palasi.com
IT and management

Programación para el Web con Java Tema 2

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

Programa del curso

- Objetivos y metodología del curso.
- Tema 1. Introducción a la programación Web en Java.
- Tema 2. Fundamentos de la programación orientada a objetos.
- Tema 3. Motores de persistencia.

Tema 2. Fundamentos de la programación orientada a objetos

Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

La crisis del software

- Es el nombre que damos al estado insatisfactorio en que se encuentra el desarrollo de software.
- Las tareas que nos gustaría resolver mediante las computadoras son en la práctica:
  - demasiado difíciles de resolver.
  - suelen extenderse más allá del costo y el tiempo previsto.
  - es muy probable que contengan errores.

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

La crisis del software

- Los proyectos de programación de gran tamaño consumen 150% del tiempo previsto.
- El 25% de estos proyectos son cancelados.
- El 75% de los proyectos no cancelados:
  - O bien no funcionan como se quería.
  - O bien no se utilizan para nada.

Estadísticas sobre proyectos de software

Tamaño	Temprano	A tiempo	Retraso	Cancelados
1 PF	14.68%	83.16%	1.92%	0.25%
10 PF	11.08%	81.25%	5.67%	2.00%
100 PF	6.06%	74.77%	11.83%	7.33%
1000 PF	1.24%	60.76%	17.67%	20.33%
10000 PF	0.14%	28.03%	23.83%	48.00%
100,000PF	0.00 %	13.67%	21.33%	65.00%

- Fuente: Patterns of Software Systems Failure And Success, Capers Jones.

Soluciones a la crisis del software

- La crisis se detectó en la conferencia de la OTAN de 1968 sobre Ingeniería del Software y sigue vigente.
- Se han investigado varias soluciones:
  - Derivación y verificación automática de software (de interés más teórico).
  - Programación con componentes reusables de software.

Programación con componentes reusables de software

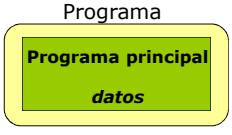
- Una de las técnicas para mitigar los problemas del desarrollo de software.
- Se basa en la idea extendida en otras ingenierías que, para construir un producto, sólo deben ensamblarse piezas preconstruidas.
- La idea básica es:
  - existen componentes reusables
  - programar una aplicación consiste en ensamblarlos.

Programación con componentes reusables de software

- Objetivo perseguido durante muchos años. El Santo Grial del Desarrollo de Software.
- Para conseguirlo se han desarrollado varias técnicas de programación:
  - programación no estructurada.
  - programación procedimental.
  - programación modular.
  - programación orientada a objetos.

Programación no estructurada

- Es el primer tipo de programación que apareció.
- Hay un único programa principal que trabaja con unos datos globales.



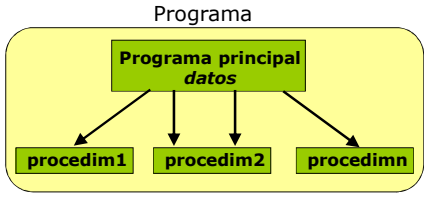
Mail: palasi@palasi.com Web: www.palasi.com

Características de la programación no estructurada

- Hace muy difícil la programación a gran escala:
  - no hay ninguna reusabilidad del código.
  - si hay que realizar una tarea dos veces, se debe volver a escribir el código.
- Prácticamente no se utiliza en el mundo real.

Programación procedimental

- El código de tareas que se repiten se escribe en procedimientos ("subprogramas"). El programa principal los llama cada vez que debe hacer dichas tareas.



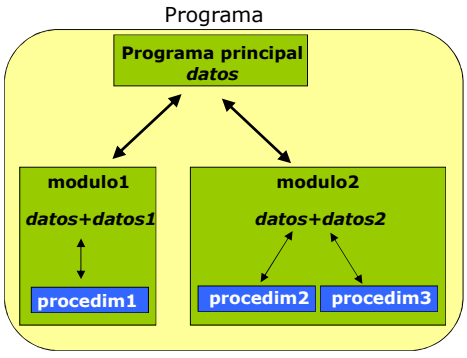
Características de la programación procedimental

- Reusabilidad a nivel elemental:
  - se reusa el código de los procedimientos
  - se evita la tarea de repetir código dentro de un programa
- El problema es que los procedimientos no pueden ser reusados por otros programas.

Programación modular

- Los procedimientos referentes a un mismo conjunto de datos se agrupan en un módulo (compilado separadamente).
- Cada programa consta de varios módulos. El programa principal coordina las llamadas a los procedimientos de los otros módulos.

Programación modular



Características de la programación modular

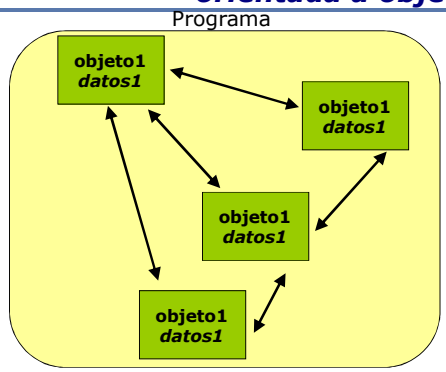
- Hay una reusabilidad de los módulos.
- De hecho, los módulos tienen sus propios datos y se convierten en parecidos a un tipo de datos con sus operaciones (ejemplo: lista).
- Pero hay inconvenientes:
  - los módulos no se acaban de comportar como un tipo de datos.
  - se vuelve complicado tener y gestionar varias instancias del módulo (varias listas).

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

Programación orientada a objetos

- Resuelve los problemas mencionados.
- Desaparece el concepto de programa principal.
- Lo que hay es una red de ***objetos*** interactuantes.
- Los objetos
  - contienen datos y operaciones sobre estos datos.
  - interactúan intercambiando mensajes (análogo a llamadas a procedimiento).

Programación orientada a objetos



Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

Programación orientada a objetos

- Es una de las técnicas para implementar programación con componentes reusables de software.
- Estándar dominante actualmente de programación.
- Evoluciona de otras formas anteriores de programación, pero introduce una manera diferente de pensar (más parecida a la vida real).
- Se basa en el concepto de objeto.

Objeto (de software)

- Concepto fundamental de la programación orientada a objetos.
- Es análogo a un objeto de la vida real.

Un objeto de software es análogo a un objeto de la vida real

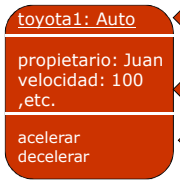
- Por ejemplo, un objeto real es el auto que es propiedad del licenciado.
- De un objeto real, nos interesan dos cosas.

Las dos cosas que nos interesan de un objeto

- 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.



Nombre del objeto y clase subrayados (ya veremos que es clase)

Atributos

Métodos

Este objeto de software modela el carro anterior

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**

Una clase es una descripción de las características de los objetos

- La descripción de las características de estos objetos similares se llama **clase**
- A partir de estas características podemos crear objetos. Por eso, una clase puede verse cómo las instrucciones para fabricar objetos.

Un ejemplo

- La clase "Toyota Corola" indica cómo serán los objetos (autos) que se crearán con ese modelo
- La clase sería la descripción que describe cómo es un "Toyota Corola"
- Un objeto sería el auto "Toy.Corola" de Juan

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

Resumen: Tres formas de ver lo mismo

- Una clase es el conjunto de todos los objetos del mismo tipo.
- Una clase es una descripción de las características de objetos similares.
- Una clase son las instrucciones para crear objetos similares.

Un poco de terminología

- Cuando un objeto pertenece a una clase, se dice que es una **instancia** de esta clase.
- Así, los objetos autoDePerez, autoDeBonilla, autoDeAlvarado son instancias de la clase Auto.
- Decimos que instanciamos una clase cuando creamos un objeto de la misma.

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.

Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

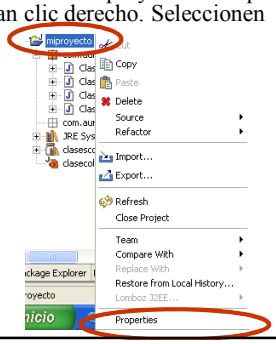
Mail: palasi@palasi.com Web: www.palasi.com

Vamos a ver el ciclo de vida de un objeto

- Para ello, nos vamos a ayudar de unas clases previamente construidas que nos ayudarán a aprender los conceptos.
- Estas clases se encuentran en un archivo llamado **`autos.jar`**.
- Copien esa carpeta a la carpeta **`webapps\ROOT\WEB-INF\lib`** de su instalación de Tomcat.
- Reinicien Tomcat.

