete.ClaseHijo" />
</map>
```

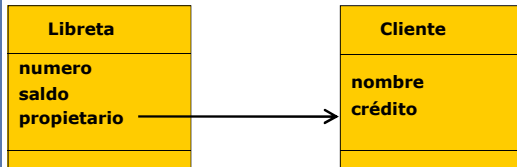
### 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”



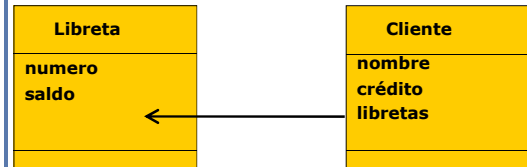
### Composiciones bidireccionales

- Recordemos cuando veíamos una composición muchos a uno.

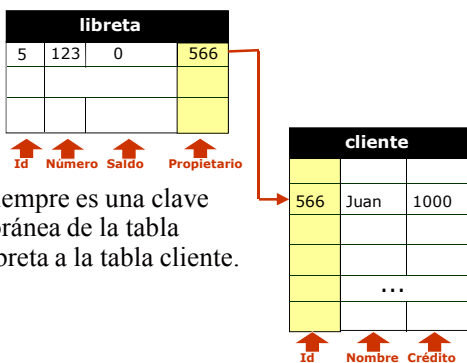


### Ahora tenemos la composición uno a muchos (la inversa)

- libretas** es un conjunto de referencias a objetos de la clase **Libreta**



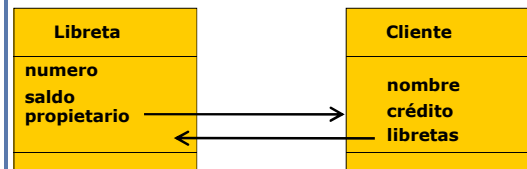
### Fijémonos que la estructura de las tablas es la misma en los 2 casos



- Siempre es una clave foránea de la tabla libreta a la tabla cliente.

### ¿Por qué no tenemos las dos?

- Así, a partir de una libreta podríamos encontrar su propietario o, a partir de una cliente, sus libretas.
- Esto se le llama composiciones bidireccionales.



### Veamos primero cómo implementar las composiciones bidireccionales en Java

- Las composiciones bidireccionales no son tan sencillas de implementar en Java.
- Primero debemos saber cómo se programan en Java antes de pensar como implementarlas con Hibernate.
- Parece obvio que debemos tener una referencia de **Libreta** a **Cliente** (un atributo propietario) y una referencia de **Cliente** a **Libreta** (un atributo libretas, que es una colección).

### Cliente debería tener una colección de libretas como atributo

```
public class Cliente {
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new HashSet();
 }
 public void agregaLibreta(Libreta lib){
 this.libretas.add(lib);
 }
 public Collection getLibretas(){
 return this.libretas;
 }
}
```



### Libreta debería tener un atributo propietario de tipo Cliente

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero;
 private int saldo;
 private Cliente propietario;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public Cliente getPropietario(){
 return this.propietario;
 }
 public void setPropietario(Cliente propie){
 this.propietario = propie;
 } //Otros métodos
}
```

### Esto está muy bien, siempre que tengamos presente que

- Para incluir un objeto una relación bidireccional, hay que fijar las dos referencias desde las dos clases.

```
cliente1.agregaLibreta(libreta1);
libreta1.setPropietario(cliente1);
```

- Se pueden programar las clases para que esto lo haga automáticamente, aunque el código no sale tan sencillo.

### Este es el código para que la doble referencia la haga automáticamente

```
package com.aurumsol.banco.dominio;
public class Libreta {
 private int numero, saldo;
 private Cliente propietario;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public Cliente getPropietario(){
 return this.propietario;
 }
 public void setPropietario(Cliente propie){
 this.propietario = propie;
 if (!propie.poseeLibreta(this)){
 propie.agregaLibreta(this);
 }
 } //Otros métodos
}
```

### Este es el código para que la doble referencia la haga automáticamente

```
public class Cliente {
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre; this.credito = 0;
 this.libretas = new HashSet();
 }
 public void agregaLibreta(Libreta lib){
 this.libretas.add(lib);
 if (lib.getPropietario() != this){
 lib.setPropietario(this);
 }
 }
 public boolean poseeLibreta(Libreta lib){
 return this.libretas.contains(lib);
 }
 public Collection getLibretas(){
 return this.libretas;
 }
}
```

### Esto en cuanto el código. ¿Cómo persistimos estas asociaciones?

- En la base de datos, es sencillo: sólo tenemos que crear la misma estructura de tablas que hemos visto.
- En Hibernate, también es fácil.
  - Se usa el mismo texto que en la asociación muchos a uno y uno a muchos **COMBINÁNDOLOS**.
  - La parte uno a muchos se marca con la expresión **inverse="true"**

### La estructura de las tablas será la misma que siempre

libreta			
5	123	0	566

↑ Id    ↑ Número    ↑ Saldo    ↑ Propietario

cliente		
566	Juan	1000
	...	

↑ Id    ↑ Nombre    ↑ Crédito

- Siempre es una clave foránea de la tabla libreta a la tabla cliente.



### Libreta.hbm.xml como en la composición muchos a unos

```
<hibernate-mapping>
<class name="com.aurumsol.cursojava.banco.dominio.Libreta" table="libreta">
 <id name="id" type="int" access="field">
 <column name="id"/>
 <generator class="identity"/>
 </id>
 <property name="numero" access="field"/>
 <property name="saldo" access="field"/>
 <many-to-one name="propietario"
 class="com.aurumsol.cursojava.banco.dominio.Cliente" column="propietario" access="field"
 lazy="false"/>
</class>
</hibernate-mapping>
```

### Cliente.hbm.xml como en composición uno a muchos con inverse="true"

```
<hibernate-mapping>
<class name="com.aurumsol.cursojava.banco.dominio.Cliente" table="cliente">
 <id name="id" type="int" access="field">
 <column name="id"/>
 <generator class="identity"/>
 </id>
 <property name="nombre" access="field"/>
 <property name="credito" access="field"/>
 <set name="libretas" inverse="true" lazy="false"
 access="field" cascade="all,delete-orphan">
 <key column="propietario"/>
 <one-to-many class="com.aurumsol.cursojava.banco.dominio.Libreta"/>
 </set>
</class>
</hibernate-mapping>
```

### Composición bidireccional

- Con todo esto ya tenemos una composición bidireccional.
- Podemos acceder, a partir de una libreta, a su propietario (que es un cliente).
- Podemos acceder, a partir de un cliente, a sus libretas.

### ¿Necesitamos todo esto?

- Como hemos visto, una relación bidireccional no es tan sencilla de programar.
- Además, crea una dependencia entre las dos clases bastante poco elegante.
- Por eso, se recomienda usarla sólo en casos específicos, que son menos de los que parece a simple vista.

### En principio, parecería que todo deberían ser relaciones bidireccionales

- Así, entre Libreta y Cliente, para que podamos:
  - A partir de una libreta, saber cuál es su propietario.
  - A partir de un cliente, qué libretas posee.
- Pero esto se puede hacer con una relación unidireccional. Por ejemplo, una muchos a unos de Libreta a Cliente.
- Para saber el propietario de una libreta, seguimos esta relación.
- Para saber qué libretas tiene un cliente no usamos la relación sino que simplemente buscamos en la base de datos (con HQL, por ejemplo).

### 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”



### Asociaciones “muchos a muchos”

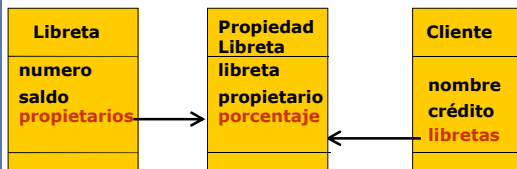
- Hasta ahora, hemos visto:
  - Asociaciones uno a uno.
  - Asociaciones uno a muchos.
  - Asociaciones muchos a uno.
- Nos falta por ver la asociación muchos a muchos.

### Ejemplo: que una libreta pueda tener más de un propietario

- Relación muchos a muchos: un cliente puede tener muchas libretas, una libreta puede ser de muchos clientes.

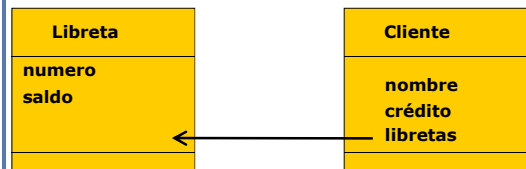
### Quizás la mejor opción sea crear una nueva clase

- De esta manera, descomponemos la relación muchos a muchos en dos relaciones uno a muchos.
- Esto permite añadir atributos a la relación (como porcentaje, fecha, derecho, etc).



### Veamos como hacer 1 relación de muchos a muchos

- Primero veremos una relación unidireccional.
- El cliente quiere acceder a las libretas que posee, pero no necesitamos el acceso a los propietarios desde una libreta.



### La clase Libreta no tiene referencia a la clase Cliente

```

public class Libreta {
 private int numero;
 private int saldo;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 //Otros métodos
}

```

### La clase Cliente tendrá una colección de libretas

```

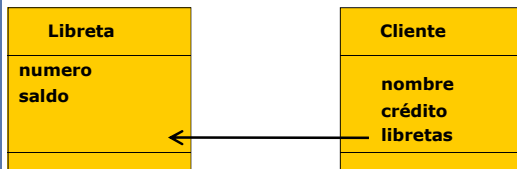
public class Cliente {
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new HashSet();
 }
 public void agregaLibreta(Libreta lib){
 this.libretas.add(lib);
 }
 public Collection getLibretas(){
 return this.libretas;
 }
}

```



### Un poco de terminología para aclararnos

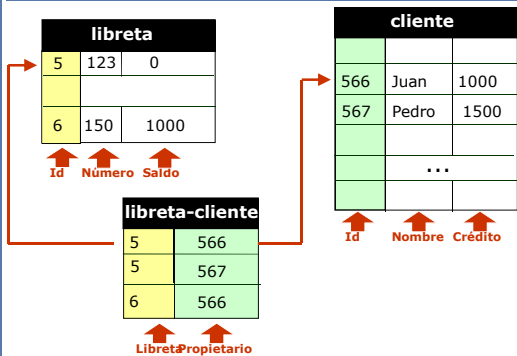
- La clase que contiene la colección a la otra clase la llamaremos “Clase que Referencia” y a la otra la llamaremos “Clase Referenciada”.
- En nuestro caso, son respectivamente Cliente y Libreta.



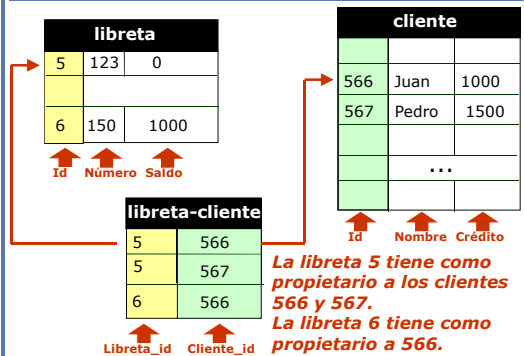
### En la base de datos

- Para una relación “muchos a muchos” se necesitará una “tabla de enlace”.
- La tabla de enlace es una tabla que contiene claves foráneas a las tablas que intervienen en la relación muchos a muchos.

### En nuestro caso, la tabla de enlace tiene claves foráneas a libreta y cliente



### La tabla de enlace dice qué propietarios tiene cada libreta



### En Hibernate, pondríamos esto en la clase que referencia

```

<set name="atributoColeccion" table="tablaEnlace"
lazy="false" access="field"
cascade="save-update">
 <key column="claveForáneaClaseQueReferencia"/>
 <many-to-many class="ClaseReferenciada"
column="claveForáneaClaseReferenciada"/>
</set>

```

- El cascade “save-update” nos dice que cuando se grabe o actualice la colección también se grabarán o actualizarán los elementos de la colección.
- También se pueden usar listas, arreglos o mapas.

### En nuestro caso, pondríamos en Cliente.hbm.xml

```

<set name="libretas" table="libreta-cliente"
lazy="false" access="field"
cascade="save-update">
 <key column="cliente_id"/>
 <many-to-many class="com.aurumsol.cursojava.
banco.dominio.Libreta" column="libreta_id"/>
</set>

```

- Esto definiría el atributo de colección.

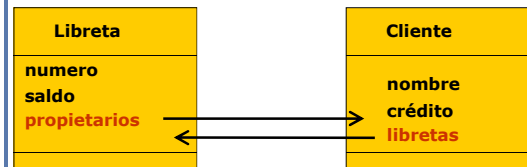


### 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”

### Ahora veremos una relación bidireccional muchos a muchos

- Ahora hay una colección en cada una de las clases que tiene colecciones con la otra.



### La clase Libreta ahora tiene una colección de propietarios

```
public class Libreta {
 private int numero;
 private int saldo;
 private Collection propietarios;
 public Libreta(int numero){
 this.numero = numero;
 this.saldo = 0;
 }
 public void agregaPropietario (Cliente propie){
 this.propietarios.add(propie);
 }
 public Collection getPropietarios(){
 return this.propietarios;
 }
 //Otros métodos
}
```

### La clase Cliente tendrá una colección de libretas

```
public class Cliente {
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre;
 this.credito = 0;
 this.libretas = new HashSet();
 }
 public void agregaLibreta(Libreta lib){
 this.libretas.add(lib);
 }
 public Collection getLibretas(){
 return this.libretas;
 }
}
```

### De todas maneras

- Recordemos que en toda relación bidireccional, cada vez que se agrega un objeto hay que fijar las dos referencias:

```
libreta1.agregaPropietario(cliente1);
cliente1.agregaLibreta(libreta1);
```

- Sabemos que podemos evitar esto programando las clases con cuidado, pero no es tan obvio.

### Código de Libreta para que no debamos acordarnos de fijar las dos referencias

```
public class Libreta {
 private int numero, saldo;
 private Collection propietarios;
 public Libreta(int numero){
 this.numero = numero; this.saldo = 0;
 }
 public void agregaPropietario (Cliente propie){
 this.propietarios.add(propie);
 if (!propie.poseeLibreta(this)){
 propie.agregaLibreta(this);
 }
 }
 public Collection getPropietarios(){
 return this.propietarios;
 }
 public boolean esDePropietario(Cliente propie){
 return this.propietarios.contains(propie);
 }
 //Otros métodos
}
```



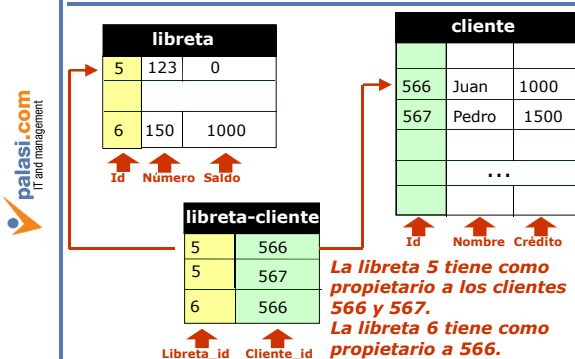
### Código de Cliente para que no debamos acordarnos de fijar las dos referencias

```
public class Cliente {
 private Collection libretas;
 public Cliente (String nombre){
 this.nombre = nombre; this.credito = 0;
 this.libretas = new HashSet();
 }
 public void agregaLibreta(Libreta lib){
 this.libretas.add(lib);
 if (!lib.esDePropietario(this)){
 lib.agregaPropietario(this);
 }
 }
 public Collection getLibretas(){
 return this.libretas;
 }
 public boolean poseeLibreta(Libreta lib){
 return this.libretas.contains(lib);
 }
}
```

### Esto en cuanto al código

- Nos falta la base de datos y los archivos de correspondencia de Hibernate.

### La base de datos igual que en relación muchos a muchos unidireccional



### Ahora pondríamos este trozo de código en los archivos de corr., en vez de sólo 1

```
<set name="atributoColeccion" table="tablaEnlace"
lazy="false" access="field"
cascade="save-update">
 <key column="claveForáneaClaseQueReferencia" />
 <many-to-many class ="ClaseReferenciada"
 column ="claveForáneaClaseReferenciada" />
</set>
```

- Ahora bien, una de las dos colecciones estará marcada con **inverse = "true"**, para indicar que es la relación inversa a la otra (puede ser cualquiera de las dos).