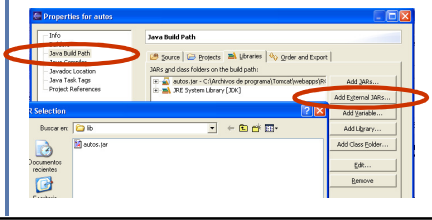
Además, configuren Eclipse de la siguiente manera

- Seleccionen el nombre del proyecto en la parte izquierda y hagan clic derecho. Seleccionen Properties.



Además, configuren Eclipse de la siguiente manera

- Seleccionen Java Build Path|Libraries| Add External JARs. Después vayan al directorio **`webapps\ROOT\WEB-INF\lib`** y seleccionen **`autos.jar`**



Importante

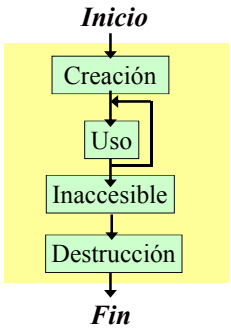
- Para usar las clases, tendremos que poner como primera línea de nuestras JSP la siguiente línea.

```
<%@ page
  import="com.aurumsol.autos.*" %>
```

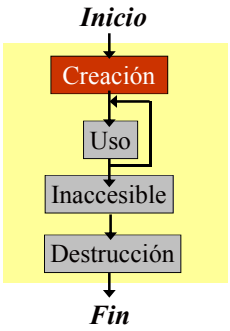
(todo en una sola línea).

- Cuidado de no dejar espacios alrededor de =
- Por ahora no vamos a explicar con detalle esta línea. Sólo la usaremos.

Ciclo de vida de un objeto



Ciclo de vida de un objeto



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

Creación de un objeto

- Para crear un objeto, debe indicarse a partir de qué clase se creará el objeto.
- Se crea el objeto siguiendo esa descripción que es la clase.
- Veamos cómo se hace en Java

¿Cómo se codifica esto?

- Se necesitan dos pasos: definición de una variable y creación del objeto.
- 1. Se declara una variable de la clase que queremos instanciar (definición). Fíjense que, como hemos dicho, la clase es un tipo de datos.

```
Auto objAuto1;
```

- 2.- Se crea un nuevo objeto y su referencia se almacena en esta variable.

```
objAuto1 = new Auto();
```

1. Definición

- Necesitamos declarar una variable para guardar el objeto que creemos.
- En los lenguajes convencionales, hay que declarar una variable para guardar el valor de un tipo de datos.
- Como hemos dicho que la clase es un tipo de datos y un objeto es el valor de ese tipo, necesitaremos una variable para guardarlo. Esta variable será del tipo de datos de la clase.

```
Clase var;
```

- En nuestro caso:

```
Auto objAuto1;
```

2. Creación

- Una vez declarada la variable, la creación consta de dos tareas.
  - Crear el objeto.
  - Asignarlo a la variable.
- Esto se realiza como una única instrucción.

```
var = new Clase();
```

  - **new** ***Clase()*** crea un objeto nuevo de la clase (tipo) ***Clase*** (no olviden los paréntesis).
  - La asignación, ***var*** =, asigna este nuevo objeto a la variable.
- En nuestro caso,

```
objAuto1 = new Auto();
```

Por supuesto, todo esto se puede hacer en una sola instrucción

```
Auto objAuto1 = new Auto();
```

- **Pero hay que tener en cuenta que conceptualmente, son dos conceptos diferentes.**
- 1. Declarar una variable de la clase que queremos instanciar (definición).

```
Auto objAuto1;
```

- 2.- Crear un nuevo objeto y su referencia se almacena en esta variable.

```
objAuto1 = new Auto();
```

Nota

- Fíjense que las clases en Java suelen tener la primera letra en mayúscula (en vez de las variables que la tienen en minúscula).
- Esto no es obligatorio para la compilación pero es una convención universalmente aceptada y así lo tienen que hacer.
- Si hay más palabras, cada una empieza por mayúscula. Así:

```
Auto
ClaseNueva
TarjetaDeCredito
```

Ejercicio

- Creen una aplicación Web que contenga un botón con el título “Crear auto”.
- Cuando se haga clic sobre ese botón, debe crearse un objeto de la clase “**Auto**”.
- Despliegen la aplicación Web y hagan clic varias veces sobre el botón.

Solución (sólo ponemos la JSP, el formulario HTML es sencillo)

```
<%@ page
  import="com.aurumsol.autos.*"%>
<%Auto objAuto1 = new Auto();%>
<HTML>
<HEAD>
<TITLE>Crea objeto</TITLE>
</HEAD>
<BODY>
<H2>Se cre&oacute; un nuevo objeto de
  la clase Auto.</H2>
</BODY>
</HTML>
```

Aparece un letrero

- Sí, eso está muy bien, pero ¿dónde esta el objeto?.

Los objetos son invisibles

- Se encuentran en la memoria de la computadora pero no los podemos ver.
- Observaremos sus efectos en pantalla pero los objetos normalmente no los veremos.
- Hay que hacer un acto de fe (y de imaginación). Pero esto no es nuevo de la programación orientada a objetos: también pasaba, por ejemplo, con las variables de los lenguajes tradicionales.

Sin embargo, podemos obtener información sobre los objetos

- Por ahora, para obtener información sobre los objetos de clase Auto, utilizaremos la siguiente expresión JSP

```
<%= obj.informacion() %>
```

- Donde ***obj*** es el nombre del objeto del cual queremos obtener la información
- Por ahora, no explicaremos esta expresión: sólo la usaremos.

Por ejemplo, despleguemos esta JSP y hagamos clic varias veces en el botón

```
<%@ page
  import="com.aurumsol.autos.*"%>
<%Auto objAuto1 = new Auto();%>
<HTML>
<HEAD>
<TITLE>Crea objeto</TITLE>
</HEAD>
<BODY>
<H2>Se cre&oacute; un nuevo objeto de
  la clase Auto, con la siguiente
  informaci&oacute;n:<BR></H2>
<%= objAuto1.informacion() %>
</BODY>
</HTML>
```

Cada vez que se pulsa en el botón se crea un objeto nuevo

- Aparece la información de cada uno de los objetos que se crean.

Se creó un nuevo objeto de la clase Auto, con la siguiente información:

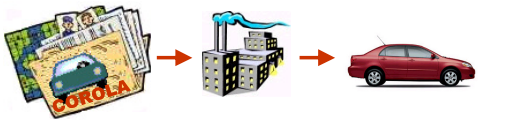
Auto número 4.
Propietario: Desconocido.
Velocidad: 0

Qué hemos hecho

- La clase **`Auto`** era una descripción para crear objetos de esa clase. Lo que se ha hecho es crear un objeto a partir de la clase.

`new Auto()`

CLASE Auto → **OBJETO objAuto1**

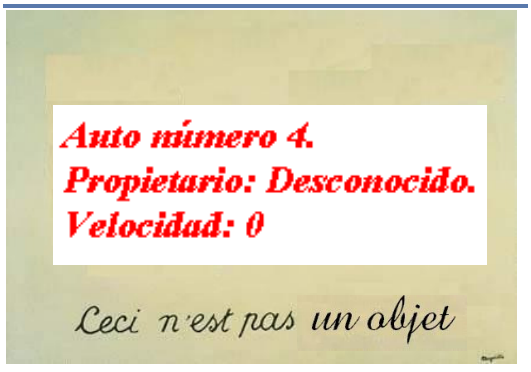


La información que aparece en pantalla es la representación del objeto

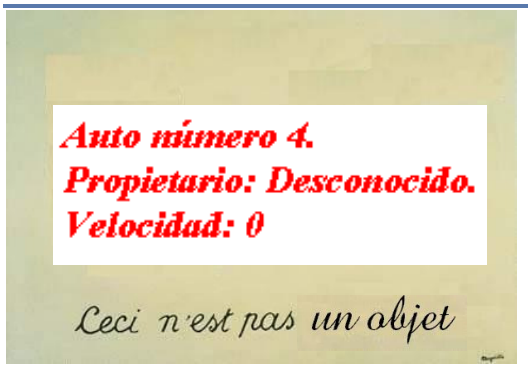
- Hemos programado que cada vez que se hace `<%= obj.informacion() %>` aparezca la información del objeto en la pantalla.
- Esto no es así por defecto, sino que lo hemos programado así para que entiendan bien el concepto de objeto.
- Por ahora no piensen cómo está programado, sólo piensen que lo que aparece por pantalla es una representación de objeto, **pero no es el objeto mismo.**

Esto no es una pipa (Magritte)

Esto no es un objeto



Es la representación de un objeto



El texto de información es la representación de un objeto

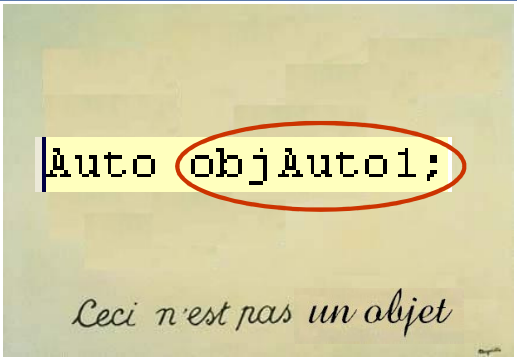
- Igual que un icono es la representación de un archivo, pero no el archivo mismo.

Atención

```
Auto objAuto1 = new Auto();
```

- En la variable **objAuto1** no se almacena el objeto sino una **referencia** al objeto.
- La referencia representa al objeto y permite que lo manipulemos pero NO es el objeto.
- Es similar al icono de un archivo, que nos permite manipular el archivo pero no es el archivo.

Es decir, pasa lo mismo que antes

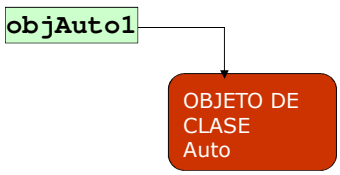


La variable no es un objeto

- La variable es una representación, un icono del objeto.
- Nos permite manipular y utilizar un objeto, pero no es el objeto mismo. Se puede ver como el mango que agarra un objeto.
- Se dice que la variable es una **referencia** del objeto. O bien, que la variable **apunta** al objeto.
- En términos de lenguaje máquina, sería la dirección de memoria donde está guardado el objeto. En un lenguaje como C, sería un puntero.

Referencia al objeto

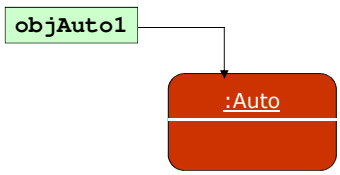
```
Auto objAuto1 = new Auto();
```



- La variable es una referencia. Permite manipular el objeto, pero no es el objeto mismo.

Para abreviar

```
Auto objAuto1 = new Auto();
```

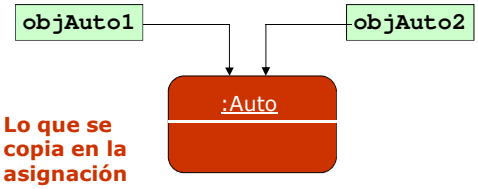


- :Auto quiere decir en UML, un objeto de la clase "Auto"

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

Varias referencias pueden referenciar el mismo objeto

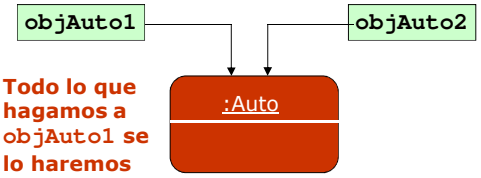
```
Auto objAuto1 = new Auto();
Auto objAuto2 = objAuto1;
```



Lo que se copia en la asignación no es el objeto sino la referencia

No se han creado dos objetos sino dos referencias al mismo objeto

```
Auto objAuto1 = new Auto();
Auto objAuto2 = objAuto1;
```



Todo lo que hagamos a `objAuto1` se lo haremos también a `objAuto2`

Esto contrasta con los tipos primitivos

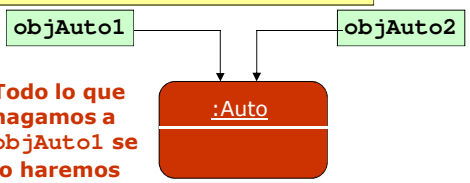
```
int entero1 = 5;
int entero2 = entero1;
```

entero1	entero2
5	5

En los tipos primitivos tenemos dos copias diferentes y cada una se manipula independientemente. La asignación copia valores, no referencias.

En objetos no son dos copias sino dos referencias a dos copias

```
Auto objAuto1 = new Auto();
Auto objAuto2 = objAuto1;
```



Todo lo que hagamos a `objAuto1` se lo haremos también a `objAuto2`

Ejercicio. Cambien su JSP para que sea como ésta.

```
<%@ page import="com.aurumsol.autos.*"%>
<%Auto objAuto1 = new Auto();
  Auto objAuto2 = objAuto1;%>
<HTML>
<HEAD>
<TITLE>Referencias</TITLE>
</HEAD>
<BODY>
<H2>Informacion de objAuto1:<BR></H2>
<%= objAuto1.informacion() %><BR>
<H2>Informacion de objAuto2:<BR></H2>
<%= objAuto2.informacion() %>
</BODY>
</HTML>
```

Pregunta

- ¿Qué es lo que obtienen al ejecutar la anterior JSP?

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

Respuesta

- Se obtiene el mismo objeto, no importa qué variable usemos.

Informacion de objAuto1:

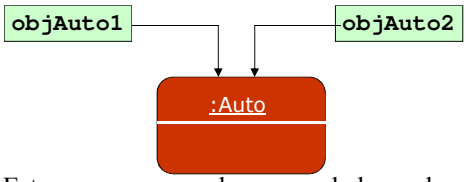
Auto número 5.
Propietario: Desconocido.
Velocidad: 0

Informacion de objAuto2:

Auto número 5.
Propietario: Desconocido.
Velocidad: 0

Las dos variables apuntan al mismo objeto

```
Auto objAuto1 = new Auto();
Auto objAuto2 = objAuto1;
```



- Esto nos convence de una verdad grande:
  las variables contienen referencias a los objetos pero no objetos.

Pregunta

- ¿Cómo haríamos que **`objAuto1`** y **`objAuto2`** referenciaran dos objetos diferentes?

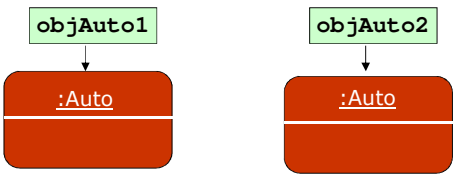
Respuesta

- Invocando dos veces la creación con **`new`**:

```
Auto objAuto1 = new Auto();
Auto objAuto2 = new Auto();
```

Respuesta

```
Auto objAuto1 = new Auto();
Auto objAuto2 = new Auto();
```



Ejercicio

- Comprueben esta solución, incluyéndola en la aplicación Web anterior.

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

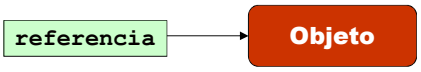
Importante

```
Auto objAuto1 = new Auto();
```

- Como sabemos, **objAuto1** no es el objeto: es una referencia al objeto.
- Sin embargo, muchas veces, para no hacerlo largo, diremos que es un objeto.
- Esto es un abuso de lenguaje. Lo importante es tener las cosas claras, digamos como las digamos.

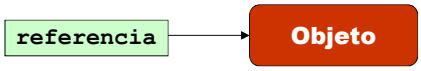
Resumiendo

- En Java hay dos cosas completamente diferentes:
- **Objetos**. Grupo de datos que el programa trata como una unidad.
- **Referencias a objetos (variables).** La forma con que accedemos a los objetos.



Tanto las referencias como los objetos tienen clase

- **Objetos**. La clase del objeto es la que se escribe detrás de la palabra **new**. Se le llama clase de creación.
- **Referencias a objetos (variables).** La clase de una variable es la clase con la cual se declara la variable.



1 variable y el objeto que refer. deben ser de la misma clase

```
Auto objAuto1;
objAuto1 = new Auto();
```

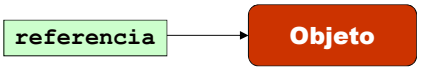
OK

```
referencia de clase Auto.
objeto de clase Auto.
```

```
Bicicleta obj1;
obj1 = new Auto();
```

NO

```
referencia de clase Bicicleta.
objeto de clase Auto.
```



Repasando

```
Auto objAuto1;
objAuto1 = new Auto();
```

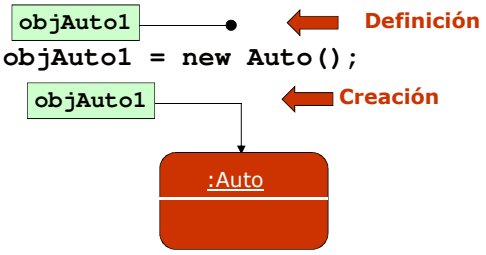
Definición

Creación

- A la creación se le llama a veces instanciación.
- Se dice que instanciamos la clase (ya que creamos una instancia –objeto- de ella).
- A veces, abusamos del lenguaje y decimos que instanciamos el objeto.