### En nuestro caso, pondríamos en Cliente.hbm.xml

```
<set name="libretas" table="libreta-cliente"
lazy="false" access="field"
cascade="save-update">
 <key column="cliente_id"/>
 <many-to-many class ="com.aurumsol.cursojava.
 banco.dominio.Libreta" column ="libreta_id"/>
</set>
```

En Libreta.hbm.xml.

```
<set name="propietarios" table="libreta-cliente"
lazy="false" access="field" inverse="true">
 <key column="libreta_id"/>
 <many-to-many class ="com.aurumsol.cursojava.
 banco.dominio.Cliente" column ="cliente_id"/>
</set>
```

### Hay algunas complicaciones

- En el extremo “inverso” de la relación se puede implementar con **Set**.
- El extremo “no inverso” de la relación se puede usar **Set**, **List**, arreglos y **Map**.



### ¿Necesitamos todo esto?

- Como hemos visto, una relación bidireccional no es tan sencilla de programar.
- Además, crea una dependencia entre las dos clases bastante poco elegante.
- Por eso, se recomienda usarla sólo en casos específicos, que son menos de los que parece a simple vista.

### En principio, parecería que todo deberían ser relaciones bidireccionales

- Así, entre Libreta y Cliente, para que podamos:
  - A partir de una libreta, saber cuáles son sus propietarios.
  - A partir de un cliente, qué libretas posee.
- Pero esto se puede hacer con una relación unidireccional. Por ejemplo, una muchos a muchos de Libreta a Cliente.
- Para saber los propietarios de una libreta, seguimos esta relación.
- Para saber qué libretas tiene un cliente no usamos la relación sino que simplemente buscamos en la base de datos (con HQL, por ejemplo).

## 3.8. Composición con colecciones

- 3.8.1. Composición “uno a muchos”.
  - Conjuntos.
  - Listas.
  - Arreglos.
  - Correspondencias.
- 3.8.2. Composiciones bidireccionales “1 a muchos”
- 3.8.3. Composiciones “muchos a muchos”.
  - Unidireccionales.
  - Bidireccionales.
- 3.8.4. La expresión “cascade”

### La expresión cascade

```
<set name="libretas" lazy="false"
 access="field" cascade="all,delete-orphan">
 <key column="propietario"/>
 <one-to-many class="com.aurumsol.
 cursojava.banco.dominio.Libreta"/>
</set>
```

- Hasta ahora hemos puesto **cascade="all, delete-orphan"** o **cascade="save-update"**. ¿Qué es esto?

### Esta expresión nos indica que, en una relación padre a hijo

```
<set name="libretas" lazy="false"
 access="field" cascade="all,delete-orphan">
 <key column="propietario"/>
 <one-to-many class="com.aurumsol.
 cursojava.banco.dominio.Libreta"/>
</set>
```

- Si se guarda o actualiza un objeto padre (cliente), se guardan y actualizan todos sus hijos (libretas).
- Si se borra un padre (cliente), se borran todos sus hijos (libretas).
- Si un hijo (libreta) se queda sin padre (cliente), se borra el hijo de la base de datos.

### Útil en relaciones en la que la duración del hijo está comprendida en la del padre

- Así, la duración de las libretas está comprendida en la duración del cliente.
- Sin embargo, en otro tipo de relaciones, puede ser contraproducente.
- Por ejemplo, si tenemos una relación entre Empresa y Persona, si se borra una empresa, puede ser que no nos interese borrar una persona.
- Para estos casos, es mejor
  - Poner **cascade="save-update"**.
  - No poner cascade.



### Una guía para las relaciones uno a uno, uno a muchos o muchos a uno

- Si tenemos una relación unidireccional entre A y B, con la dirección A -> B (las relaciones bidireccionales se tratan como dos relaciones unidireccionales diferentes).
- Si la duración de B está incluida en la duración de A, **cascade="all, delete-orphan"**
- Si no,
  - Si los objetos A se guardan antes que los B, **cascade ="none"**.
  - Si los objetos A se guardan después que los B, **cascade="save-update"**.

### Una guía para las relaciones muchos a muchos

- Si tenemos una relación unidireccional entre A y B, con la dirección A -> B (las relaciones bidireccionales se tratan como dos relaciones unidireccionales diferentes).
  - Si los objetos A se guardan antes que los B, **cascade ="none"**.
  - Si los objetos A se guardan después que los B, **cascade="save-update"**.
- Nota: No se puede poner cascade="all" pues cada hijo tiene más de un padre y un "hijo" sería borrado cuando se borra sólo uno de sus padres.

## 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- **3.9. Herencia.**
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

## 3.9. Herencia

- 3.9.1. Introducción a herencia.
- 3.9.2. Jerarquías de herencia.
- 3.9.3. Métodos abstractos y sobreescritura.
- 3.9.4. Conversión de tipos y herencia.
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.

## 3.9. Herencia

- **3.9.1. Introducción a herencia.**
- 3.9.2. Jerarquías de herencia.
- 3.9.3. Métodos abstractos y sobreescritura.
- 3.9.4. Conversión de tipos y herencia.
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.

## Un ejemplo

- Una empresa se dedica a la compra/venta de autos. Trata con dos tipos de auto:
- **Autos importados.** Interesa el nombre, el valor (en dólares), el país de importación y el impuesto sobre autos.
- **Autos nacionales.** Interesa el nombre, el valor, la empresa que lo fabricó y el impuesto sobre autos.
- El impuesto de importación es el mismo para autos importados y nacionales (el 10% de su valor a partir de los 1000 dólares). Además nos interesa tener un registro de todos los autos.



### Una primera aproximación

- Como necesitamos un registro de todos los autos eso quiere decir que en este registro trataremos tanto los importados como los nacionales de la misma forma.
- En otras ocasiones, los tratamos de forma diferente (por ejemplo, del auto nacional necesitamos su fabricante y del auto importado su año de importación).
- Es decir, necesitamos polimorfismo.

### Necesitamos polimorfismo

- Necesitamos tratar a veces un auto importado como auto importado y a veces como auto.
- Necesitamos tratar a veces un auto nacional como auto nacional y a veces como auto.
- Polimorfismo es tratar un mismo objeto de formas diferentes según nos interesa.
- **Como necesitamos polimorfismo podemos usar interfaces.**

### Tendríamos las siguientes clases

- Veamos que tienen métodos comunes (en rojo) que serán los que usaremos en la interfaz.

**AutoImportado**

```

getValor
setValor
getPais
getImpuesto

```

**AutoNacional**

```

getValor
setValor
getEmpresa
getImpuesto

```

### La interfaz Auto

```

public interface Auto {
 public double getValor();
 public void setValor (double valor);
 public double getImpuesto();
}

```

### La clase AutoImportado

```

public class AutoImportado implements Auto{
 protected double valor; protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
 public Pais getPais(){
 return this.pais;
 }
}

```

### Nota importante

- Ven que para los atributos hemos usado **protected** y no **private**.
- Más adelante, veremos porque es esto.
- Por ahora, traten a **protected** como si fuera **private**.



### La clase AutoNacional

```
public class AutoNacional implements Auto{
 protected double valor; protected Empresa empresa;
 public AutoNacional(Empresa empresa){
 this.empresa = empresa;
 }
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
 public Empresa getEmpresa(){
 return this.empresa;
 }
}
```

### Ahora bien

- Vemos que hay mucho código que se repite en las dos clases.
- Excepto los constructores y lo referente a la empresa y al país, todo lo otro es repetido.

### El código en rojo está repetido en las dos clases

```
public class AutoImportado implements Auto{
 protected double valor; protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
 public Pais getPais(){
 return this.pais;
 }
}
```

### El código en rojo está repetido en las dos clases

```
public class AutoNacional implements Auto{
 protected double valor; protected Empresa empresa;
 public AutoNacional(Empresa empresa){
 this.empresa = empresa;
 }
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
 public Empresa getEmpresa(){
 return this.empresa;
 }
}
```

### Mandamiento del mantenimiento

#### *No repetirás código*

- La repetición de código hace el programa difícil de mantener y propenso a errores.
- Cuando cambiamos una parte del código, debemos cambiar la otra parte que es igual. Cuando nos olvidamos ocasionamos inconsistencias.



### Mandamiento del mantenimiento

#### *No repetirás código*

- Esto es bien conocido desde el principio de la programación.
- Para esto se inventaron los procedimientos y funciones en el pasado remoto.





### ¿Qué hacemos?

- Necesitamos:
  - Polimorfismo.
  - Reutilizar código (tener un solo código y reusarlo en varias partes, en vez de repetirlo).
- Para el polimorfismo ya nos iban bien las interfaces.
- Necesitamos algo que no sólo nos de polimorfismo sino también reutilización de código.

### Este mecanismo existe: se llama herencia

- Es un mecanismo que nos permite implementar las dos cosas a la vez:
  - Polimorfismo.
  - Reutilización de código.

### Definimos una clase llamada **Auto** que contiene todo el código común

```
public abstract class Auto{
 protected double valor;
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
}
```

- Fíjense que esto es una clase, no una interfaz como antes.

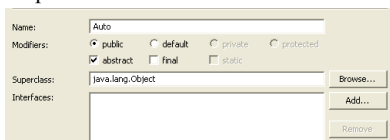
### Fíjense que se le pone la palabra **abstract**

```
public abstract class Auto{
 protected double valor;
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
}
```

- Se le llama clase abstracta. Esto no es absolutamente necesario pero sí muy conveniente

### Para crear una clase abstracta

- Se puede seleccionar la casilla de verificación **abstract** en Eclipse y generará el código correspondiente.



- Simplemente, poner la palabra **abstract** en el código fuente sin necesidad de seleccionar la casilla de verificación anteriormente.

### Herencia

- Con la ayuda de la palabra reservada **extends** podemos definir **AutoImportado** como una clase específica de **Auto**.
- Comparte todos los miembros de **Auto** pero, además tiene miembros específicos: como los relacionados con el país de importación.
- Se dice que **AutoImportado** extiende o hereda de **Auto**.

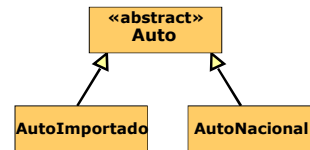


### Ahora las clases AutoImportado y AutoNacional quedan así

```
public class AutoImportado extends Auto{
 protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public Pais getPais(){
 return this.pais;
 }
}

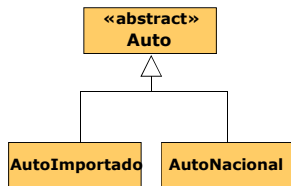
public class AutoNacional extends Auto{
 protected Empresa empresa;
 public AutoNacional(Empresa empresa){
 this.empresa = empresa;
 }
 public Empresa getEmpresa(){
 return this.empresa;
 }
}
```

### En UML que una clase extienda otra se representa con una flecha



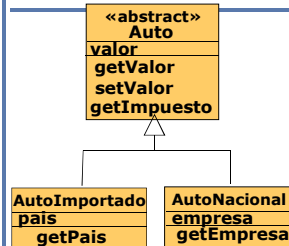
- Es una flecha continua con punta triangular.
- La flecha es la forma de UML de decir "extends".

### También se puede representar así, por comodidad



- Son dos flechas, no sólo una flecha.

### También se pueden detallar los miembros de cada clase



- Recordemos que en UML las tres divisiones dan:
  - Nombre de la clase.
  - Atributos.
  - Métodos.

### Todo el código común se pone en la clase abstracta

```
public abstract class Auto{
 protected double valor;
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
}
```

### Todo el código específico se pone en cada una de estas clases

```
public class AutoImportado extends Auto{
 protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public Pais getPais(){
 return this.pais;
 }
}

public class AutoNacional extends Auto{
 protected Empresa empresa;
 public AutoNacional(Empresa empresa){
 this.empresa = empresa;
 }
 public Empresa getEmpresa(){
 return this.empresa;
 }
}
```

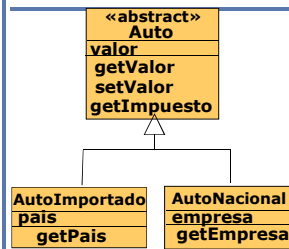


### Concentrémonos en la siguiente clase

```
public class AutoImportado extends Auto{
 protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public Pais getPais(){
 return this.pais;
 }
}
```

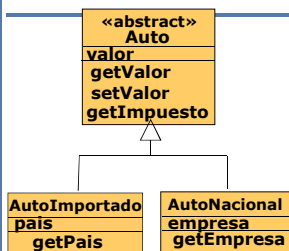
- Mucho más corta que la anterior. Sólo incluimos los miembros específicos de los autos importados.
- Los otros miembros están disponibles pues decimos que **heredamos (extends)** la clase **Auto**. Por lo tanto, todo lo de **Auto** está disponible aquí

### Todo lo de Auto está en AutoImportado



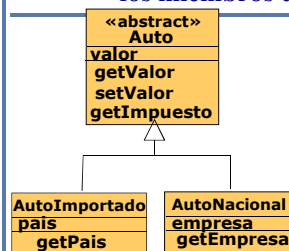
- Cualquier objeto de tipo **AutoImportado** tiene los atributos: **pais** (definido en **AutoImportado**) y **valor** (definido en **Auto** y heredado).

### Todo lo de Auto está en AutoImportado



- Cualquier objeto de tipo **AutoImportado** tiene los métodos: **getImpuesto** (definido en **Auto**) y **getValor**, **getAutoImportado** y **getAutoNacional** (definidos en **Auto**).

### Se dice que AutoImportado hereda todos los miembros de Auto



- Es decir, **AutoImportado** tiene todos los miembros de **Auto** y además unos específicos de él.

### Con la herencia se obtiene la siguiente clase

```
public class AutoImportado extends Auto{
 protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public Pais getPais(){
 return this.pais;
 }
}
```

- Es como si hubiéramos copiado los atributos y métodos de la clase **Auto** dentro de la clase **AutoImportado** (y de la clase **AutoNacional**).
- Pero sin los problemas de repetir código.

### Ahora cambios en la implementación son fáciles

```
public abstract class Auto{
 protected double valor;
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public double getImpuesto(){
 return 0.30*(this.valor-1000);
 }
}
```

Si quiero cambiar el cálculo del impuesto, lo hago en la clase **Auto**, ya que **no hay repetición de código**.



### Ahora cambios en la implementación son fáciles

```
public class AutoCentroamericano extends Auto{
 protected Sucursal sucursal;
 public AutoCentroamericano (Sucursal sucursal){
 this.sucursal = sucursal;
 }
 public Sucursal getSucursal(){
 return this.sucursal;
 }
}
```

Si quisiéramos añadir un nuevo tipo de auto, sólo debo extender la clase abstracta **Auto**.

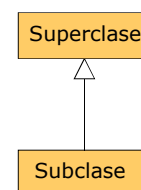
### Ventajas de la herencia

- Código más legible y claro.
- Código más fácil de mantener.
- Código más flexible.

### 3.9. Herencia

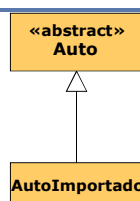
- 3.9.1. Introducción a herencia.
- 3.9.2. Jerarquías de herencia.
- 3.9.3. Métodos abstractos y sobrescritura.
- 3.9.4. Conversión de tipos y herencia.
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.

### Un poco de notación



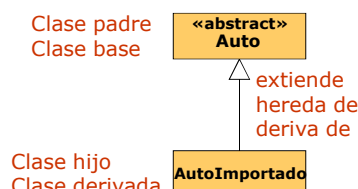
- La clase que tiene la palabra **extends** se le llama subclase.
- La clase a la cual hace referencia con **extends** se le llama superclase.

### En nuestro caso



- **Auto** es la superclase y **AutoImportado** es la subclase.

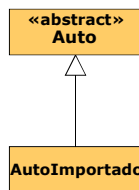
### Más terminología



- Se dice que **AutoImportado** hereda de **Auto** o bien que **AutoImportado** extiende **Auto** o bien que **AutoImportado** deriva de **Auto**.

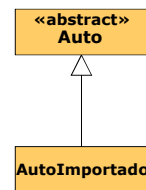


### Es fácil ver que AutoImportado es un tipo especial de Auto



- Esto es general: siempre la subclase debe ser un tipo especial de la superclase.
- La superclase es más general y la subclase es más específica.

### Dicho de otra manera: AutoImportado es un Auto



- Por ello, a la relación de herencia se le llama a veces relación “ES UN” (“is a”).

### Imaginemos que ahora queremos modelar una bicicleta

- Las bicicletas no tienen impuesto, pero sí valor, además de un diferente número de marchas.

```

public class Bicicleta {
 protected double valor; protected int
 marchas;
 public Bicicleta (int marchas){
 this.marchas = marchas;
 }
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public int getMarchas(){
 return this.marchas
 }
}

```

### Esta clase tiene mucho código en común con Auto

- Todo lo que está en rojo.