Definición y creación de un objeto

```
Auto objAuto1;
```



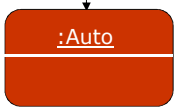
Mail: palasi@palasi.com Web: www.palasi.com

Pregunta: ¿Qué referencia objAuto1 después de definición?

```
Auto objAuto1;
objAuto1 = new Auto();
```

objAuto1 ← Definición

objAuto1 ← Creación



Respuesta: No referencia nada

```
Auto objAuto1;
objAuto1 = new Auto();
```

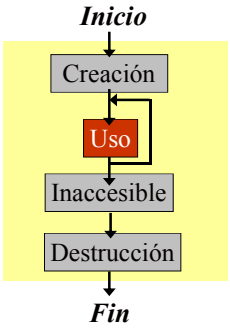
objAuto1 ← Definición

Se dice que referencia a `null`

Ejercicio

- Crear tres objetos de la clase **`Auto`** en la misma JSP y escribir su información.

Ciclo de vida de un objeto



Las dos cosas que nos interesan de un objeto (ejemplo, de clase Auto).

- Sus características o **atributos.**
  - Su velocidad.
  - Su propietario
- El tipo de acciones que pueden realizarse con él o **métodos**
  - Acelerar.
  - Decelerar.
  - Cambiar el propietario.

- A los atributos y métodos se les llama **miembros.**

Uso de un objeto

- Consistirá en dos tipos de modalidades de uso diferente:
- Acceder a los atributos de un objeto.
- Ejecutar los métodos del objeto.

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

Uso de un objeto

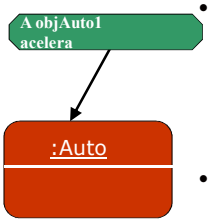
- Consistirá en dos tipos de modalidades de uso diferente:
- Acceder a los atributos de un objeto.
- Ejecutar los métodos del objeto.

Conocer los métodos de un objeto

- Más adelante, veremos maneras en Eclipse de saber cuáles son los métodos de un objeto.
- Por ahora, nos basta con saber que los objetos de clase **`Auto`** tienen los métodos **`acelera`**, **`decelera`**.

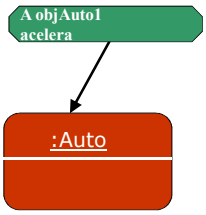
Cómo ejecutar los métodos

- Esta ejecución se llama **llamada a método o mensaje**.



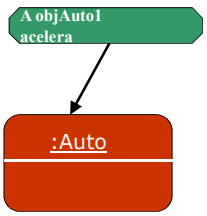
- Es una petición a un objeto para que realice un método (tarea) de las que sabe hacer este objeto.
- El objeto realiza la tarea siguiendo el método que tiene definido.

Método



- El objeto para realizar la tarea que ha pedido el mensaje tiene un método definido.
- Este método está oculto:
  - sólo lo conoce el objeto
  - no se conoce desde el exterior (excepto su nombre).

Desde el punto de vista de la programación



- Los métodos no son más que procedimientos, funciones o subrutinas.
- Por ello tienen
- Nombre.
- Parámetros.
- A veces, resultado.
- Una implementación o cuerpo.

Método o mensaje

- Hay que distinguir entre:
- Método (análogo a procedimiento)
- Mensaje o llamada a Método. (análogo a llamada a procedimiento)

- **Mensaje o llamada a método** es la petición (el clic) que hemos hecho para ejecutar **`acelera.`**
- **Método** es el procedimiento que está implementado internamente en el objeto y que se ejecuta, permitiendo al objeto "acelerar". **Por ahora, no lo conocemos.**

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

Cómo se haría una llamada a método en Java

- Se utiliza la notación del punto.

```
objeto.nombremétodo();
```

- donde ***objeto*** es el nombre de la variable que contiene la referencia al objeto.
- ***nombremétodo*** es el nombre del método al que queremos llamar.

Cómo se haría una llamada a método en Java

- Se utiliza la notación del punto.

```
objeto.nombremétodo();
```

- Por ejemplo

```
objAuto1.acelera();
```

Ejercicio

- Usando el ejercicio anterior, hagan que cuando se haga clic sobre el botón, se hagan las siguientes acciones:
  - Se cree un auto.
  - Se obtenga su información.
  - Se acelere el auto.
  - Se vuelva a obtener su información.
  - Se vuelva a acelerar el auto.
  - Se vuelva a obtener su información

Solución

```
<%@ page import="com.aurumsol.autos.*" %>
<% Auto auto1 = new Auto();%>
<HTML>
<HEAD>
<TITLE>Acelerando</TITLE>
</HEAD>
<BODY>
Cuando se crea: <%= auto1.informacion()%>
<% auto1.acelera(); %>
Despues de acelerar: <%= auto1.informacion()%>
<% auto1.acelera(); %>
Despues de reacelerar: <%= auto1.informacion()%>
</BODY>
</HTML>
```

Obtenemos esto

- Como vemos, cada vez que se ejecuta `auto1.acelera()`, la velocidad aumenta 10 (kilómetros por hora)

Cuando se crea:

Auto número 1.
Propietario: Desconocido.
Velocidad: 0

Despues de acelerar:

Auto número 1.
Propietario: Desconocido.
Velocidad: 10

Despues de reacelerar:

Auto número 1.
Propietario: Desconocido.
Velocidad: 20

¿Cómo aceleríamos el auto a 200 kilómetros por hora?

- A parte de ser muy salvaje, parece un poco tedioso.
- ¿No hay una mejor opción?

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

Parámetros

- Cambien el proyecto anterior para que cuando se haga clic se ejecute el siguiente fragmento de código.

```
Auto auto1 = new Auto();
auto1.fijaVelocidad(200);
```

- Ejecuten la aplicación Web y comprueben la información del objeto **auto1**.

Solución

```
<%@ page import="com.aurumsol.autos.*" %>
<% Auto auto1 = new Auto();
   auto1.fijaVelocidad(200);%>
<HTML>
<HEAD>
<TITLE>Acelerando a lo salvaje</TITLE>
</HEAD>
<BODY>
Informacion: <%= auto1.informacion(); %>
</BODY>
</HTML>
```

El auto tiene velocidad de 200 en una sola acción

Informacion:

Auto número 2.
Propietario: Desconocido.
Velocidad: 200

- Esto es porque se ha hecho la llamada a método (mensaje)

```
auto1.fijaVelocidad(200);
```

Parámetros

```
auto1.fijaVelocidad(200);
```

- Al llamar al método **fijaVelocidad**, hemos especificado la velocidad que queremos dar al objeto.
- En general, cuando llamamos a un método, les proporcionamos ("pasamos" en el argot informático) unos datos que necesita para su ejecución.
- Estos datos se llaman parámetros. Como ven es lo mismo que los parámetros de un procedimiento en la programación convencional.

Terminología

- A los datos que proporcionamos a los métodos para que puedan ejecutarse, se les llama "parámetros".
- Al acto de proporcionar un parámetro a un método cuando se le llama, se le llama "pasar un parámetro"

Cómo se pasan parámetros en Java

- Para un procedimiento que tiene parámetros, la forma de llamarlo es la siguiente:

```
objeto . nombremétodo ( valorespar ) ;
```

- donde ***objeto*** es la variable que guarda su referencia al objeto.
- donde ***valorespar*** es la lista de los parámetros que queremos pasar separada por comas.
- Por ejemplo

```
auto1.fijaVelocidad(200);
```


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

Ejercicio

- 1. Creen un formulario HTML con dos cuadros de texto en los que pondrán la velocidad del auto y su propietario.
- Cuando se hace clic en el botón, se debe crear un auto, darle la velocidad y propietario que se ha introducido en el formulario HTML y mostrar la información por pantalla.
- Utilicen el método **`fijaPropietario`** (que tiene un parámetro de tipo **`String`**) para cambiar el propietario de un auto.

Resultado.
Desplieguen y ejecuten lo siguiente.

```
<%@ page import="com.aurumsol.autos.*" %>
<% Auto auto1 = new Auto();
   auto1.fijaVelocidad(200);%>
<HTML>
<HEAD>
<TITLE>Resultado</TITLE>
</HEAD>
<BODY>
<H1>El auto tiene una velocidad de <%=
  auto1.consigueVelocidad()%> kilómetros
  por hora.</H1>
</BODY>
</HTML>
```

Se muestra la posición horizontal y vertical del objeto

El auto tiene una velocidad de 200 kilómetros por hora.

```
<H1>El auto tiene una velocidad de
<%= auto1.consigueVelocidad() %>
kilómetros por hora.</H1>
```

Resultado

- Algunos métodos proporcionan una información cuando se acaba su ejecución.
- A esta información se le llama **resultado o valor de retorno.**
- Cuando el método proporciona esta información, se dice que **devuelve o retorna un resultado.**

Resultado

- Los métodos en Java pueden devolver un resultado (o pueden no devolver nada).
- El resultado puede ser usado en cualquier expresión válida en Java.