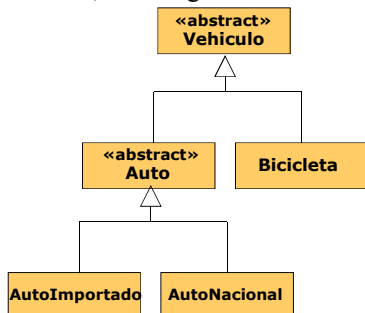
```

public class Bicicleta {
 protected double valor; protected int
 marchas;
 public Bicicleta (int marchas){
 this.marchas = marchas;
 }
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
 public int getMarchas(){
 return this.marchas
 }
}

```

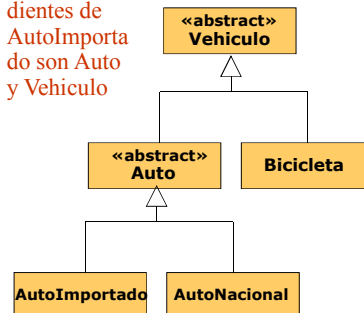
### No sería mala idea crear una superclase para Bicicleta y Auto

- Es decir, tener algo así.



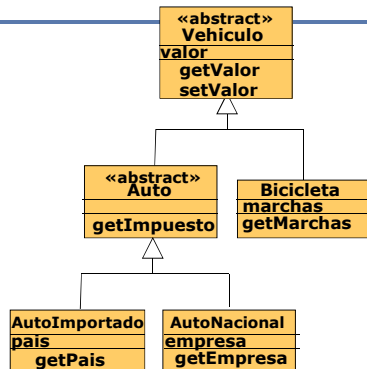
### Se usan los términos ascendientes y descendientes

- Los ascendientes de AutoImportado son Auto y Vehiculo
- Así, los descendientes de vehiculo son todas las otras clases.





### Detallando los miembros



### Veamos el código de la clase Vehiculo

- Todo lo que está en rojo.

```

public abstract class Vehiculo {
 protected double valor;
 public double getValor() {
 return this.valor;
 }
 public void setValor (double valor) {
 this.valor = valor;
 }
}

```

### Bicicleta queda así

- Ahora **Bicicleta** sólo tiene lo específico de la bicicleta: el número de marchas. Todo lo del valor lo hereda de **Vehiculo**.

```

public class Bicicleta extends Vehiculo {
 protected int marchas;
 public Bicicleta (int marchas) {
 this.marchas = marchas;
 }
 public int getMarchas() {
 return this.marchas
 }
}

```

### Ahora la clase Auto queda así

```

public abstract class Auto extends Vehiculo{
 public double getImpuesto() {
 return 0.30*(this.valor-1000);
 }
}

```

Es una clase abstracta que hereda de otra clase abstracta.

### Las clases AutoImportado y AutoNacional siguen como antes

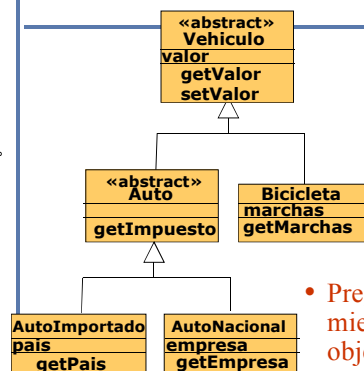
```

public class AutoImportado extends Auto{
 protected Pais pais;
 public AutoImportado(Pais pais) {
 this.pais = pais;
 }
 public Pais getPais() {
 return this.pais;
 }
}

public class AutoNacional extends Auto{
 protected Empresa empresa;
 public AutoNacional(Empresa empresa) {
 this.empresa = empresa;
 }
 public Empresa getEmpresa() {
 return this.empresa;
 }
}

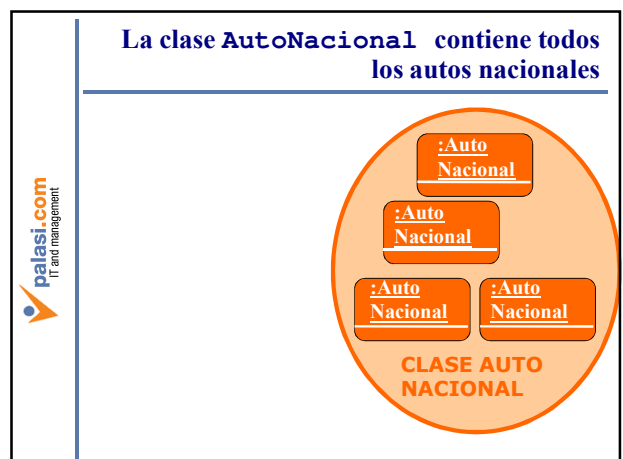
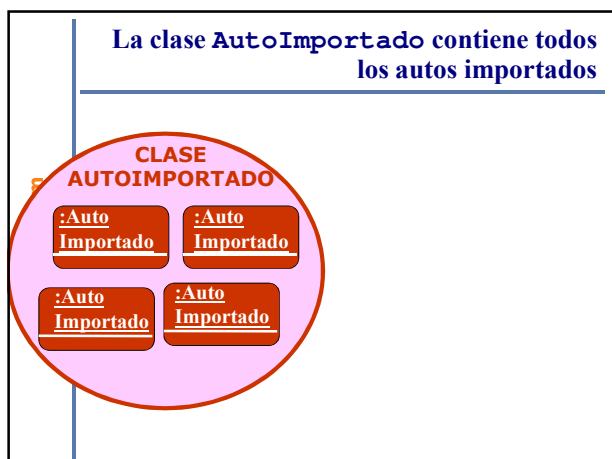
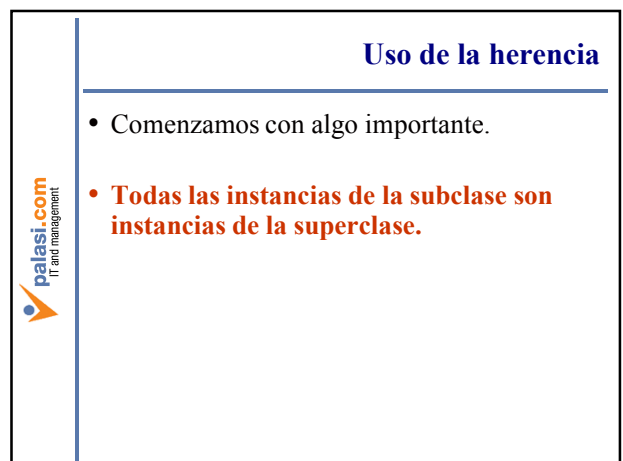
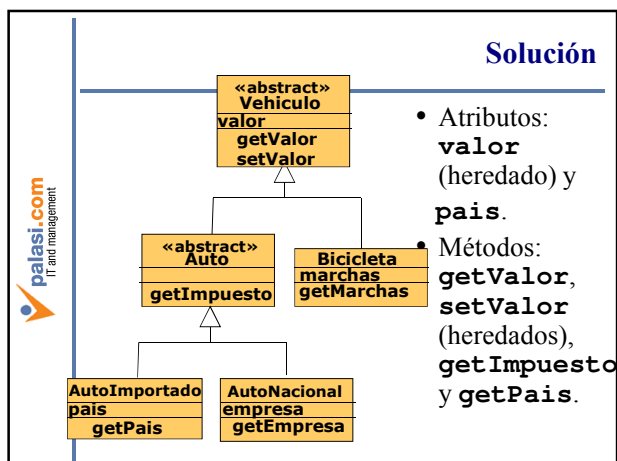
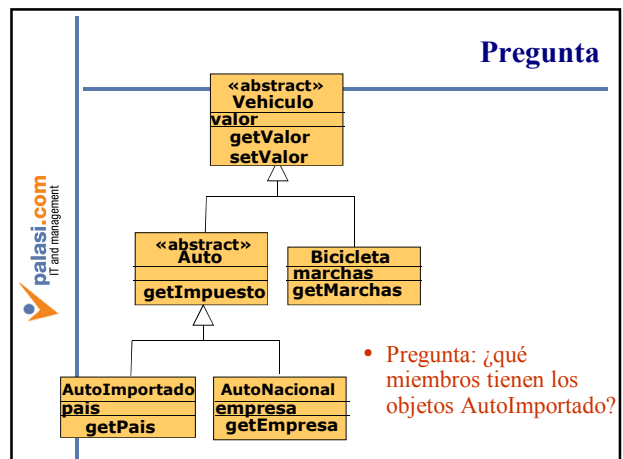
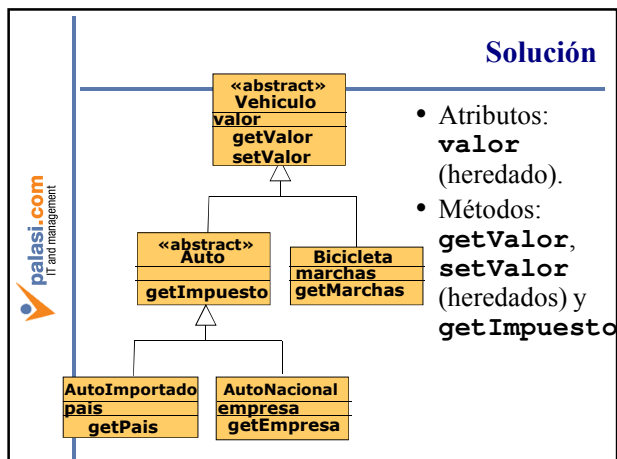
```

### Pregunta



- Pregunta: ¿qué miembros tienen los objetos Auto?



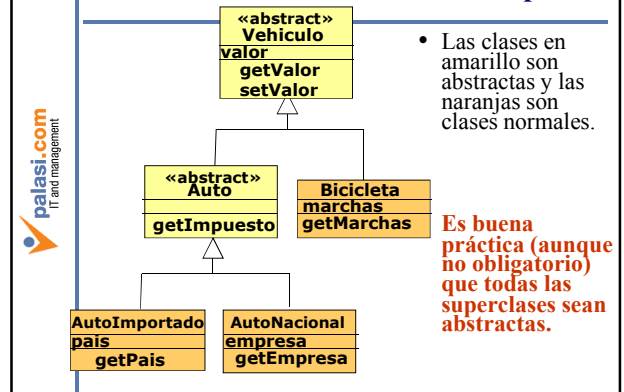




Todos los autos importados y nacionales forman parte de la clase **Auto**



Otro punto



Una observación

- De una clase abstracta no se puede hacer new.

Ejemplo

```
Vehiculo vehiculo = new AutoImportado();
Auto auto = new AutoNacional();
```

- Se instancian las subclasses y se pueden manipular con variables de las superclases.

Esto se parece mucho a las interfaces

```
Vehiculo vehiculo = new AutoImportado();
Auto auto = new AutoNacional();
```

- Las interfaces y las superclases son parecidas en unos aspectos.
- Las dos se llaman supertipos.
- Las dos contienen objetos de otros tipos.

Pero hay muchas diferencias

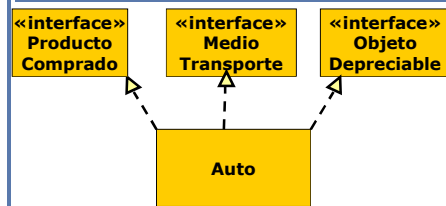
- Las clases abstractas son clases: pueden tener miembros estáticos, atributos e implementaciones de métodos de instancia.
- Las interfaces no son clases: son colecciones de definiciones de métodos de instancia.



### Pero hay muchas diferencias

- La herencia permite que las subclases hereden atributos y métodos de las superclases.
- Las clases que implementan las interfaces no heredan nada más que el nombre y tipo de los métodos.

### Una clase puede implementar todas las interfaces que quiera



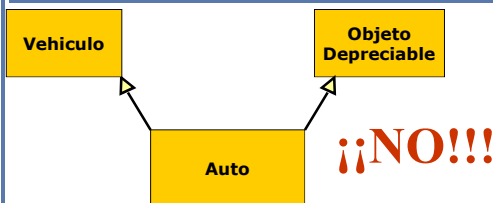
- En código sería:

```

public class Auto implements
 ProductoComprado, MedioTransporte,
 ObjetoDepreciable {
 //Aquí el código de la Clase
}

```

### Una clase no puede heredar de más de una superclase



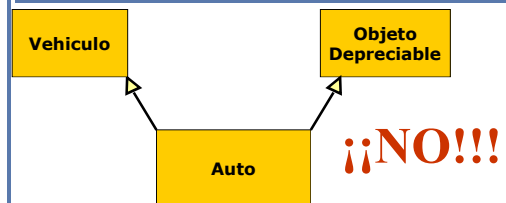
- En código no se puede hacer

```

public class Auto extends
 Vehiculo, ObjetoDepreciable {
 //Aquí el código de la Clase
}

```

### ¡¡ Padre no hay más que uno !!



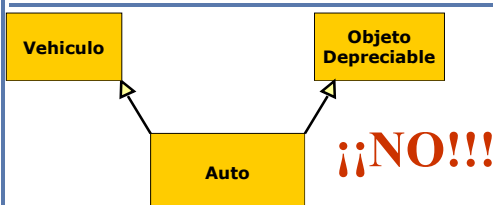
- En código no se puede hacer

```

public class Auto extends
 Vehiculo, ObjetoDepreciable {
 //Aquí el código de la Clase
}

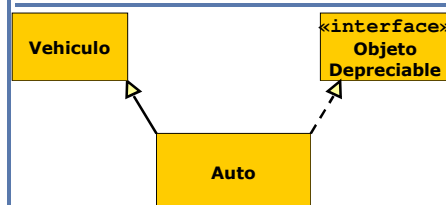
```

### A esto se le llama herencia múltiple



- Existe en otros lenguajes (como C++) pero dar muchos problemas. Por ejemplo, si vehículo y objeto depreciable tienen diferentes implementaciones del mismo método, ¿cuál heredará **Auto**?

### Lo que sí se puede hacer es lo siguiente



- Tener una herencia y todas las interfaces que queramos.
- Recordemos que de las interfaces no se hereda nada, sino que se implementan sus definiciones de métodos.



### \*Cómo se elige la superclase

- Informar-nos.
- Crec que la superclasse és de totes les candidates aquella que
- Mai la subclasse va a deixar de ser ella.
  - Un Auto sempre serà un Vehiculo, però una Persona no serà sempre un Empleat.
- Volem fer reutilització de codi: de la majoria dels mètodes de la superclasse. Quan més mètodes millor.
- Volem fer que canvis a la superclasse produïsquen canvis automàtics a la subclasse.

### 3.9. Herencia

- 3.9.1. Introducción a herencia.
- 3.9.2. Jerarquías de herencia.
- 3.9.3. **Métodos abstractos y sobreescripción.**
- 3.9.4. Conversión de tipos y herencia.
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.

### Ahora las leyes cambian

- Hasta ahora, los autos importados y los nacionales tenían un 10% de impuesto a partir de los 1000 dólares.
- Ahora las leyes cambian:
  - Los autos nacionales conservarán el mismo tipo de impuesto.
  - Los autos importados se les aplicará un 20% de impuesto a partir de los 500 dólares.

### Ahora tenemos que

- **Auto** tiene un método **getImpuesto()**, cuya implementación es diferente para **AutoImportado** y **AutoNacional**.
- De alguna manera, necesitamos que **Auto** defina la cabecera (nombre y parámetros) de sus métodos y que **AutoImportado** y **AutoNacional** definan su cuerpo (implementación).
- Para ello, necesitamos una nueva clase de método, llamado método abstracto.

### Ahora la clase Auto queda así

```
public abstract class Auto extends Vehiculo{
 public abstract double getImpuesto();
}
```

Un método abstracto es un método que se define en las superclases y se implementa en las subclases (o en la descendencia en general).

### Las clases AutoImportado quedan así

```
public class AutoImportado extends Auto{
 protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public Pais getPais(){
 return this.pais;
 }
 public double getImpuesto(){
 return 0.20*(this.valor-500);
 }
}
```



### Las clases AutoNacional queda así

```
public class AutoNacional extends Auto{
 protected Empresa empresa;
 public AutoNacional(Empresa empresa){
 this.empresa = empresa;
 }
 public Empresa getEmpresa(){
 return this.empresa;
 }
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
}
```

### Métodos abstractos

```
public abstract class Auto extends
Vehiculo{
 public abstract double getImpuesto();
}
```

- El método lleva la palabra **abstract** (que va después de **static** si existe y antes del tipo del resultado).
- Hasta ahora, decíamos que las clases abstractas eran convenientes, pero no obligatorias.
- Sin embargo, si una clase tiene métodos abstractos debe ser abstracta obligatoriamente y no se podrá hacer **new** con ella.

### Esto es parecido al método de una interfaz

```
public abstract class Auto extends
Vehiculo{
 public abstract double getImpuesto();
}
```

- Sin embargo, es diferente en varios aspectos.
- Primero, la clase abstracta no es como una interfaz pues puede tener atributos e implementación de métodos.
- Por ejemplo, ¿qué atributos e implementación de métodos tiene la clase abstracta **Auto**?

### Los que hereda de la clase Vehiculo

- Todo lo que está en rojo.

```
public abstract class Vehiculo {
 protected double valor;
 public double getValor(){
 return this.valor;
 }
 public void setValor (double valor){
 this.valor = valor;
 }
}
```

### Métodos abstractos

```
public abstract class Auto extends
Vehiculo{
 public abstract double getImpuesto();
}
```

- Si una clase no implementa todos los métodos abstractos que hereda entonces la clase será abstracta y no se podrá hacer **new** con ella.

### Métodos abstractos

```
public abstract class Auto extends
Vehiculo{
 public abstract double getImpuesto();
}
```

- Dicho de otra manera, si un método es abstracto, **todas las clases** que son **descendientes** de esta deben implementarlo **si no quieren ser abstractas**.



### Otra solución a este problema

- Hasta ahora, los autos importados y los nacionales tenían un 10% de impuesto a partir de los 1000 dólares.
- Ahora las leyes cambian:
  - Los autos nacionales conservarán el mismo tipo de impuesto.
  - Los autos importados se les aplicará un 20% de impuesto a partir de los 500 dólares.

### Sobreescritura de métodos

```
public abstract class Auto extends Vehiculo{
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
}
```

- Esta es otra solución. Usar un método no abstracto con una implementación por defecto.

### La clase AutoNacional queda así

```
public class AutoNacional extends Auto{
 protected Empresa empresa;
 public AutoNacional(Empresa empresa){
 this.empresa = empresa;
 }
 public Empresa getEmpresa(){
 return this.empresa;
 }
}
```

- Ya no hace falta incluir la implementación de **getImpuesto**, pues la heredamos de **Auto** y ya es la buena **return 0.10\*(this.valor-1000);**

### La clase AutoImportado queda así

```
public class AutoImportado extends Auto{
 protected Pais pais;
 public AutoImportado(Pais pais){
 this.pais = pais;
 }
 public Pais getPais(){
 return this.pais;
 }
 public double getImpuesto(){
 return 0.20*(this.valor-500);
 }
}
```

- Esta clase sí necesita dar una implementación de **getImpuesto**, pues la calcula de manera diferente de la que recibe de la superclase.

### Sobreescritura de métodos

```
public abstract class Auto extends Vehiculo{
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
}
public class AutoImportado extends Auto{
 //Otros atributos y métodos
 public double getImpuesto(){
 return 0.20*(this.valor-500);
 }
}
```

- Fijémonos que **AutoImportado** cambia la implementación que recibe de **Auto**.

### Sobreescritura de métodos

```
public abstract class Auto extends Vehiculo{
 public double getImpuesto(){
 return 0.10*(this.valor-1000);
 }
}
public class AutoImportado extends Auto{
 //Otros atributos y métodos
 public double getImpuesto(){
 return 0.20*(this.valor-500);
 }
}
```

- Se dice que **AutoImportado** **sobreescribe** ("overrides") el método **getImpuesto**.



### Métodos abstractos o sobrescritura de métodos

- Solución al mismo problema: métodos de las superclases que pueden tener diferentes implementaciones de las subclases.
- **Métodos abstractos.** Si queremos obligar a que cada subclase dé su implementación del método.
- **Sobrescritura de métodos.** Si queremos dar una implementación por defecto del método, que las subclases pueden cambiar o no.

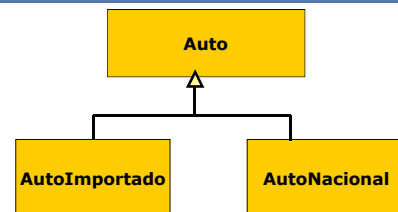
### Una opinión personal

- La sobrescritura de métodos no debería de usarse nunca.
- Hace el código poco legible y poco mantenible.
- Las superclases deberían implementar sólo los métodos que están seguros de que todas sus subclases heredarán. En caso contrario, se debe dejar la implementación del método a las subclases.
- Por ejemplo, si la subclase implementa un método que hace algo diferente que la superclase se viola el Principio de Sustitución de Liskov: una subclase debería poder usarse en lugar de su superclase.
- Más problemas en 153 de R.J.

### 3.9. Herencia

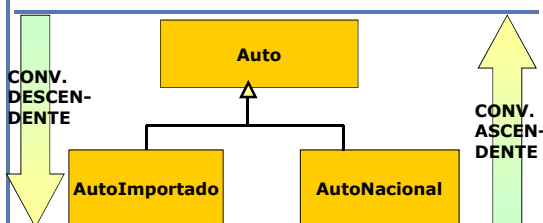
- 3.9.1. Introducción a herencia.
- 3.9.2. Jerarquías de herencia.
- 3.9.3. Métodos abstractos y sobrescritura.
- **3.9.4. Conversión de tipos y herencia.**
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.

### Conversión de tipos y herencia



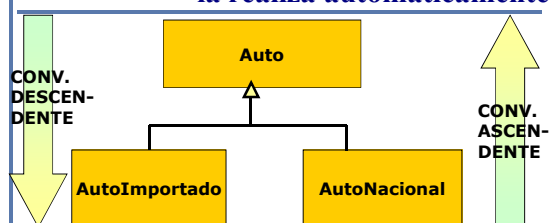
- En inglés, se llama “Casting”.
- Es parecido a la conversión de tipos con interfaces.

### Conversión de tipos y herencia



- **Conversión ascendente (“upcasting”)** es cuando transformamos una subclase en una superclase.
- **Conversión descendente (“downcasting”)** es cuando transformamos una superclase en una subclase.

### La conversión ascendente Java la realiza automáticamente



- No hace falta escribir ningún código especial.
- Veamos unos ejemplos.

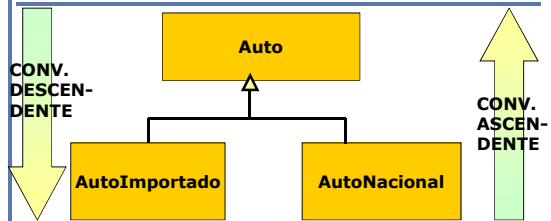


### Conversión ascendente

```
Auto auto1 = new AutoImportado(pais1);
```

- Como vemos, la conversión ascendente es automática.
- Supongamos que tenemos el método  
`public void agregaAuto(Auto a1)`
- Como vemos, un método que acepta un **Auto**, como parámetro puede aceptar un **AutoImportado** sin ningún problema.
- La conversión entre **AutoImportado** y **Auto** se realiza automáticamente, pues es ascendente.

### La conversión descendente no se realiza automáticamente



- Se necesita el operador de “casting”, que se usa así:

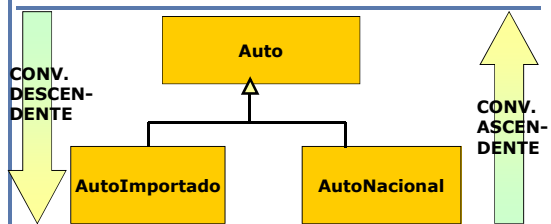
**(TipoClase)expresion**

### Conversión descendente

```
AutoImportado auto2 = (AutoImportado)auto1
```

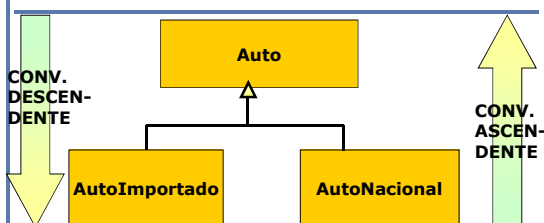
- Donde **auto1** es de tipo **Auto**.
- Es decir, la conversión entre **Auto** y **AutoImportado** (descendente) necesita un operador de casting.
- En el caso de no poner el operador de casting, se produce un error de compilación.

### Conversión de tipos y herencia



- La conversión ascendente (“upcasting”) siempre se puede hacer. Siempre podemos transformar un **AutoImportado** en un **Auto**, pues **Auto** es una subclase.
- Conversión descendente (“downcasting”) no siempre se puede hacer.

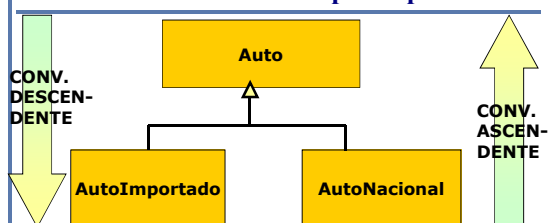
### Conversión de tipos y herencia



```
AutoImportado auto1 = new AutoImportado();
Auto auto2 = auto1;
AutoImportado auto3 = (AutoImportado)auto2;
```

**OK**

### La conversión descendente no siempre se puede hacer



```
AutoImportado auto1 = new AutoImportado();
Auto auto2 = auto1;
AutoNacional auto3 = (AutoNacional)auto2;
```

**ERROR**



### Conclusión

- Los objetos tienen una identidad.
- Si un objeto se creó (**new**) como **AutoImportado**, el objeto es un auto importado aunque lo podamos ver (referenciar) como un **Auto**.
- Un objeto auto importado puede usarse como auto importado y como auto y vehiculo.
- ¡¡Pero no puede utilizarse como auto nacional!!
- En general, un objeto puede verse como la clase que se creó (**new**) y todas sus ascendientes.

### 3.9. Herencia

- 3.9.1. Introducción a herencia.
- 3.9.2. Jerarquías de herencia.
- 3.9.3. Métodos abstractos y sobreescritura.
- 3.9.4. Conversión de tipos y herencia.
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.

### Herencia y constructores

- Los constructores no se heredan.
- Sin embargo, se puede llamar al constructor de la superclase usando **super (parámetros)**
- Así, por ejemplo, una clase abstracta puede tener un constructor:
  - No para hacer **new** porque no se puede con una clase abstracta.
  - Sino porque podemos llamarla de las subclases usando **super**.

### Ejemplo de super

```
public abstract class Persona {
 private String nombre;
 public Persona(String nombre) {
 this.nombre = nombre;
 }
}

public class Alumno extends Persona {
 private int curso;
 public Alumno(String nombre, int curso) {
 super(nombre);
 this.curso = curso;
 }
}
```

### Herencia y constructores

- Una clase que tiene como constructor **private** no puede tener subclases.
- Como conclusión, el singleton no puede tener subclases (lo que es un inconveniente).

### Acceso de miembros (atributos y métodos)

- **public**. El acceso al miembro es público. Se puede utilizar el miembro en cualquier circunstancia.
- **protected**. El miembro es accesible desde la clase que lo define, las clases que la extiendan y las clases del mismo paquete.
- Sin modificador. El miembro es accesible dentro del mismo paquete.
- **private**. El miembro sólo puede ser usado dentro de la clase que lo define, lo que impide su uso desde otras clases.



Modificadores de acceso (atributos y métodos)				
Resumiendo: tenemos el siguiente cuadro				
Acceso desde	Misma clase	Clases mismo paquete	Subclases	Clases otro paquete
<b>public</b>	OK	OK	OK	OK
<b>protected</b>	OK	OK	OK	
<b>(nada)</b>	OK	OK		
<b>private</b>	OK			

### Mandamiento del mantenimiento

*Harás los miembros lo más privados posible*

- Los atributos no privados se deben usar con mucha prudencia.
- Los miembros deben ser lo más privados posibles.



### Mandamiento del mantenimiento

*Harás los miembros lo más privados posible*

- Hacer los miembros privados minimiza la posibilidad de errores y facilita el mantenimiento.



### Unas guías

- Los atributos deberían ser privados siempre.
- Esto impide que los atributos sean heredados y sean corrompidos por las subclases.
- Hasta ahora, cuando hemos hablado de herencia hemos usado **protected** por motivos didácticos, pero no se debería hacer así.

### Unas guías

- Los métodos deberían ser privados si sólo son usados por la misma clase.
- Deben ser protegidos (**protected**) si deben ser usados por las subclases pero no por otras clases.
- Si deben ser usados por otras clases deberían ser públicos.
- En caso de duda, es mejor privado que protegido y protegido que público.

### La palabra final

- Si se pone a una clase, significa que esta clase no puede tener subclases.
- Si se pone a un método, indica que este método no se puede sobrescribir.



### Paréntesis: hay tantas palabras para declarar un método

**public** **static** **final**  
**parámetros**  
**nombreMetodo**  
**abstract** **void**

### ¿En qué orden van?

**public** **static** **final**  
**parámetros**  
**nombreMetodo**  
**abstract** **void**

### Esquema de la declaración del método

- Las partes opcionales van entre corchetes

```
[ámbito] [static] [abstract|final]
tipores nombre (listaparam) {
 Sentencias
}
```

- abstract** y **final** son alternativas: no pueden darse las dos a la vez.

### La clase Object

- Es una clase universal en Java. Todas las clases que no son subclases de nada son subclases de **Object**.

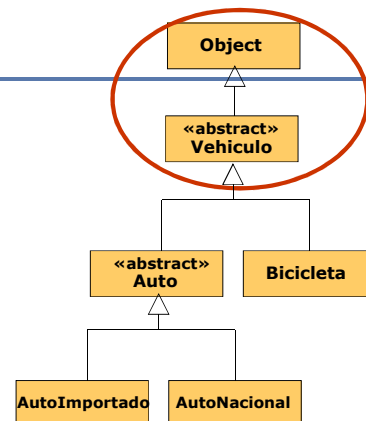
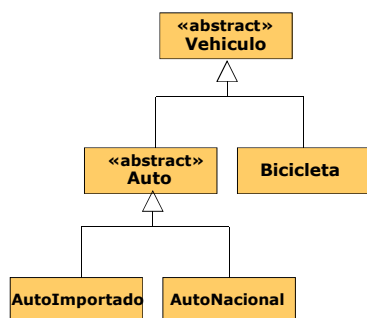
- Así cuando escribimos

```
public class NombreClase {
```

- Es como poner

```
public class NombreClase extends
Object {
```

### Recordemos esta jerarquía de clases





### Toda clase que no extienda nada extiende a Object

- Así toda clase o es subclase de **Object**, o es subclase de subclase de **Object**, o es subclase de subclase de subclase de **Object**.
- Así toda clase es descendiente de **Object** en última instancia (**Object** es el "Adán" de las clases).
- Una referencia de clase **Object** puede apuntar a cualquier objeto de cualquier clase.

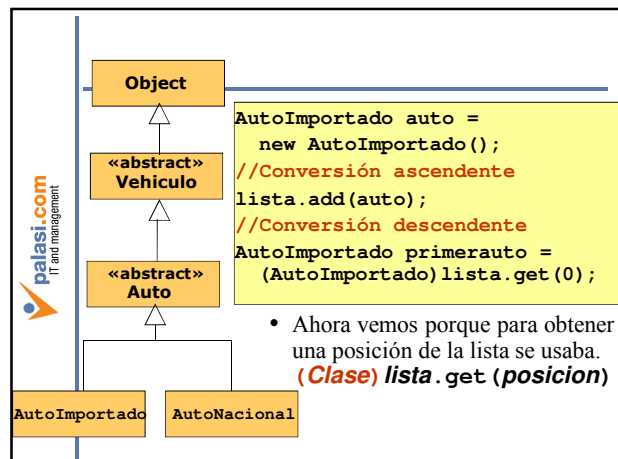
### Entendiendo las cosas

- Recordemos que una lista está compuesta por elementos.
- ¿De qué tipo son estos elementos?
- Son de tipo **Object**.
- Esto quiere decir que cualquier tipo se puede poner en una lista, pues todos descienden de **Object**.

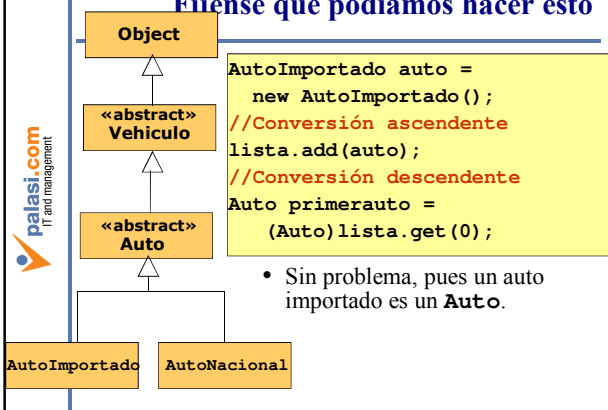
### Veamos un poco de código

```
public interface List{
//Aquí más definiciones de métodos.
public void add(Object elemento);
public Object get(int posicion);
}
```

- Es decir, se puede obtener `lista.add(0)` donde `0` es de clase **Object**.  
`lista.get(posición)` devuelve un **Object**.