Atención

- Los métodos se diferencian de los procedimientos y funciones convencionales en que están ligados a un objeto.
- Los convencionales eran totalmente autónomos.
- Sin embargo, los métodos son procedimientos o funciones que realizamos sobre un objeto y se refieren a ese objeto y a la información de ese objeto.

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

consigueVelocidad()

- Es un método que retorna como resultado la velocidad del objeto de clase **Auto** que estamos tratando.
- Hay otro método **consiguePropietario**, que retorna el propietario del objeto.

Ejecutando un método con resultado

- En principio es lo mismo que con un procedimiento:

objeto . nombremétodo (valorespar)

- donde ***valorespar*** es la lista de los parámetros que queremos pasar separada por comas.
- Sin embargo ahora, se devuelve un resultado. Con él, hay que hacer algo (integrarlo en una expresión, pasarlo como parámetro a un método, asignarlo, etc)
- Por ejemplo

```
velocidad =
  auto1.consigueVelocidad();
```

Ejecutando un método con resultado

- Normalmente, llamamos al método y hacemos algo con el resultado

```
velocidad =
  auto1.consigueVelocidad();

<%=auto1.consigueVelocidad(); %>
```

Entendiendo las cosas

- Ahora podemos entender cuando hacíamos

```
<%= auto1.informacion() %>
```

informacion() es un método que devuelve un resultado que es una cadena (**String**) con información sobre el objeto.

- Este valor se coloca en el HTML de la página Web mediante la expresión JSP.

Ejercicio

- Creen un formulario con dos cuadros de texto, donde introduciremos dos numeros.
- El primer número es la velocidad inicial de un auto. El segundo es las veces que apretamos el acelerador (es decir, que ejecutamos **acelera()**).
- La JSP debe crear el objeto, fijar la velocidad inicial y después acelerar tantas veces como dice el segundo número.
- Finalmente, debemos obtener la velocidad del auto con **consigueVelocidad** y mostrarla por pantalla.

Uso de un objeto

- Consistirá en dos tipos de modalidades de uso diferente:
- Acceder a los atributos de un objeto.
- Ejecutar los métodos del objeto.

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

Recordemos: Las dos cosas que nos interesan de un objeto.

- Sus características o **atributos.**
  - Su velocidad.
  - Su propietario
- El tipo de acciones que pueden realizarse con él o **métodos**
  - Acelerar.
  - Decelerar.
  - Cambiar el propietario.

- A los atributos y métodos se les llama **miembros.**

Ya hemos visto los métodos

- Nos queda por ver los atributos, que veremos a continuación.
- Sabemos que una de las cosas que nos interesan de un objeto son sus características, que llamamos atributos.
- Los atributos no son más que una clase de variables.

Los atributos son una clase especial de variables

- Son unas variables que están ligadas al objeto (que pertenecen a un objeto), de la misma forma que los métodos son procedimientos y funciones pero ligados al objeto.
- Se diferencian de otro tipo de variables en que:
  - Se definen dentro y asociadas a un objeto.
  - Están accesibles dentro de todo ese objeto.

Conocer los atributos de un objeto

- Más adelante, veremos maneras en Eclipse de saber cuáles son los atributos de un objeto.
- Por ahora, nos basta con saber que los objetos de **`Auto`** tienen los atributos **`velocidad`** y **`propietario`**.

Ejemplo. Despliegen y ejecuten lo siguiente.

```
<%@ page import="com.aurumsol.autos.*" %>
<% Auto auto1 = new Auto();
   auto1.fijaVelocidad(150); %>
<HTML>
<HEAD>
<TITLE>Atributos</TITLE>
</HEAD>
<BODY>
<H1>El auto tiene una velocidad de <%=
  auto1.velocidad %> kilometros por
  hora.</H1>
</BODY>
</HTML>
```

Aparece la velocidad del objeto

El auto tiene una velocidad de 150 kilometros por hora.

- **`velocidad`** es el atributo que guarda la velocidad del objeto.

Obtener el valor de un atributo

- Para obtener el valor de un atributo, se utiliza la notación del punto:

objeto.nombreatributo

- donde ***objeto*** es el nombre de la variable que contiene la referencia al objeto.
- ***nombreatributo*** es el nombre del atributo del cual queremos saber su valor.

Así, para obtener la velocidad

- Obtenemos el valor del atributo **`velocidad`**, con la siguiente expresión:

```
auto1.velocidad
```

- Esto nos devuelve un valor, que podemos incluir en cualquier expresión, mostrarlo por pantalla, etc.

Sabemos que los atributos son una clase especial de variables.

- Hasta ahora, hemos recuperado su valor con la notación del punto.
- Pero de una variable, nos interesan dos acciones: recuperar el valor y guardar un valor.
- ¿Cómo guardamos (modificamos) el valor de un atributo?

Modificando el valor de un atributo

- Para guardar o saber el valor de un atributo, se utiliza la asignación:

objeto.nombreatributo = expresión ;

- donde ***expresión*** es el nombre del atributo del cual queremos saber su valor.
- Por ejemplo

```
auto1.velocidad = 15;
```

Ejercicio

- Escriban una aplicación Web que pida la velocidad y el propietario de un auto.
- La aplicación debe crear un objeto Auto y ponerle esa velocidad y propietario, usando atributos, no métodos. Después comprueben que efectivamente el objeto tiene esa velocidad y propietario, también con atributos.
- Despliegen la aplicación y comprueben qué es lo que pasa .

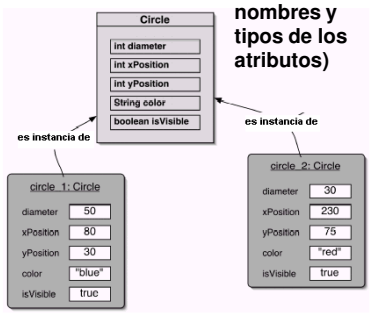
Aclarando las ideas

- Sabemos que los atributos son variables y, por lo tanto, nos interesa su nombre, su tipo y su valor.
- Los objetos de una misma clase (por ejemplo, de la clase **`Auto`**) pueden tener diferente estado, es decir, diferente valor de los atributos.
- En cambio, el nombre y el tipo de los atributos es el mismo para todos los objetos de la misma clase.

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

Una clase y sus objetos

CLASE (Define los nombres y tipos de los atributos)



OBJETOS (Definen los valores de los atributos)

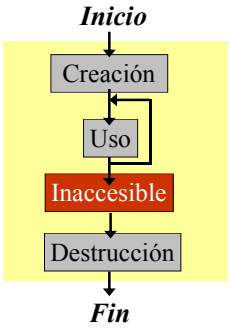
Comparación con el mundo real

- La clase (especificación) Auto define cuál son los atributos: velocidad, propietario y el tipo de ellos.
- Cada objeto, cada auto, tiene una velocidad diferente y un propietario diferente.

Estado

- Sabemos que una de las cosas que nos interesan de un objeto son sus características, que llamamos atributos.
- Cada uno de estos atributos tiene un valor.
- A partir de ahora, al conjunto del valor de los atributos, se le llamará **Estado**.
- En nuestro caso, el estado es el conjunto de valores de los atributos **`velocidad`**, **`propietario`**.

Ciclo de vida de un objeto



Un objeto inaccesible

- Antes de que el objeto se destruya, se hace inaccesible.
- Esto quiere decir que no podemos hacer nada con él: no podemos acceder a sus atributos ni ejecutar sus métodos.
- El objeto se sigue conservando en memoria, pero para el programa es como si ya no existiera.

Hacer inaccesible un objeto

- Un objeto se hace inaccesible porque no hay referencias que apunten a él. Ejemplo:

```
<%@ page import="com.aurumsol.autos.*" %>
<% Auto auto1;
   auto1 = new Auto();%>
<HTML><HEAD>
<TITLE>Objetos que se hacen
  inaccesibles</TITLE>
</HEAD>
<BODY>
</BODY>
</HTML>
```


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

Análisis de la ejecución de esta JSP

```
<% Auto auto1;
```

Crea una referencia `auto1`

```
   auto1 = new Auto();%>
```

Crea un objeto `auto1` → `:Auto`

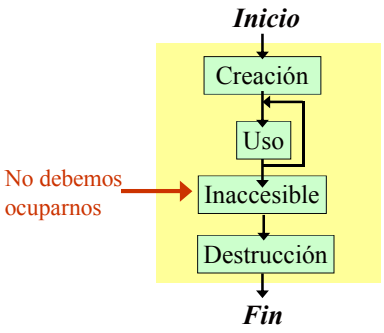
```
<HTML><HEAD><TITLE>Objetos que se hacen
  inaccesibles</TITLE>
</HEAD>
<BODY></BODY></HTML>
```

Se acaba la ejecución. La variable auto1, que referenciaba el objeto, se libera. Como el objeto no tiene referencias, el objeto queda inaccesible.

Si no hay mango, el objeto no se puede agarrar

- Si no hay referencias que apunten a él, el objeto se vuelve inaccesible.
- Fíjense que esto se produce automáticamente, sin que debamos programar nada.
- La mayoría de las veces no debemos preocuparnos.

Ciclo de vida de un objeto



Pero, ¿y si queremos hacerlo inaccesible?

- Si queremos hacer inaccesible en un objeto en medio de una JSP, podemos lograrlo asegurando que todas sus referencias no apunten a nada (o sea, apunten a `null`).
- Para hacer que una referencia apunte a `null`.

```
varReferencia = null;
```

`null` es el literal que indica que una referencia no apunta a nada

- Si queremos saber si la referencia apunta a un objeto basta con saber el valor de la expresión

```
varReferencia == null
```

El objeto se hace inaccesible cuando no hay referencias que lo apunten

```
<%@ page import="com.aurumsol.auto
<% Auto auto1;
   auto1 = new Auto();
   auto1 = null;
%>
<HTML><HEAD><TITLE>Objetos que se hacen
  inaccesibles </TITLE></HEAD><BODY></BODY>
</HTML>
```

EL OBJETO YA NO ES ACCESI-BLE EN ESTE PUNTO

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

El objeto se hace inaccesible cuando no hay referencias que lo apunten

Recordemos que un objeto inaccesible ya no existe para el programa

```
<%@ page import="com.aurumsol.auto
<% Auto auto1;
   auto1 = new Auto();
   auto1 = null;

   auto1.fijaVelocidad(20);
%>
<HTML><HEAD><TITLE>C          en
  inaccesibles </TIT</HEAD><BODY></BODY>
</HTML>
```

EL OBJETO YA NO ES ACCESI-BLE EN ESTE PUNTO

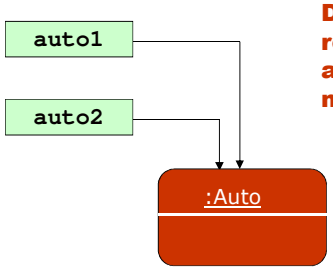
¡¡ERROR!!

Asignar a `null` no tiene porque hacerlo inaccesible

```
<% Auto auto1;
   auto1 = new Auto();
   Auto auto2 = auto1;
   auto1 = null;
   auto2.fijaVelocidad(23);  OK
%>
```

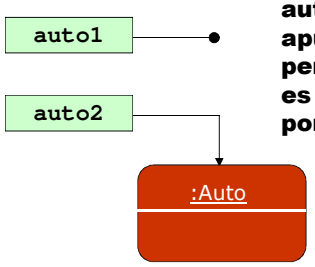
- La referencia **`auto1`** no apunta al objeto pero aún se puede acceder a él con la referencia **`auto2`** que sí que lo apunta.
- Solo se hace inaccesible cuando no hay **NINGUNA** referencia que lo apunta.

Antes de asignar el `null` la situación es la siguiente



Dos referencias apuntan al mismo objeto

Después de asignar el `null` la situación es la siguiente



auto1 no apunta a nada pero el objeto es referenciado por auto2

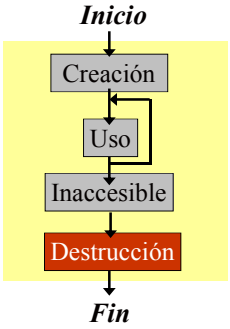
NO QUEDA INACCE-SIBLE

Sólo se hace inaccesible cuando no hay ninguna referencia que lo apunte.

Despliegen

- Despliegen este último fragmento de código y comprueben que el objeto no desaparece.
- Ahora, pongan a **`null`** la segunda referencia y comprueben que el objeto ya no está accesible.

Ciclo de vida de un objeto



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

El objeto ha quedado inaccesible

- Ya no podemos usar el objeto.
- Para nuestro programa, es como si hubiera desaparecido.
- Pero no se ha desaparecido: **sigue ocupando espacio en memoria**.

El objeto está "por ahí"

- Está en memoria. No podemos usarlo ni verlo, pero ocupa memoria.
- Decimos que el objeto se destruye, cuando se elimina de la memoria.
- La memoria liberada puede ocuparse en otras cosas.

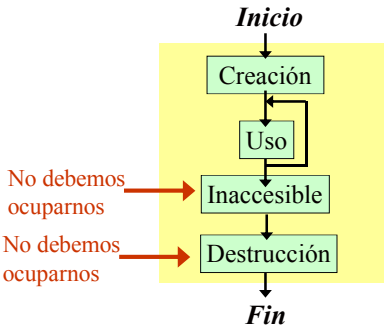
En Java, el programa que destruye los objetos

- Es un programa que se llama el "recolector de basura" ("garbage collector").
- Cuando la máquina virtual detecta que hay mucha memoria ocupada en objetos inaccesibles, activa automáticamente el recolector de basura.
- Este destruye todos los objetos inaccesibles y libera toda la memoria ocupada por ellos.

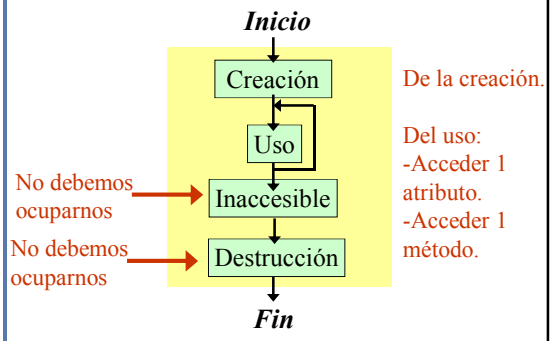
¿Y esto, qué nos importa a nosotros?

- Esa es la cuestión: **NO NOS IMPORTA.** Nosotros programamos y el recolector de basura ya se dedicará a liberar memoria, si hay mucha desperdiciada.
- En esto Java aventaja a lenguajes como C++ en el que el programador debe liberar todos los objetos que ha ocupado.
- Es una cosa menos de la que nos tenemos que ocupar.

Ciclo de vida de un objeto



En resumen, normalmente sólo debemos ocuparnos de dos cosas

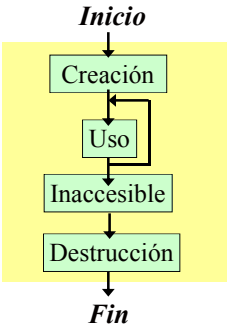


Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

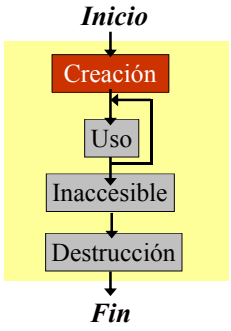
Ciclo de vida de un objeto

Pregunta: ¿Hemos cubierto todo lo que debemos saber de los objetos?



Ciclo de vida de un objeto

Respuesta: No, nos hemos dejado un aspecto muy importante de la creación de un objeto



Recordemos: Definición y creación de un objeto

- Para crear un objeto, debe indicarse a partir de qué clase se creará el objeto.
- Se crea el objeto siguiendo ese "molde" (esa especificación) que es la clase.

Recordemos: Definición y creación de un objeto

- Para crear un objeto, debe indicarse a partir de qué clase se creará el objeto.
- Se crea el objeto siguiendo ese "molde" (esa descripción) que es la clase.

Hasta ahora se crearon objetos de clases ya programadas.

- Hasta ahora habíamos considerado las clases "desde fuera": nos limitábamos a usarlas sin saber cómo estaban programadas. Éramos usuarios de las clases.
- Esto simplifica la programación y es la forma en qué debemos considerarlas cuando estamos usando unas clases mientras programamos código.
- El problema es que es lo único que sabemos.

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

Sólo sabemos crear objetos de clases ya programadas.

- Esto nos limita, ya que no podemos crear un objeto con características diferentes a las que los otros han programado por nosotros.
- En el ejemplo anterior, no podemos crear un objeto con forma de estrella.