### Éfiense que podíamos hacer esto



### 3.9. Herencia

- 3.9.1. Introducción a herencia.
- 3.9.2. Jerarquías de herencia.
- 3.9.3. Métodos abstractos y sobrescritura.
- 3.9.4. Conversión de tipos y herencia.
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.



## Mandamiento del mantenimiento

*Intentarás programar con supertipos.*

- Los supertipos son tanto las interfaces como las superclases.
- Ahora bien, ¿qué usamos interfaces o superclases (herencia)?



## ¿Cuándo usar herencia o interfaces?

- Lo veremos en tres fases.
- ¿Cuándo se necesita polimorfismo?
- ¿Cuándo se necesita reutilización de código?
- ¿Cuándo usamos herencia o interfaces?

## ¿Cuándo se necesita polimorfismo? Si se necesita

- Tratar un mismo objeto de varias formas diferentes
  - **Auto** como auto o como producto comprado.
  - **AutoImportado** como autoimportado o como auto.
- Tratar instancias de diferentes clases con la misma manera.
  - **Auto** y **TanqueGasolina** como producto comprado.
  - **AutoImportado** y **AutoNacional** como auto.

## ¿Cuándo se necesita reutilización de código?

- Cuando varias clases comparten el mismo código.
- Se desea que este código no se repita para actualizarlo sólo en un punto y facilitar el mantenimiento.

## ¿Cuándo usar?

- Si la clase ya tiene una superclase:
  - **Interfaces.** Pues no puede haber varias superclases.
- Si la clase no tiene una superclase:
  - **Interfaces.** Cuando sólo se necesita polimorfismo.
  - **Herencia.** Cuando se necesita polimorfismo y reutilización de código.
- Si hay dudas, interfaces.

## Una nota

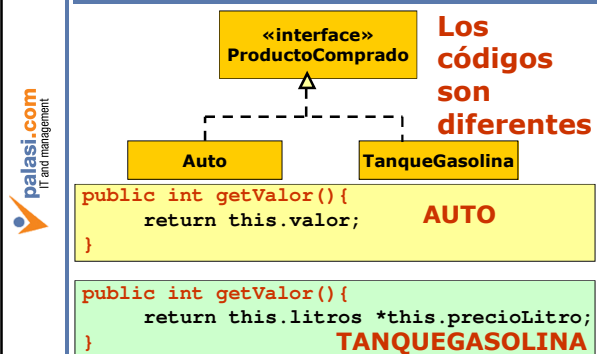
- Las jerarquías complejas de herencia no son recomendables.
- Si las tenemos, debemos considerar si no sería mejor usar el patrón Decorator que se basa en las interfaces.



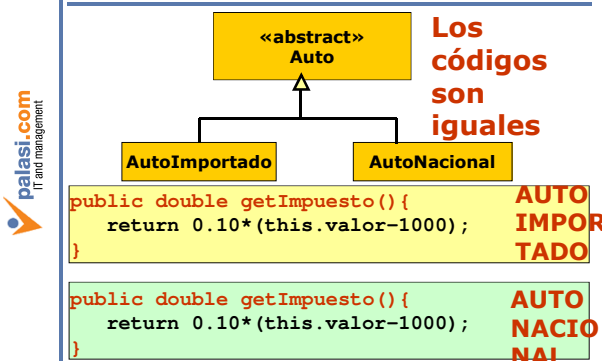
### Ejemplo

- Veamos los ejemplos que conocemos y por qué utilizamos interfaces o herencia.

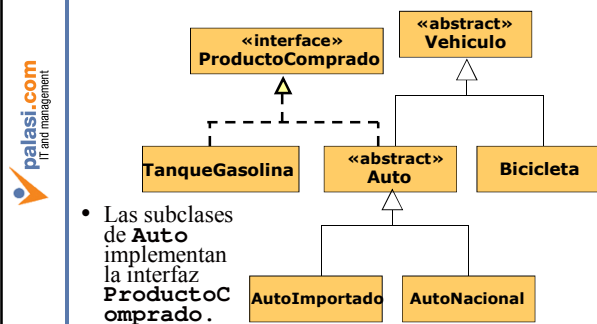
### Entre Auto y TanqueGasolina queremos polimorfismo pero no reutil. de código



### Entre AutoImportado y AutoNacional queremos polimorfismo y reutil. de código



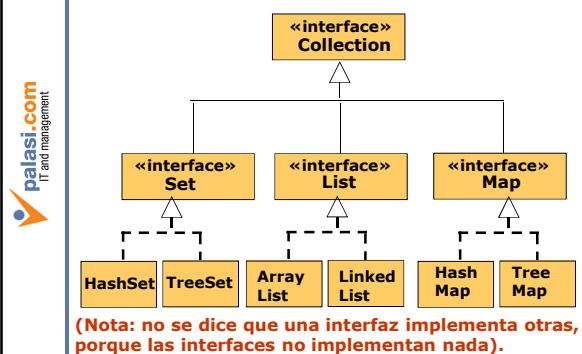
### Se pueden combinar los dos



### Otra guía

- Programamos la jerarquía de herencia pensando en **QUÉ ES** el objeto.  
Así, una **Computadora** es una **Maquina**
- Programamos la jerarquía de interfaces pensando en **QUÉ SE PUEDE HACER** (como se puede interactuar con) el objeto.  
Así, una **Computadora** es un **ObjetoComprable** (se puede comprar) **ObjetoFisico**, **MaterialDeOficina**, **ActivoFijo**, **ObjetoAmortizable**, etc.

### Las interfaces pueden extender otras interfaces





### Las interfaces pueden extender varias interfaces

- Al contrario de las clases, que sólo pueden extender una clase.
- El “padre no hay más que uno” se aplica a las clases, no a las interfaces.
- Sin embargo, que una interfaz extienda más de una interfaz no es tan común.

### 3.9. Herencia

- 3.9.1. Introducción a herencia.
- 3.9.2. Jerarquías de herencia.
- 3.9.3. Métodos abstractos y sobrescritura.
- 3.9.4. Conversión de tipos y herencia.
- 3.9.5. Otros aspectos sobre herencia.
- 3.9.6. Cuando usar herencia.
- 3.9.7. Herencia e Hibernate.

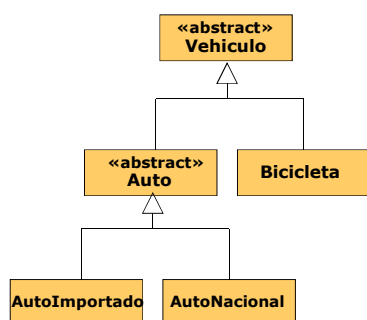
### ¿Qué quiere decir implementar herencia en Hibernate?

- Quiere decir que podemos guardar y recuperar objetos de la base de datos tanto de la superclase como de la subclase.
- Por ejemplo:
  - Podemos recuperar todos los autos.
  - Podemos recuperar los autos importados.

### Nota

- Usar herencia no quiere decir que necesitamos implementarla en Hibernate.
- Por supuesto, las clases y las subclases que no se persistan pueden usar herencia sin necesidad de implementarla en Hibernate.
- Incluso si las clases se persisten sólo implementamos si queremos acceder en la base de datos tanto con la superclase como la subclase.

### Recordemos el concepto de jerarquía



- Todas las clases que heredan de una clase que sólo hereda de **Object**.

### Tres estrategias para implementar la herencia en un motor de persistencias

- Tal como se explican en el artículo famoso de Scott Ambler, “Mapping Objects to Relational Databases”.
- **Tabla por cada clase concreta.** Se crea una tabla por cada clase concreta. Para las clases abstractas e interfaces no se crean tablas.
- **Tabla por cada subclase.** Cada clase tiene una tabla donde guarda los atributos que no ha heredado. Si A es subclase de B, la tabla de A tiene una clave foránea a la tabla de B.
- **Tabla por jerarquía.** Toda la jerarquía se pone en una sola tabla. Esto significa desnormalizar y tener un campo (llamado “discriminador”) que indica de que clase es cada registro.



### Hibernate implementa las tres estrategias

- **Tabla por cada clase concreta.** La más intuitiva. Sin embargo, tiene problemas cuando se consultan las interfaces o las superclases o si hay una composición a una interfaz o una superclase.
- **Tabla por cada subclase.** La más apropiada para jerarquías complejas. Es un poco más difícil para reportes, pero se puede solucionar con una vista que la transforme en la tercera modalidad y sobre la que se hagan los reportes.
- **Tabla por jerarquía.** La más apropiada para jerarquías simples. Más rápida y más fácil para reportes.

### Una guía

- Como defecto, se toma la estrategia “tabla por jerarquía”.
- Si las jerarquías son complejas o las clases difieren mucho en sus atributos:
  - Se considera usar “tabla por subclase”.
  - O bien se considera usar el patrón Decorator en vez de herencia, pues las jerarquías complejas de herencia no son recomendables.

### Aquí solo veremos tabla por jerarquía

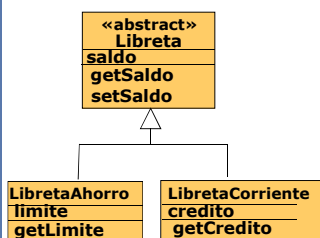
- Por falta de tiempo.
- Las otras pueden verse en la documentación de Hibernate.

### Tabla por jerarquía

- Toda la jerarquía se pone en una única tabla que tiene como campos:
  - Los atributos de todas las clases.
  - Un campo llamado discriminador que nos dirá de qué clase concreta es cada registro.

### Ejemplo: Vamos a persistir esta jerarquía de herencia

- Utilizaremos la estrategia “clase por jerarquía”.



### Toda la jerarquía está en una única tabla

- Esta tabla tiene los atributos de todas las clases de la jerarquía.
- Tiene un campo discriminador que dice de qué clase es cada registro.

libreta				
1	1000	5000	NULL	AHORRO
2	1500	NULL	3000	CORRIENTE
54	2000	NULL	6000	CORRIENTE
55	2000	5500	NULL	AHORRO

↑ Id    ↑ Saldo    ↑ Limite    ↑ Credito    ↑ Disc

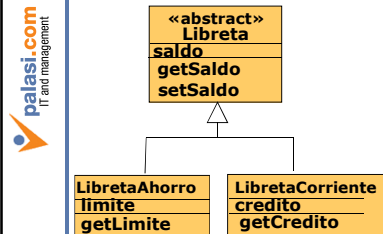


### Se debe poner en un archivo de correspondencia

```
<hibernate-mapping>
<class name="ClaseRaizJerarquia" table="tabla" >
 <id ...
 </id>
 <discriminator column="coldisc" type="string"/>
 //Todos los atributos de la superclase
 //incluyendo composición
 <subclass name="Subclase1" discriminator-
 value="ValorDiscSubclase1">
 //La descripción de la subclase 1, con
 //atributos, composiciones y subclases
 </subclass>
 //Lo mismo para las otras subclases
</class>
</hibernate-mapping>
```

### Ejemplo: Vamos a persistir esta jerarquía de herencia

- Utilizaremos la estrategia "clase por jerarquía".



### Libreta.hbm.xml

```
<hibernate-mapping>
<class name="Libreta" table="libreta">
 <id name="id" type="int" access="field">
 <column name="id"/>
 <generator class="identity"/>
 </id>
 <discriminator column="disc" type="string"/>
 <property name="saldo" access="field"/>
 <subclass name="LibretaAhorro" discriminator-
 value="AHORRO">
 <property name="limite" access="field"/>
 </subclass>
 <subclass name="LibretaCorriente" discriminator-
 value="CORRIENTE">
 <property name="credito" access="field"/>
 </subclass>
</class>
</hibernate-mapping>
```

### ¿Qué pasa si tenemos una jerarquía de más de dos niveles?

- Simplemente, en el archivo de configuración, ponemos una **<subclass>** dentro de otra **<subclass>**

### \*Consultas de HQL en herencia

### Paréntesis: Plugins de Eclipse para Hibernate

- Hasta ahora hemos escrito las clases persistentes, la base de datos y los archivos de configuración a mano.
- Esto es sencillo, pero no tiene por qué ser así.
- Se pueden generar automáticamente los archivos de configuración con herramientas como Ant, XDoclet.
- Incluso si utilizamos Hibernate Annotations, los archivos de configuración no tienen porque existir.
- También Hibernate incluye herramientas de línea de comandos como SchemaExport y SchemaUpdate que, a partir de los archivos de configuración, construyen el DDL de la base de datos.
- También se pueden usar plugins de Eclipse para Hibernate, que no veremos aquí por falta de tiempo.



### Paréntesis: Plugins de Eclipse para Hibernate

- **Hibernate Synchronizer.** <http://www.binamics.com/hibernatesync/>
  - Crea y actualiza las clases Java a partir de los archivos de configuración.
  - Adecuado si se sigue un enfoque basado en los archivos.



### Paréntesis: Plugins de Eclipse para Hibernate

- **Hiberclipse** <http://hiberclipse.sourceforge.net/>
  - Genera y actualiza los archivos de configuración a partir de la base de datos.
  - Es más adecuado con un enfoque centrado en la BD, sobre todo, cuando queremos aplicar Hibernate a una base de datos preexistente.



### Paréntesis: Plugins de Eclipse para Hibernate

- **Hibernator.** <http://hibernator.sourceforge.net/>
  - Va en la dirección opuesta, creando y actualizando los archivos de configuración a partir de las clases persistentes y, a partir de allí, generando la base de datos.
  - Es más adecuado con un enfoque centrado en el código Java, que es el que hemos utilizado aquí.
  - Descontinuado.



### Paréntesis: Plugins de Eclipse para Hibernate

- **Hibernate Tools.** <http://www.hibernate.org/255.html>
- Crea y actualiza archivos de configuración a partir de las clases Java.
- Crea y actualiza archivos de configuración y clases Java a partir de la base de datos.
- Permite múltiples enfoques.
- En versión alfa.



### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- **3.10. El patrón Decorator.**
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### Cuando usar el patrón Decorator y cuando usar herencia

- Documentació.
- Documents adjunts a aquesta presentació. "Why extends is evil", carpeta Interfaces.
- Documents en paper. Prefer composició over inheritance del llibre de Bloch, llibre de Rod Johnson, llibre de Head First Design Patterns.



### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### El patrón Singleton tiene una serie de problemas

- 1. Como hemos visto, la clase Singleton no puede tener subclases.
- 2. Hay un principio “el principio de única responsabilidad” que dice que una clase solo debería ocuparse de una responsabilidad para que sea fácil de mantener. Pero la clase Singleton se ocupa dos: de lo que haga su código y de que sólo haya una clase de ella.
- 3. Hemos dicho que “programaremos con supertipos” (interfaces o clases abstractas). Sin embargo, el singleton no funciona bien con los supertipos, ya que **Clase.getInstance() sólo funciona con una clase concreta** (las interfaces no tienen métodos estáticos y las clases abstractas no tienen new para crear la única instancia).

### El patrón Singleton tiene una serie de problemas

- 4. El código no es flexible pues llamar a un singleton, es siempre llamar a una clase concreta. Así, si queremos cambiar el singleton por otra clase con la misma interfaz (por efectos de testing, refactoring), deberemos cambiar todas las llamadas al singleton (lo que a veces es tedioso y a veces imposible).
- 5. Estos problemas se acentúan porque el singleton es muy tentador y hay tendencia a usarlo en muchas partes de nuestra aplicación.

### ¿Qué es lo que queríamos del singleton?

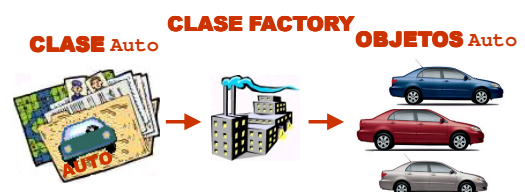
- Lo que queríamos era que la clase que lo implementaba tuviera una sola instancia.
- Esto se puede conseguir con otros patrones.
- Nosotros usaremos el patrón Factory. Concretamente, una versión de Factory que Martin Fowler llama “Registry” en su libro “Patterns of Enterprise Application Architecture”.

### El patrón Factory

- La idea del patrón Factory es sencilla y tiene que ver con la creación de objetos.
- Hasta ahora, los objetos se creaban con **new**. Había **new** en muchas clases diferentes y, por lo tanto, la creación de objetos estaba dispersa.
- El patrón Factory se trata de concentrar la creación de varios objetos en una sola clase, llamada Factory porque es una “fábrica” de objetos.