Sólo sabemos crear objetos de clases ya programadas.

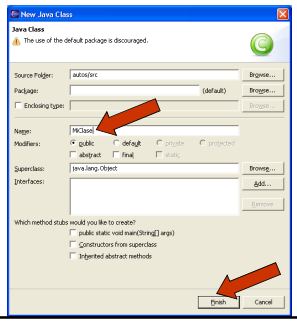
- Conclusión: Para poder crear cualquier tipo de objeto que deseemos, tenemos que saber programar clases ("moldes").
- Y eso aún no lo sabemos.

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**

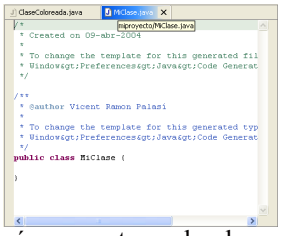


Paréntesis: el nombre de una clase

- 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?

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

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

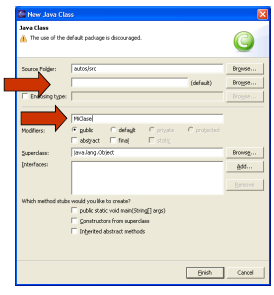
- Recuerden que el nombre de la clase empieza por mayúscula y que las siguientes palabras del nombre comienzan por mayúscula.

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 ***nombrepaquete.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 especifico 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 es de 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 = new Auto();
private int velocidad;
```


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

Ejercicio

- Crear un nuevo proyecto.
- Crear en él una clase que modele una libreta de banco. En principio, sólo nos interesará el saldo de la libreta y su número de libreta.
- Por simplicidad, suponemos sólo consideramos cifras enteras.
- Escribir esta clase especificando sólo los atributos.
- Háganlo en Eclipse.

Ejercicio

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

Solución

```
package com.aurumsol.planilla;
public class Empresa {
  public String razonSocial;
  public int facturacionAnual;
}
```

Ejercicio

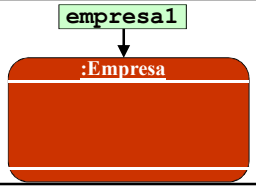
- Queremos programar una clase que recoja toda la información de un empleado es decir, su nombre, sueldo y empresa. Escribir esta clase especificando sólo los atributos.
- Háganlo en Eclipse.

Solución

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

Uso de estas clases (1)

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

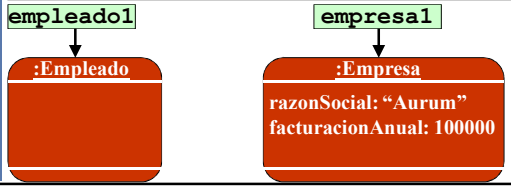


Uso de estas clases (2)

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

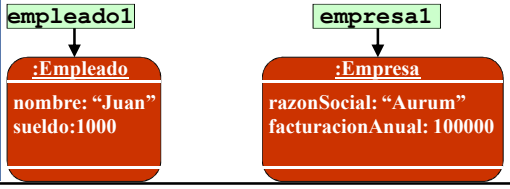
Uso de estas clases (3)

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



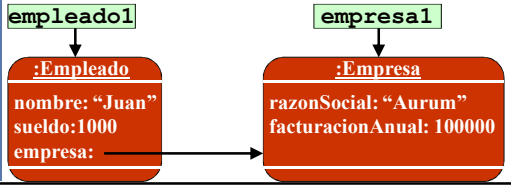
Uso de estas clases (4)

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



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;
```



Pregunta

- Supongamos que tenemos una variable **`empleado1`**. ¿Cómo obtendríamos la facturación anual de su empresa?

Solución

- Se puede hacer de dos formas.

```
Empresa empresaEmpleado1=empleado1.empresa;
int facturacionEmpresaEmpleado1 =
  empresaEmpleado1.facturacionAnual;
```

- O bien.

```
int facturacionEmpresaEmpleado1 =
  empleado1.empresa.facturacionAnual;
```

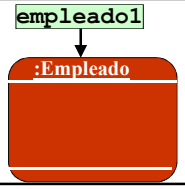

Mail: palasi@palasi.com Web: www.palasi.com

Pregunta

- Añadir un atributo a la clase Empleado que especifique el Jefe de ese empleado.

Uso de estas clases (1)

```
Empleado empleado1= new Empleado();
```

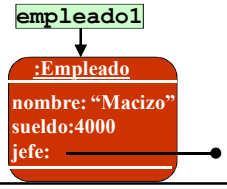


Uso de estas clases (2)

```
Empleado empleado1= new Empleado();
empleado1.nombre ="Macizo";
empleado1.sueldo = 4000;
```

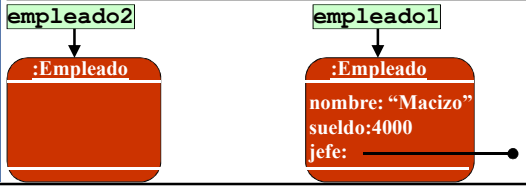
Uso de estas clases (3)

```
Empleado empleado1= new Empleado();
empleado1.nombre ="Macizo";
empleado1.sueldo = 4000;
empleado1.jefe = null;
```



Uso de estas clases (4)

```
Empleado empleado1= new Empleado();
empleado1.nombre ="Macizo";
empleado1.sueldo = 4000;
empleado1.jefe = null;
Empleado empleado2 = new Empleado();
```

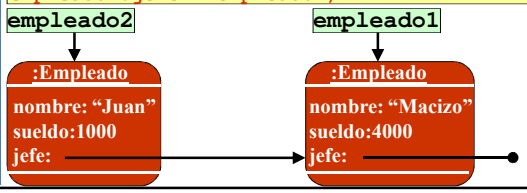


Uso de estas clases (5)

```
Empleado empleado1= new Empleado();
empleado1.nombre ="Macizo";
empleado1.sueldo = 4000;
empleado1.jefe = null;
Empleado empleado2 = new Empleado();
empleado2.nombre ="Juan"
empleado2.sueldo = 1000;
```

Uso de estas clases (6)

```
Empleado empleado1= new Empleado();
empleado1.nombre ="Macizo";
empleado1.sueldo = 4000;
empleado1.jefe = null;
Empleado empleado2 = new Empleado();
empleado2.nombre ="Juan"
empleado2.sueldo = 1000;
empleado2.jefe = empleado1;
```



Sintaxis de una clase

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

- Vamos a ver la declaración de métodos.

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.

Ejemplo

```
public void escribeSuma(int s1,int s2) {
  out.print(s1+s2);
}
```

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

- Método que recibe dos enteros como parámetros e escribe su suma en la página HTML (para funcionar debería declararse en la JSP). Observen la forma de declarar los parámetros.
- Observen también que no devuelve resultado, pues aparece la palabra **`void`**.

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

Otro ejemplo

```
public int suma(int s1,int s2) {
   return s1+s2;
}
```

ámbito tipores nombre(*listaparam*) {
    Sentencias
}

- Método que recibe dos enteros como parámetros y devuelve su suma. Observen que el tipo del resultado es **`int`**
- Observen también que el valor del resultado se indica con la sentencia **`return.`**

Ejercicio

- Completar el ejemplo de la libreta de banco con métodos para inicializar la libreta (entonces se fijará un número de libreta), añadir depósitos, añadir retiros, obtener el saldo y el número de libreta.
- Por simplicidad, suponemos sólo consideramos cifras enteras.

Solución

```
package com.aurumsol.banco;
public class Libreta {
  private int numero, saldo;
  public void iniciar(int nLibreta){
     numero = nLibreta;
     saldo = 0;
  }
  public int obtenerNumero(){
     return numero;
  }
  public int obtenerSaldo(){
     return saldo;
  }
```

Solución

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

Nota importante: Encapsulación

- Como vemos los atributos son privados y se accede a ellos con métodos públicos.
- Esto nos permite un mayor control. Por ejemplo, no podemos introducir un saldo negativo, lo que sería posible con un atributo saldo público.
- Esta es la práctica habitual. Poner los atributos privados y gestionar su acceso a través de métodos, lo que nos permite más control y menos errores.

Paréntesis. Comentarios en las clases

```
/* Clase que modela
una libreta de banco */
package com.aurumsol.banco;
public class Libreta {
  private int numero, saldo;//Atributos
  //Aquí comienzan los métodos
}
```

- Dos tipos de comentarios:
- 1. Precedidos por **`//`**, llegan hasta el final de la línea.
- 2. Encerrados entre **`/*`** y **`*/`**, pueden tener varias líneas.