### La clase Factory

- Es una clase cuyo fin es la creación de objetos de otras clases.





### Código simplificado de la clase Factory (versión sólo para clases con única instancia)


```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClass1() {
 return this.obj1;
 }
 public Clase2 getClass2() {
 return this.obj2;
 }
 ...
}
```

### Se ve que la Factory es un singleton. Sólo hay un objeto Factory

```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClass1() {
 return this.obj1;
 }
 public Clase2 getClass2() {
 return this.obj2;
 }
 ...
}
```


### Ese único objeto Factory sólo tiene un único objeto de cada clase

```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClass1() {
 return this.obj1;
 }
 public Clase2 getClass2() {
 return this.obj2;
 }
 ...
}
```



### Cada método devuelve ese único objeto de una clase

```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClass1() {
 return this.obj1;
 }
 public Clase2 getClass2() {
 return this.obj2;
 }
 ...
}
```



### Es claro que, si sólo se usa el objeto Factory, sólo hay 1 objeto de cada clase

```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClass1() {
 return this.obj1;
 }
 public Clase2 getClass2() {
 return this.obj2;
 }
 ...
}
```

### ¿Está loco? ¿Para solucionar los problemas del singleton usamos un singleton?

- Es cierto, pero hay diferencias:
  - Ahora sólo tenemos un singleton (la Factory) por cada capa. Antes podíamos tener decenas en cada capa.
  - La Factory es una clase de infraestructura: no va a necesitar ser una interfaz, una clase abstracta, tener subclases o cambiarse por otra clase para testing o refactoring. Va a estar siempre así. Por ello, los inconvenientes del singleton no los tiene.
- Esta versión de la Factory a veces se llama Singleton Factory.



### Ahora ya no tenemos algunos inconvenientes del Singleton

- 1. Ahora las clases pueden tener subclases.
- 2. Se sigue “el principio de única responsabilidad”: cada clase se ocupa de su código y sólo la clase Factory se ocupa de que sólo se cree un objeto de las clases.
- Estos son los dos primeros inconvenientes del Singleton. Después veremos que los demás (3, 4 y 5) también desaparecen.

### Un ejemplo

- Tenemos un programa de banco. En su capa de datos, tenemos una serie de clases DAO, como **DAOCliente**, **DAOLibreta**, **DAOCredito**.
- Las clases DAO (de este programa y de todos los otros) no tienen atributos. Es decir, sólo necesitamos un objeto de cada una de ellas.
- Esto es bueno para ahorrar dramáticamente tiempo y memoria.
- Hasta ahora, hubiéramos usado un singleton para cada una. Ahora usaremos el patrón Factory.

### La clase se llamará FactoryDatos

```
public class FactoryDatos() {
 private static FactoryDatos instancia =
 new FactoryDatos();
 private DAOCliente daoCliente = new DAOCliente();
 private DAOLibreta daoLibreta = new DAOLibreta();
 private DAOCredito daoCredito = new DAOCredito();
 public static FactoryDatos getInstancia() {
 return FactoryDatos.instancia;
 }
 public DAOCliente getDAOCliente() {
 return this.daoCliente;
 }
 public DAOLibreta getDAOLibreta() {
 return this.daoLibreta;
 }
 public DAOCredito getDAOCredito() {
 return this.daoCredito;
 }
}
```

### Uso de la Factory

- Cada vez que queramos acceder al objeto único de una clase:

```
Factory factory = Factory.getInstancia();
Clase1 clase1 = factory.getClase1();
```

o bien

```
Clase1 clase1 =
 Factory.getInstancia().getClase1();
```

Por ejemplo, si queremos crear el DAOLibreta.

```
FactoryDatos factory = FactoryDatos.getInstancia();
DAOLibreta daoLib=factory.getDAOLibreta();
```

```
DAOLibreta daoLibreta =
 FactoryDatos.getInstancia().getDAOLibreta();
```

### ¿Qué pasa si ahora decidimos que la Clase1 queremos tener más de una instancia?

```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClase1() {
 return this.obj1;
 }
 public Clase2 getClase2() {
 return this.obj2;
 }
 ...
}
```

### Fácil. Simplemente hacemos

```
public class Factory() {
 private static Factory instancia = new Factory();
 //Ya no nos hace falta el atributo
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClase1() {
 return new Clase1();
 }
 public Clase2 getClase2() {
 return this.obj2;
 }
 ...
}
```



### Otras ventajas del patrón Factory

- No tiene que ser sólo para “singletons”: puede servir para construir cualquier clase.
- Fácilmente una clase puede convertirse de un singleton en una clase normal y viceversa.

### Si queremos que Clase1 vuelva a ser un singleton

```
public class Factory(){
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2();
 ...
 public static Factory getInstancia(){
 return Factory.instancia;
 }
 public Clase1 getClase1(){
 return this.obj1();
 }
 public Clase2 getClase2(){
 return this.obj2;
 }
 ...
}
```

Volvemos a poner el atributo y ya

### El resto del programa ni se entera de estos cambios

- Que una clase tenga una instancia o varias es interno a la clase Factory.
- En general, cualquier cuestión de creación de la clase es interna a la clase Factory.
- Podemos modificar cualquier aspecto de creación sin que el resto del programa se entere.

### Otro aspecto de creación que podemos cambiar

- Supongamos que ahora Clase2 queremos hacerla una interfaz para más flexibilidad.
- No queremos cambiar el resto del programa.
- Cambiaremos la Clase2 para que sea una interfaz.
- Necesitaremos una implementación de esta interfaz, una clase que llamaremos Clase2Impl.

### Se crea el objeto como la implementación pero se devuelve como la interfaz.

```
public class Factory(){
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 private Clase2 obj2 = new Clase2Impl();
 ...
 public static Factory getInstancia(){
 return Factory.instancia;
 }
 public Clase1 getClase1(){
 return this.obj1;
 }
 public Clase2 getClase2(){
 return this.obj2;
 }
 ...
}
```

### Si queremos varios objetos y no sólo uno

```
public class Factory(){
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 //Ya no necesitamos el atributo
 ...
 public static Factory getInstancia(){
 return Factory.instancia;
 }
 public Clase1 getClase1(){
 return this.obj1;
 }
 public Clase2 getClase2(){
 return new ClaseImpl();
 }
 ...
}
```



### Si ahora queremos variar la implementación de la interfaz

```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 //Ya no necesitamos el atributo
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClass1() {
 return this.obj1;
 }
 public Clase2 getClass2() {
 return new ClaseNuevaImpl();
 }
 ...
}
```

### Usando interfaces y Factory es muy fácil cambiar la implementación de una clase

```
public class Factory() {
 private static Factory instancia = new Factory();
 private Clase1 obj1 = new Clase1();
 //Ya no necesitamos el atributo
 ...
 public static Factory getInstancia() {
 return Factory.instancia;
 }
 public Clase1 getClass1() {
 return this.obj1;
 }
 public Clase2 getClass2() {
 return new ClaseNuevaImpl();
 }
 ...
}
```

**EL RESTO DEL  
PROGRAMA NI SE  
ENTERA**

### Otra ventaja

- Ahora la clase de la implementación (la clase concreta) sólo se conoce dentro de la Factory.
- Todo el resto del programa sólo usa la interfaz.
- Lo mismo se podría hacer con una clase abstracta en vez de la interfaz.
- Esto hace el programa más flexible: recordemos.

### Mandamiento del mantenimiento

*Intentarás programar  
con supertipos.*

- Es decir, con interfaces o clases abstractas.



### ¿Qué problema había con esto?

- Que siempre había que instanciar una clase concreta.

```
Interfaz objInterfaz = new Clase();
objInterfaz.metodo();
//El resto del programa usaba
objInterfaz y por lo tanto un
supertipo.
```

### ¿Qué problema había con esto?

- Aunque todo el programa usaba la interfaz, siempre teníamos el problema con el new.
- Si queríamos cambiar la clase concreta, debíamos cambiar el new.

```
Interfaz objInterfaz = new Clase();
objInterfaz.metodo();
//El resto del programa usaba
objInterfaz y por lo tanto un
supertipo.
```



### Ahora también debemos cambiar el new

- Pero todos los new están en la Factory, así que todo el resto del código queda independiente de la clase concreta.

```
Interfaz objInterfaz =
 Factory.getInstancia().getInterfaz();
objInterfaz.metodo();
//El resto del programa usaba
objInterfaz y por lo tanto un
supertipo.
```

### Si queremos cambiar el new lo haremos en la Factory (como hemos visto)

- El resto no cambiará. Mucho más mantenible.

```
Interfaz objInterfaz =
 Factory.getInstancia().getInterfaz();
objInterfaz.metodo();
//El resto del programa usaba
objInterfaz y por lo tanto un
supertipo.
```

### Mandamiento del mantenimiento

*Identifica los aspectos de tu aplicación que varían y sepáralos de las partes que permanecen igual.*

- En este caso, lo hacemos.



### Una de las partes de nuestra aplicación que varía es la creación (new)

- Porque podemos cambiar de clase (por ejemplo, para el testing).
- Porque podemos poner una interfaz o una clase abstracta donde había una clase concreta.
- Porque podemos hacer que sólo se devuelva un objeto, varios o muchos.

### Aplicando el mandamiento anterior

- Como la construcción de objetos varía, debemos separarla de las partes que no varían.
- La separamos en la Factory.

### Problemas del Singleton que aún no habíamos visto que la Factory resolvía

- 3. La factory puede trabajar con interfaces y clases abstractas, es decir, con supertipos.
- 4. El código es flexible. Si queremos cambiar el singleton con otra clase con la misma interfaz, sólo cambiamos la Factory y todo lo otro sigue igual.
- 5. Ya no hay proliferación de singletons, el único singleton es la Factory.



### ¿Qué clases deben crearse en la clase Factory?

- Hay una opinión que dice que todas las clases deberían crearse en la Factory.
- Esto da un máximo de flexibilidad pero es un poco incómodo.
- Hay frameworks como Spring que hacen esto mucho más cómodo, implementando factories más avanzadas que las que hemos visto aquí.

### En nuestro caso, tomaremos una posición moderada

- Todas las clases de las cuales queremos una única instancia las pondremos en la Factory (para no usar Singleton).
- Todas las clases de las que sea probable que su creación cambie en el futuro:
  - Porque puedan pasar de clase concreta a clase abstracta o interfaz.
  - Porque puede variar el número de instancias que tenemos de ellas.
  - Porque cualquier aspecto de su creación creemos que puede cambiar.

## 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### \*Excepciones

- Posar tot el material que es té i a més:
- Quan usar excepcions i quan uns altres mètodes d'error.
- Quan usar excepcions comprovades i no comprovades.
- Basar-se en el llibre de Bloch, Rod Johnson, Thinking in Java i en discussions de TheServerSide.com

## 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento

### \*Algunas clases del lenguaje Java

- Posar BigDecimal i unes altres que tinc de l'altre curs.



### Hay muchas clases que vienen programadas en el JDK

- No tenemos que programarlas: sólo usarlas.
- Pero, ¿cómo sabemos como se usan?

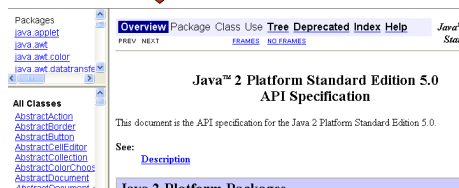
### Buscando información sobre las clases del JDK

- Se accede a <http://java.sun.com/reference/api/index.html> y se hace clic en J2SE 5.0.



### La pantalla aparece partida en tres partes.

Todos los paquetes que tiene el JDK



Todas las clases que tiene el JDK

### Si elegimos una clase

- En la parte principal, aparece toda su información.



### Ahora ya sabemos como consultarlas

- Porque sabemos todos los conceptos que usa esta documentación:
- Sobrecarga.
- Constructores.
- Atributos y métodos estáticos.
- Clases y métodos abstractos.
- Interfaces y herencia.

### 3. Ampliación del lenguaje Java

- 3.1. Sobrecarga y constructores.
- 3.2. Miembros estáticos.
- 3.3. El patrón Singleton.
- 3.4. Composición.
- 3.5. Arreglos.
- 3.6. Polimorfismo e interfaces.
- 3.7. Colecciones.
- 3.8. Composición con colecciones.
- 3.9. Herencia.
- 3.10. El patrón Decorator.
- 3.11. El patrón Factory.
- 3.12. Excepciones y manejo de errores.
- 3.13. Algunas clases del lenguaje Java.
- 3.14. Los mandamientos del mantenimiento



### \*Manaments del manteniment

- 1. Evitarás la repetición de código.
- 2. Encapsularás los datos y métodos relacionados.
- Harás todo lo más privado posible.
- 3. Intentarás programar con supertipos.
- 4. Separarás lo que varía de lo que no lo hace.
- 5. Programarás en n-capas
- Coupling, etc. (mirar anàlisi i disseny O-O).
- Mirar llibres de patrons i de Rod Johnson.

### Programa del curso

- 1. Objetivos y metodología del curso.
- 2. Repaso del curso anterior.
- 3. Ampliación del lenguaje Java.
- 4. Cookies y rastreo de sesión.
- 5. Otros aspectos.
- 6. Un ejemplo práctico.

### Ejecución de un programa

- Acciones que ejecuta un usuario en un programa
  - Desde que inicia el programa
  - Hasta que sale de él.

### En un programa hay dos tipos de información

- **Información permanente.** Es la que permanece entre ejecuciones. Se guarda en la base de datos o archivos.
- **Información transitoria.** Es la que sólo está disponible dentro de la misma ejecución. Se pierde cuando acaba la ejecución del programa.

En un programa tradicional se guarda en memoria y la llamaremos **estado de ejecución**.

### Ejemplos de información transitoria (informaciones en el estado de ejecución)

- La identidad del cliente (que se determina cuando se hace login en el programa).
- Datos que se han seleccionado anteriormente en la ejecución del programa. Por ejemplo, en un programa multiempresa, qué empresa ha seleccionado el cliente (todas las operaciones se harán en esa empresa hasta que se cambie).
- Otro ejemplo de lo anterior en la programación Web son los datos de la tarjeta de crédito que entramos antes de comprar en una tienda electrónica.

### Un problema en la programación Web

- Es determinar qué entendemos por ejecución.
- En un programa tradicional no hay problema. Acciones que ejecuta un usuario en un programa
  - Desde que inicia el programa
  - Hasta que sale de él.
- Sin embargo, en la programación Web es difícil saber:
  - Cuando se sale de un programa (muchas veces se cierra el navegador y el servidor ni se entera).
  - Cómo se determina cuál es el usuario.



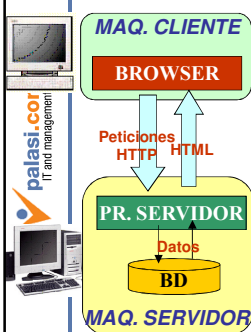
### ¿Qué es una ejecución de una aplicación Web?

- Se considera que la ejecución comienza cuando el usuario hace login.
- A veces se considera que la ejecución se acaba cuando la instancia del navegador se cierra. Otras veces no.
- Se considera que la ejecución también acaba cuando
  - el usuario hace clic en un botón de logout o bien
  - cuando ha pasado un tiempo desde la última vez que el usuario accedió la aplicación.
- Todas las acciones que un mismo usuario hace durante ese tiempo se le llama ejecución o, más propiamente, sesión.

### ¿Cómo determinar el usuario?

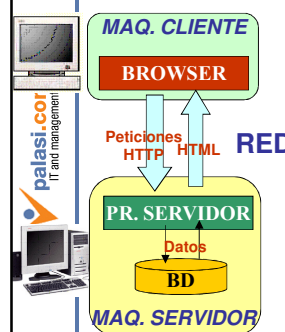
- Parecería lógico decir que el usuario lo determina la máquina. Pero:
  - ¿Qué pasa si la misma persona accede a la misma máquina pero quiere hacer procesos diferentes? (ejecuciones diferentes).
  - ¿Cómo distinguimos las diferentes máquinas? A veces, varias tienen la misma dirección IP porque vienen de un proxy.

### En la práctica se sigue esta política



- Cuando se hace login, se envía un archivito (cookie) al cliente que identifica al usuario.
- Cada vez que el cliente accede a la aplicación, envía la cookie que lo identifica.

### ¿Dónde se guarda el estado de ejecución en un programa Web?



- Hay dos sitios:
- En el cliente (en disco, toda con cookies).
- Una cookie en el cliente que identifica al usuario y todo lo otro en el servidor (en memoria). A esto se le llama rastreo de sesión.

### Cookies

- Pequeñas porciones de información sobre el estado de ejecución en forma de texto que
  - un servidor Web envía a un navegador cliente y que
  - el navegador devuelve sin cambiar cuando más tarde visita el mismo sitio Web.
- Las cookies se almacenan en el disco duro del cliente para que puedan ser conservadas.
- Las cookies se transmiten al servidor dentro de la petición y se reciben del servidor dentro de la respuesta.

### Cookies

- Las cookies están asociadas a las aplicaciones Web.
- Una cookie sólo se envía a la aplicación Web que la creó.
- Las cookies caducan después de un tiempo de creadas: tienen una duración máxima



### Rastreo de sesión

- Información transitoria (estado de ejecución) que se guarda en la memoria del servidor.
- La sesión caduca cuando se cierra el navegador o después de un tiempo de no operar la aplicación: tiene una inactividad máxima.

### ¿Cookies o API de rastreo de sesión?

- Los dos son métodos para mantener la información transitoria (el estado de ejecución) del cliente.
- Las cookies son de bajo nivel. Más difíciles de programar y sólo pueden guardar información textual.
- La API de rastreo de sesión es de alto nivel. Más fácil de programar y se puede guardar cualquier tipo de objeto.
- Las cookies guardan todo el estado de ejecución en el cliente.
- La API de rastreo de sesión sólo guarda un pequeño identificador de sesión en el cliente. El resto (prácticamente todo) del estado se guarda en el servidor.
- Con el rastreo, la sesión se acaba cuando se cierra el navegador. Con las cookies, no se acaba.

### Nota importante

- Tanto las cookies como el rastreo de sesión se ejecutan en la capa de presentación.
- Las otras capas ni se enteran.

### Ventajas de guardar la información en el cliente (cookies)

- **Menos uso de recursos.** Como la información se guarda en el cliente, el servidor puede requerir una menor cantidad de recursos (memoria, disco..)
- **Mejor escalabilidad.** Los servidores que no mantienen el estado de ejecución pueden agruparse (cluster) fácilmente, ya que cualquier servidor puede servir a cualquier cliente.
- **La sesión puede sobrevivir la caída del servidor.** Como toda la información se encuentra en el cliente, si cae un servidor, otro servidor puede hacerse cargo de la sesión.

### Desventajas de guardar la información en el cliente (cookies)

- **Acceso a información sensible por parte de otros usuarios** que comparten la misma máquina.
- Los **datos en el cliente pueden ser modificados**, por un usuario malicioso para acceder a los recursos de otros usuarios.
- **Sobrecarga de red**, ya que el estado del cliente debe ser enviado al servidor con cada petición y devuelto con cada respuesta.
- **Tamaño limitado** de datos para una cookie en algunos navegadores.
- **Implementación compleja.** Mantener el estado en el cliente puede ser muy complejo.

### Desventajas de guardar la información en el cliente (cookies)

- El usuario puede sentir que las cookies amenazan su privacidad, ya que los datos pueden quedar almacenados en una computadora que usa más de una persona.
- Por ello, el usuario puede desactivar las cookies en su navegador.
- Por ello, usaremos las cookies para añadir valor a nuestro sitio, pero no se puede depender exclusivamente de ellas, de forma que el sitio sea completamente interoperable sin cookies.



### Conclusión

- Las desventajas de guardar la información en el cliente (sobre todo, la implementación compleja) sobrepasan con mucho sus ventajas.
- Por ello, nuestro consejo es que la información o estado del cliente se guarde en el servidor usando la API de sesión.
- Aquí no veremos las cookies pero sí el rastreo de sesión.

### ¿Qué es una sesión?

- Es todo lo que hace el usuario en una aplicación Web desde que
  - entra con una contraseña
  - hasta que sale
    - Cerrando la instancia del navegador.
    - Haciendo logout.
    - Simplemente dejando de operar la aplicación durante un tiempo.
- Esta es la definición que habíamos visto hasta ahora.
- Por ejemplo entrando en Yahoo!Mail, Hotmail o cualquier otro sitio que requiera contraseña.

### Otra definición

- Una sesión es el conjunto de acciones que un cliente realiza con una aplicación Web y que:
  - están próximas en el tiempo,
  - el cliente percibe como una unidad y
  - tienen una única finalidad.
- Así, si el cliente quiere comprar dos libros, hará las siguientes acciones:
  - Buscará el primer libro.
  - Lo añadirá a la bolsa de la compra.
  - Buscará el segundo libro.
  - Lo añadirá a la bolsa de la compra.
  - Procederá a comprar.
- El conjunto de todas estas acciones constituirá una sesión.

### Una sesión es como una llamada de teléfono

- Cada frase que pronuncia uno de los interlocutores es aislada (igual que las peticiones HTTP).
- Pero todas están próximas en el tiempo y forman parte de la misma conversación, con una única finalidad.
- Así es una sesión. Los dos interlocutores son el cliente y el servidor.

### Rastreo de sesión

- Es la capacidad del servidor de identificar en qué sesión se encuentra el cliente y qué acciones ha realizado en esta sesión.
- Dicho de otra manera, es la capacidad del servidor para recuperar información sobre las acciones pasadas y recientes del usuario con el sitio Web.

### Ejemplo de rastreo de sesión

- En un sitio de comercio electrónico:
  - ¿Cómo sabe el servidor qué usuario está comprando?
  - Cuando un usuario añade algo a la bolsa de compra, ¿cómo sabe el servidor qué productos ya están en la bolsa?
  - Cuando un usuario decide comprar, ¿cómo sabe el servidor qué productos tiene en la bolsa?
- Para saber esto, el servidor debe tener información sobre las anteriores acciones del usuario con el servidor, sobre la sesión del usuario.
- Esta información debe ser diferente a cada usuario.



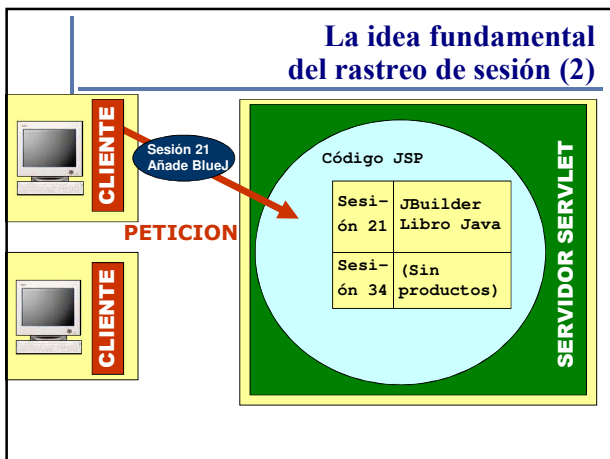
### El problema del rastreo de sesión

- Aunque el cliente percibe todas las acciones como parte de una sesión (por ejemplo, de compra), el servidor ve cada acción como una petición independiente que debe servir independientemente de las demás.
- Esto es porque HTTP es un protocolo “sin estado” (que no guarda información sobre el pasado).
- Por ello, el rastreo de sesión se debe implementar explícitamente, ya que HTTP no lo implementa por defecto.

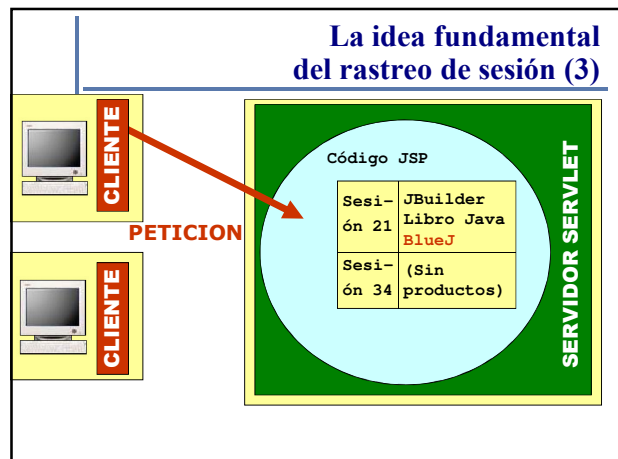
### La idea fundamental del rastreo de sesión (1)

- Para implementar el rastreo de sesión guardaremos una tabla del estilo **Map** (es decir, tipo diccionario) para todas las sesiones activas.
- Cada posición de la tabla tendrá como código el número de sesión (que deberá ser único) y como valor los datos que se guardan de cada sesión (en nuestro caso, los productos que ha colocado en la bolsa).
- Así, guardamos en el servidor la información relacionada con cada sesión y la vamos actualizando con las acciones del cliente.

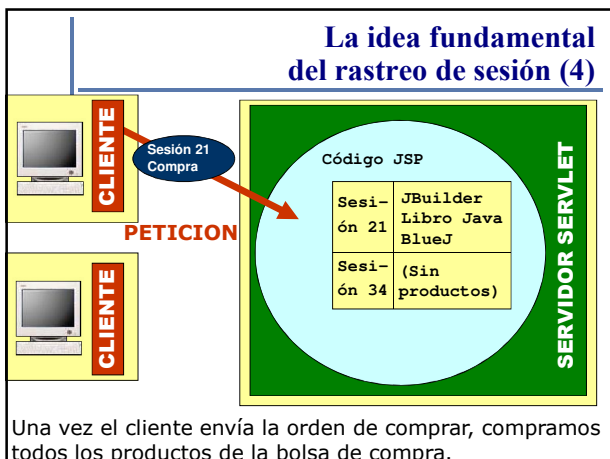
### La idea fundamental del rastreo de sesión (2)



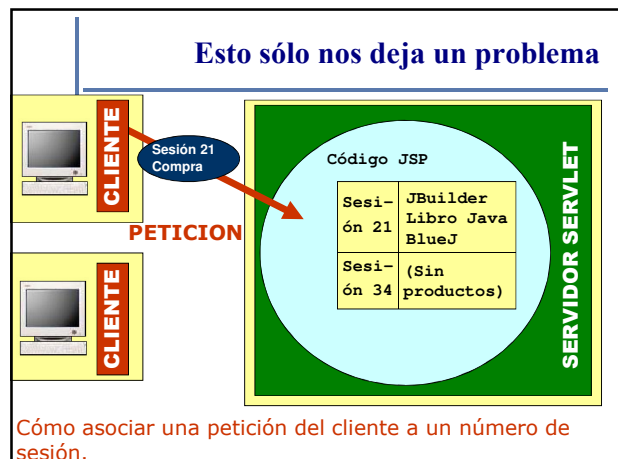
### La idea fundamental del rastreo de sesión (3)



### La idea fundamental del rastreo de sesión (4)



### Esto sólo nos deja un problema





### Cómo asociar una petición del cliente a un núm. de sesión

#### Opción descartada. Con números IP.

La idea es sencilla.

- Se obtiene el número IP de la máquina de donde proviene la petición.
- Cada número IP se asocia con un número de sesión diferente.
- Pero tenemos un grave problema.
- El número IP que nos llega puede no corresponder con el IP de la máquina del cliente. Puede haber una máquina proxy o bien el número IP del cliente puede cambiar si se conecta con acceso conmutado. Por ello, opción descartada.

### Cómo asociar una petición del cliente a un núm. de sesión

#### 1. Con cookies.

- Las cookies son una excelente opción para implementar el rastreo de sesión.
- En el ejemplo, deberíamos tener una cookie con el identificador de sesión (que caducará cuando pase mucho tiempo)-
- Esto sería una excelente solución, si no fuera por un grave inconveniente: **el usuario puede desactivar las cookies.**

### Cómo asociar una petición del cliente a un núm. de sesión

#### 2. Reescritura de URL.

- Añadir al final de la URL, datos adicionales que indican el número de sesión.
- Así se pueden generar URLs como

**http://servidor/pagina.html;numsesion=21**

- El problema es que tenemos que añadir estos datos adicionales a cada URL que accede el cliente: esto puede ser muy tedioso.
- Si el cliente deja la página Web y accede de nuevo al servidor con un enlace o marcador, **la información de sesión se pierde.**

### Cómo asociar una petición del cliente a un núm. de sesión

#### 3. Campos ocultos en formulario.

- Pueden añadirse campos ocultos en un formulario. Estos campos no aparecen por pantalla pero contienen como valor el número de la sesión. (`<INPUT TYPE="HIDDEN">`)
- El problema es que esto nos obliga a generar todas las páginas dinámicamente y no podemos guardar las páginas estáticas con HTML, lo que es incómodo.

### Otro inconveniente

- Independientemente de la aproximación que adoptemos, hay otro inconveniente.
- Hay que realizar muchas tareas tediosas:
  - Obtener un número único para cada sesión.
  - Asociar la tabla con cada petición.
  - Obtener el número de cada petición, aislándolo del resto de la cookie, de la URL o del formulario
- Pero estas tareas son las mismas para todas las aplicaciones Web. No tiene sentido repetirlas una y otra vez. Deberían estar ya programadas para que nos limitemos a utilizarlas.

### La buena noticia es

- Estas tareas ya están programadas.
- Hay un conjunto de clases y métodos (una API) que nos permite implementar rastreo de sesión de forma sencilla y eficiente.
- **Es la API de rastreo de sesión**, la cual es implementada por medio de cookies o reescritura de URL (si el usuario ha desactivado las cookies).
- Pero esto no debe preocuparnos, porque no tenemos que saber cómo está implementada.



### Todas las JSP tienen una sesión definida por defecto

- La podemos recuperar con la variable **session** que está definida por defecto en la JSP.
- Si no necesitamos mirar la sesión en esa JSP, es mejor desactivarla con

```
<@page session = "false">
```

### API de rastreo de sesión

- Nos permite llevar a cabo las tareas más comunes de manejo de sesiones de forma sencilla:
- 1. Añadir y recuperar datos a una sesión.
- 2. Cambiar y recuperar las propiedades de una sesión.
- 3. Terminar una sesión.

### 1. Añadir y recuperar datos a una sesión

- Cada sesión puede guardar varios datos asociados a ella. Cada dato recibe el nombre de atributo.
- Un atributo es de tipo **Object** y, por ello, puede guardarse cualquier cosa en él, aunque, al recuperarlo, se deberá hacer casting para reconvertirlo en el tipo específico.
- Sólo los tipos primitivos no pueden guardarse directamente, pero se pueden convertir en sus correspondientes clases envoltorio para guardarlos sin problemas.

### 1. Añadir y recuperar datos a una sesión

- Para recuperar un atributo:  

```
session.getAttribute (nombre) ;
```

 (El resultado es de tipo **Object** por lo que se hace casting) Si no existe el atributo, devuelve **null**
- Para añadir un atributo:  

```
session.setAttribute (nombre, valor) ;
```

 (Si ese atributo existía, su valor se reemplaza).
- Para borrar un atributo:  

```
session.removeAttribute (nombre) ;
```
- En estos ejemplos, nombre es de tipo **String** y **valor** es de tipo **Objeto**.

### 2. Cambiar y recuperar las propiedades de una sesión

- La propiedad más importante es la que indica cuanto tiene que pasar sin acceder a la sesión para que ésta se invalide. Se fija así:

```
session.setMaxInactiveInterval (segs) ;
```

Si el número es negativo, la sesión nunca expira.

```
session.getMaxInactiveInterval () ;
```

Recupera este número de segundos

### 3. Terminar una sesión

- La sesión termina automáticamente cuando la cantidad de tiempo que ha pasado sin accederla supera el intervalo especificado por **setMaxInactiveInterval**
- Si queremos terminar una sesión de forma explícita podemos usar

```
session.invalidate () ;
```



### Ejemplo

- Escribir una JSP que cuente cuantas veces ha accedido el usuario a la página usando rastreo de sesión

### Solución

```
<%int accesos;
Integer numAccesos =
(Integer)session.getAttribute("numAccesos");
if (numAccesos == null){//Si no defin. atrib.
 accesos = 0;
 session.setMaxInactiveInterval(1000);
} else {
 accesos = numAccesos.intValue();
}
session.setAttribute("numAccesos",
 new Integer(accesos+1));
}%>
<HTML><HEAD></HEAD><BODY> Has accedido
<%=accesos%> veces</BODY></HTML>
```

### Algunas notas sobre esta solución

- Como el atributo es entero, hemos tenido que transformarlo en la clase envoltorio **Integer** para poder guardarlo en la sesión.
- Fijense que **setMaxInactiveInterval** sólo se fija una vez y no necesita ser actualizado.

### Ejercicio

- Programar un servlet que, usando rastreo de sesión, guarde una lista de todas las veces que el cliente ha accedido a la página Web.
- Cada vez que el usuario acceda a la página Web, debe presentarle una lista de todas las veces anteriores (con número, fecha y hora).

### Ejercicio

- Complementar el ejercicio anterior para que la página Web no sólo incluya una lista con los últimos accesos del usuario sino que también indique el nombre y apellidos del mismo.
- Si la página Web no conoce el nombre y apellidos del usuario (pues es la primera vez que éste accede) debería solicitarlos.
- Implementarlo todo con la API de rastreo de sesión.

### Diferencias entre navegadores

- La forma de manejar las sesiones se basa en la forma de manejar las cookies y ésta varía para cada navegador.
- Así, normalmente, en Firefox, Mozilla y Netscape, si se abren varias ventanas del navegador en una sola máquina, todas comparten la misma sesión. También las pestañas.
- En cambio con Internet Explorer, las ventanas del navegador tienen cada una su propia sesión (excepto si se han abierto con **Archivo| Nuevo| Ventana** o **Ctrl+U**).
- Esto permite a dos usuarios de IE conectarse a un mismo sitio desde la misma máquina y ser tratados de forma diferente. No funciona para algunas combinaciones de IE+SO.



### Programa del curso

- 1. Objetivos y metodología del curso.
- 2. Repaso del curso anterior.
- 3. Ampliación del lenguaje Java.
- 4. Cookies y rastreo de sesión.
- **5. Otros aspectos.**
- 6. Un ejemplo práctico.

### Multithreading

- Receta para no tener errores.
- Las JSP no deben tener atributos (difícil, porque no lo hemos visto).
- Las clases de negocio y datos tampoco.
- Toda la información sobre el estado del cliente la guardaremos en la sesión (o en las cookies).

### Lazy initialization

- Supongamos que la clase Cliente tiene un atributo libretas que es una colección de objetos de clase Libreta. Si no hacemos lazy initialization (como ahora, que habíamos puesto lazy="false"), cada vez que recuperemos una libreta recuperaremos también sus libretas de la base de datos.
- Esto es ineficiente porque quizás las libretas nunca las miramos (o miramos sólo una cuando podrían ser mil).

### Lazy initialization

- Hibernate recomienda que siempre se use lazy initialization.
- Cuando hay lazy initialization, las libretas sólo se recuperan cuando se accede a ellas a través de algo como cliente.getLibretas().get(0).
- Esto es más eficiente y totalmente transparente al usuario pero tenemos que programar de otra forma.

### Debemos tener esta clase en la capa de datos

```
public class DAOTransaccion {
 private SessionFactory sessionFactory = new
 Configuration().configure().buildSessionFactory();
 private ThreadLocal sessionDelThread = new ThreadLocal();
 Session getSession(){
 Session session = (Session)sessionDelThread.get();
 if (session == null){
 session = this.sessionFactory.openSession();
 sessionDelThread.set(session);
 }
 return session;
 }
 public void finalizaAccesoDatos(){
 Session session = (Session)sessionDelThread.get();
 if (session != null){
 session.close();
 }
 sessionDelThread.set(null);
 }
}
```

### Debemos tener esta clase en la capa de negocio

```
public class NgcAcceso {
 public void finalizaAcceso() {
 DAOTransaccion daoTransaccion =
 FactoryDatos.getInstancia().getDAOTransaccion();
 daoTransaccion.finalizaAccesoDatos();
 }
}
```



### Toda JSP que acceda a datos de la base de datos deberán finalizar con

```
FactoryNegocio.getInstance().getNgcAcceso().finalizaAcceso();
```



### Para hacer lazy initialization

- 1. Quitar lazy="false" de todos los archivos de correspondencia de Hibernate.
- 2. Colocar en las capas de negocio y de datos NgcAcceso y DAOTransaccion respectivamente.
- 3. Todas las clases persistentes deben tener un constructor sin parámetros **public o package**.
- 4. Cada JSP que acceda a datos debe acabar con `FactoryNegocio.getInstance().getNgcAcceso().finalizaAcceso();`
- 5. Cada clase DAO no debe cerrar la sesión y debe obtener la sesión usando el método **getSession** de **DAOTransaccion**



### Los temas más importantes que nos hemos dejado en el tintero

- Multithreading.
- Arquitectura MVC.
- Transacciones y control de concurrencia en Hibernate.
- El lenguaje HQL.



### Programa del curso

- 1. Objetivos y metodología del curso.
- 2. Repaso del curso anterior.
- 3. Ampliación del lenguaje Java.
- 4. Cookies y rastreo de sesión.
- 5. Otros aspectos.
- 6. Un ejemplo práctico.

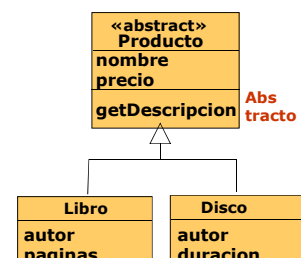


### Ejemplo

- Una tienda de comercio electrónico llamada Amazonito, que vende libros y discos.
- Podemos hacer las siguientes funciones:
  - Buscar productos y añadirlos a la cesta de la compra.
  - Consultar la bolsa de la compra y hacer el pedido.
  - Consultar los pedidos realizados.
- Por simplicidad, nos hemos dejado tres funciones.
  - Añadir discos.
  - Añadir libros.
  - Añadir clientes.



### Comencemos por la capa de dominio

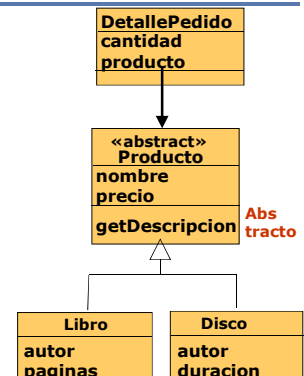




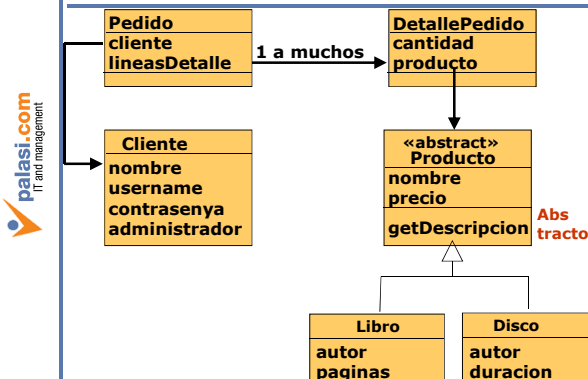
### ¿Por qué usamos herencia?

- Queremos polimorfismo (tratar a libro y disco de la misma forma, como producto).
- Queremos reutilizar código de la superclase. Seguramente, queremos tener código en la superclase que hereden las subclases. Por ejemplo, cálculos de impuestos sobre los productos.
- Si sólo quisiéramos polimorfismo, elegiríamos interfaces.

### Comencemos por la capa de dominio

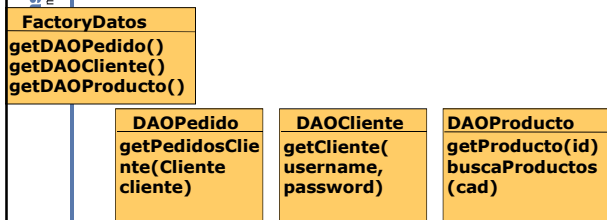


### Comencemos por la capa de dominio



### Comencemos por la capa de datos

- Versión simplificada pues falta DAOTransacción, que es como habíamos dicho antes.



### La clase DAOCliente de la capa de datos

```

public class DAOCliente {
 public Cliente getCliente(String us, String pas){
 Session sesion = FactoryDatos.getInstancia().
 getDAOTransaccion().getSession();
 List clientes = sesion.createQuery("from
 Cliente as cliente where cliente.username
 = '"+us+"'").list();
 if (clientes.isEmpty()){
 return null;
 } else {
 Cliente clien = (Cliente)clientes.get(0);
 if (!clien.getContrasena().equals(pas)){
 return null;
 } else {
 return clien;
 }
 }
 }
}

```

### La clase DAOPedido de la capa de datos

```

public class DAOPedido {
 public void agregaPedido (Pedido pedido){
 Session sesion = //Aquí lo de siempre
 sesion.save(pedido);
 sesion.flush();
 }
 public Collection getPedidosCliente(Cliente
 cliente){
 Session sesion = // Aquí lo de siempre
 List pedidos = sesion.createQuery
 ("from Pedido as pedido where pedido.cliente
 =:cliente").setEntity("cliente", cliente).
 list();
 return pedidos;
 }
}

```



### La query es así

```
session.createQuery ("from Pedido as pedido
where pedido.cliente =:cliente").
setEntity("cliente", cliente).list();
```

```
"from Pedido as pedido where
pedido.cliente =:cliente"
```

- Antes hubiéramos hecho algo como así.

```
"from Pedido as pedido where
pedido.cliente =" + cliente
```

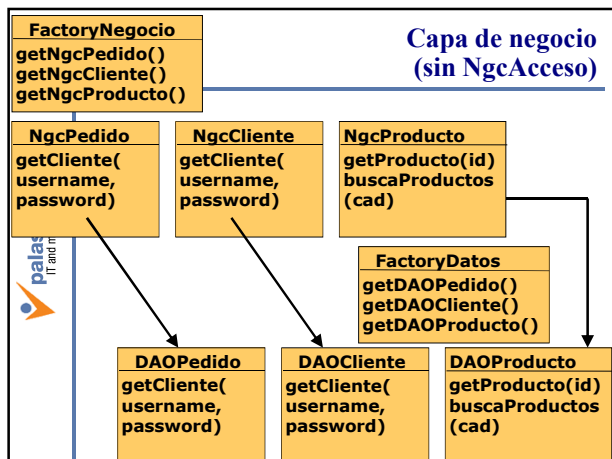
- El problema es que cliente es un objeto y no un String. ¿Cómo podemos añadirlo a la String de una query?

### La query es así

```
session.createQuery ("from Pedido as pedido
where pedido.cliente =:cliente").
setEntity("cliente", cliente).list();
```

```
"from Pedido as pedido where
pedido.cliente =:cliente"
```

- **:cliente** es un parámetro con nombre. Es una query que puede recibir un parámetro (de tipo objeto).
- El parámetro se añade después con **setEntity**. Así podemos añadir los parámetros que queramos.



### Un ejemplo de clase de negocio

```
public class NgcProducto {
 public Collection buscaProductos (String cadena){
 DAOProducto daoProducto =
 FactoryDatos.getInstance().getDAOProducto();
 return daoProducto.buscaProductos(cadena);
 }
}
```

- Sólo se obtiene la clase DAO con el FactoryDatos y después se llama a su método, devolviéndolo en seguida.

### La capa de negocio no tiene ningún misterio

- Sólo llama a la capa de datos y devuelve los resultados.
- Esto es porque en este caso no se hacen procesos, que es lo que sale la capa de negocio, sino que simplemente se accede a la base de datos.
- Los métodos para añadir un libro y un disco serán más interesantes pero no los veremos aquí.

### Todas las clases de negocio tienen la misma estructura de dos partes

```
public class NgcProducto {
 public Collection buscaProductos (String cadena){
 DAOProducto daoProducto =
 FactoryDatos.getInstance().getDAOProducto();
 return daoProducto.buscaProductos(cadena);
 }
}
```

- Consiguen el objeto de la capa inferior (datos) por la factory.
- Después hacen el trabajo.

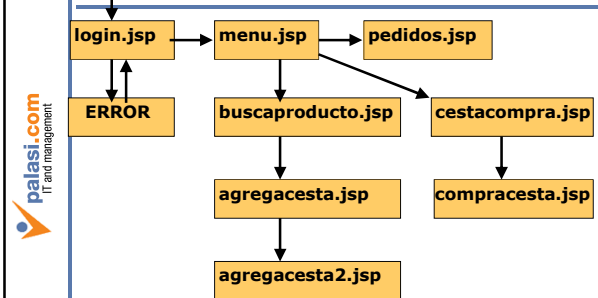


### Las JSP tendrán la misma estructura

```
public class NgcProducto {
 public Collection buscaProductos (String cadena){
 DAOProducto daoProducto =
 FactoryDatos.getInstancia().getDAOProducto();
 return daoProducto.buscaProductos(cadena);
 }
}
```

- Consiguen el objeto de la capa inferior (negocio) por la factory.
- Después hacen el trabajo.

### Veamos la capa de presentación (simplificada sin HTML)



### login.jsp

```
FactoryNegocio factoryNegocio =
 FactoryNegocio.getInstancia();
NgcCliente ngcCliente =
 factoryNegocio.getNgcCliente();

String username =
 request.getParameter("username");
String password =
 request.getParameter("password");
Cliente usuario =
 ngcCliente.getCliente(username, password);
if (usuario == null) {%>
 Usuario no autorizado

<%> else {
 session.setAttribute("usuario", usuario);
 <jsp:forward page = "menu.jsp"/>
<%>
```

### login.jsp (simplificado). Quitando la primera parte de la estructura

```
String user =
 request.getParameter("username");
String password =
 request.getParameter("password");
Cliente usuario =
 ngcCliente.getCliente(user, password);
if (usuario == null) {%>
 Usuario no autorizado

<%> else {
 session.setAttribute("usuario", usuario);
 <jsp:forward page = "menu.jsp"/>
<%>
```

### cestacompra.jsp

```
<%Pedido cesta =
 (Pedido)session.getAttribute("cesta");
%>
<HTML><HEAD><TITLE>Tienda</TITLE></HEAD><BODY>
<%DetallePedido linea;
for (Iterator iterador =cesta.getLineasDetalle().
 iterator(); iterador.hasNext();){
 linea = (DetallePedido)iterador.next();
 out.print(linea.getCantidad()+"unidades"+
 linea.getProducto().getDescripcion());
}
%></BODY></HTML>
```

### compracesta.jsp

```
<%
Pedido cesta =
 (Pedido)session.getAttribute("cesta");
ngcPedido.agregaPedido(cesta);
session.removeAttribute("cesta");
%>
```



### pedidos.jsp

```
<HTML><HEAD><TITLE>Tienda</TITLE></HEAD><BODY><%
Cliente usuario =
 (Cliente)session.getAttribute("usuario");
Collection pedidos =
 ngcPedido.getPedidosCliente(usuario);
Pedido pedido; DetallePedido linea;
for (Iterator itPedido = pedidos.iterator();
 itPedido.hasNext();){
 pedido = (Pedido) itPedido.next();
 out.print("PEDIDO NUM."+pedido.getId());
 for (Iterator itL=pedido.getLineasDetalle().
 iterator(); itL.hasNext();){
 linea = (DetallePedido) itL.next();
 out.print(linea.getCantidad()+" unidades de
 "+linea.getProducto().getDescripcion());
 }
} } %></BODY></HTML>
```

### agregacesta2.jsp

```
<%
int unidades =
 Integer.parseInt(request.getParameter("unidades")
);
Producto producto =
 (Producto)session.getAttribute("ultimoproducto");

Pedido cesta =
 (Pedido)session.getAttribute("cesta");
if (cesta == null){
 Cliente usuario =
 (Cliente)session.getAttribute("usuario");
 cesta = new Pedido(usuario);
}

DetallePedido lineaCesta = new
 DetallePedido(unidades,producto);
cesta.agregaDetalle(lineaCesta);
session.setAttribute("cesta", cesta);
%>
```

### La flexibilidad de la programación O-O

- Fijense que todo el programa trata productos. No trata las clases concretas libro y disco
- Excepto cuando las creamos en la jsp de libro y disco lo que será sólo una línea.
- Esto hace muy fácil añadir nuevas líneas de producto.
- Para añadir altavoz:
  - Programar la clase.
  - Hacer una jsp que recoja los datos.
  - Retocar Producto.hbm.xml para poner una nueva <subclass
  - Añadir los campos específicos a la tabla producto.

### La flexibilidad de la programación O-O

- Si queremos añadir ISBN a libro: nuevo atributo, jsp añade libro y todo lo otro seguirá igual.
- Si queremos que no se repitan productos en los pedidos y cesta de la compra, sólo en agregar línea de detalle.
- Si queremos cambiar a escritorio, sólo deberemos cambiar a Swing o SWT la capa de presentación.
- Si queremos cambiar de base de datos, sólo deberemos cambiar dos líneas en los archivos de configuración de Hibernate.
- Si queremos cambiar de motor de persistencia sólo deberemos cambiar la capa de datos.

### Sobre el profesor

Dr. Vicent-Ramon Palasí Lallana.

- Consultas y dudas a:
  - Teléfono: 275-4254.
  - E-mail: [vpalasi@aurumsol.com](mailto:vpalasi@aurumsol.com)
  - Web: [www.aurumsol.com](http://www.aurumsol.com)

**Vicent Palasí, PhD, MBA, MEd. Todos los derechos reservados.**

**Mail: [palasi@palasi.com](mailto:palasi@palasi.com) Web: [www.palasi.com](http://www.palasi.com)**