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

La importancia de los comentarios

```
/* Clase que modela
una libreta de banco */
package com.aurumsol.banco;
public class Libreta {
  private int numero, saldo;//Atributos
  //Aquí comienzan los métodos
}
```

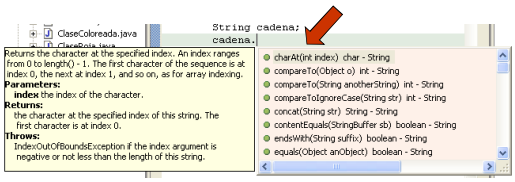
- Es vital añadir comentarios al código para hacerlo leíble y mantenible.
- En estas transparencias, no lo haremos por falta de espacio.

Paréntesis. Ayudas de Eclipse a la programación de clases

- Eclipse ayuda a la programación de clases de muchas maneras. Las más fundamentales son:
  - Informándonos de los atributos y métodos que tiene un cierto objeto.
  - Informándonos de los atributos y métodos que tiene una clase.
  - Indicándonos los errores sin necesidad de compilar.
  - Sugiriéndonos soluciones a esos errores.

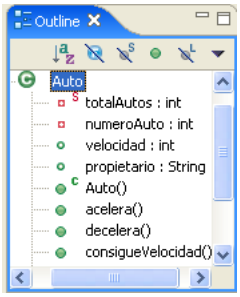
Informándonos de los miembros que puede tener un cierto objeto

- Cuando después del objeto ponemos un punto (y esperamos) aparece una lista de miembros. Podemos recorrerlos y Eclipse nos explicará cada uno con detalle.



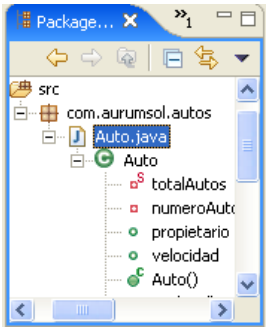
Informándonos de los miembros que puede tener una cierta clase

- Si estamos editando una clase, sus miembros aparecerán en la ventana Outline.
- Los atributos aparecen como círculos o cuadrados vacíos. Los métodos como círculos o cuadrados llenos.
- Los miembros privados aparecen en rojo y los públicos en verde.



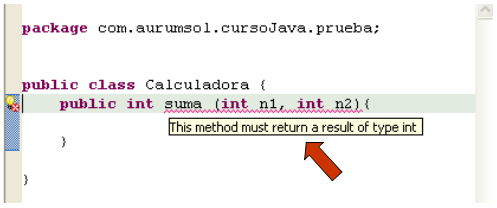
Informándonos de los miembros que puede tener una cierta clase

- Si no estamos editando una clase, también podemos conocer sus miembros expandiendo el nodo de la clase en el Package Explorer.
- Aparecerá la misma información que acabamos de ver.



Indicándonos los errores

- Eclipse señala los errores subrayándolos en rojo. Si ponemos el cursor encima de ellos, nos informa de cuál es el error.



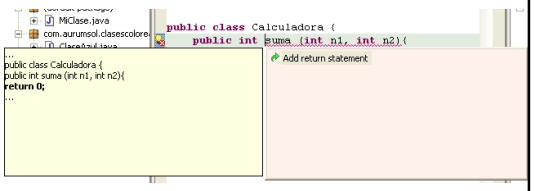
Mail: palasi@palasi.com Web: www.palasi.com

Nota importante

- Eclipse, por defecto, compila automáticamente mientras escribimos la clase.
- Por ello, no hay necesidad de que nos acordemos de compilar.
- Si queremos desactivar esta opción por defecto, podemos desactivar **Project|Build Automatically** y, entonces, cada vez que queramos compilar deberemos hacer **Project|Build Project**.

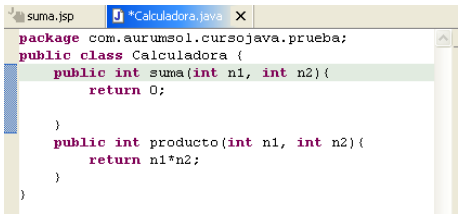
A la izquierda del error hay una bombilla

- Si hacemos clic sobre ella, nos dará sugerencias para solucionar el error.



Si hacemos clic en una sugerencia

- Eclipse la aplica automáticamente.



Ejercicio

- Completen la clase Empleado (con atributos nombre, sueldo y empresa) con métodos que permitan inicializar el Empleado, fijar el sueldo, devolver el nombre, devolver el jefe y devolver el sueldo.

Solución (1)

```
package com.aurumsol.planilla;
public class Empleado {
   private String nombre; public int sueldo;
   private Empresa empresa;
   public boolean iniciar(String nuevoNombre, int
     nuevoSueldo, Empresa nueEmpresa){
          if (nuevoSueldo<134){//Salario mínimo
               return false;
          } else {
               nombre = nuevoNombre;
               sueldo = nuevoSueldo;
               Empresa = nueEmpresa;
               return true;
          }}
```

Solución (2)

```
public boolean fijarSueldo (int nuevoSueldo){
   if (nuevoSueldo<134){//Salario mínimo
        return false;
   } else {
        sueldo = nuevoSueldo;
        return true;
   }
}
public String consigueNombre(){
        return nombre;}
public int consigueSueldo(){
        return sueldo;}
public Empresa consigueEmpresa(){
        return empresa;}}
```


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

Nota importante

- Fíjense que el hecho de hacer los atributos privados hace que yo pueda controlar los valores que se meten en ellos.
- Por ejemplo, puedo garantizar que el sueldo no es menor que el salario mínimo.

Variables locales

- Observemos el siguiente método:

```
public double media(double s1,double s2){
  double suma = s1+s2;
  return (suma/2.0);
}
```

- Hemos utilizado una variable auxiliar **suma** para realizar un cálculo auxiliar.
- Esto mejora la legibilidad y puede ser necesario o muy conveniente en métodos más complicados.

Variables locales

- Observemos el siguiente método:

```
public double media(double s1,double s2){
  double suma = s1+s2;
  return (suma/2.0);
}
```

- La variable **suma** sólo existe dentro del método. Fuera de él, no está accesible..

Variables locales

- Observemos el siguiente método:

```
public double media(double s1,double s2){
  double suma = s1+s2;
  return (suma/2.0);
}
```

Son variables que:

- Se declaran dentro de un método.
- Sólo están accesibles dentro de este método.

Declaración de variables locales

- Se declaran igual que cualquier otra variable.

tipo nombre = valorInicial;

- Pero dentro del método.

(Las partes en negro son opcionales)

Acceso a variables locales

nombrevariable
nombrevariable = expresión;

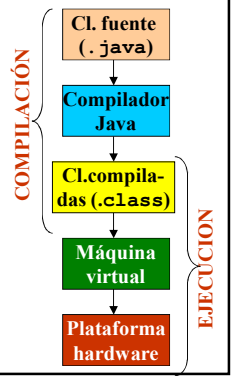
- Se accede a ellas como cualquier otra variable.
- Pero sólo son accesibles dentro del método.
- Si las accedemos fuera, dan error de compilación.

Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

Nota importante

- Eclipse nos oculta un poco esto, pero:
- Las clases Java antes de compilar se guardan en archivos con extensión `.java`.
- Las clases compiladas se guardan en archivos `.class`



¡No nos lo creemos!

- Apliquen el método de Santo Tomás, buscando la clase Java fuente en el directorio `workspace\`***`nombreproyecto`***`\src` y la clase compilada en `workspace\`***`nombreproyecto`***`\bin` .
- Por ejemplo, si nuestro proyecto se llama `libreta`, la clase es `Libreta` y su paquete es `com.aurumsol.banco`.
- La clase Java fuente será
  `workspace\libreta\src\com\aurumsol\banco\Libreta.java`
- La clase compilada será
  `workspace\libreta\bin\com\aurumsol\banco\Libreta.class`
- Abran la clase fuente y vean que hay.

Importando clases en una JSP

- Cuando queremos usar clases en una JSP, haremos:
  `<%@ page import="`***`paquete.Clase`***`" %>` o bien
  `<%@ page import="`***`paquete.*`***`" %>`
  (que quiere decir todas las clases del paquete).
- A esto se le llama "importar" las clases en la JSP.
- Ahora ya sabemos porque usábamos
  `<%@ page import="com.aurumsol.autos.*"%>`
  en las primeras JSP. Así podíamos usar la clase Auto desde la JSP. En caso contrario, no hubiéramos podido. NO espacios alrededor de =

Importando clases en una JSP

- También se pueden importar varios paquetes o clases en una sola línea. Así:

```
<%@ page import="paquete1.*,
  paquete2.*, ..., paqueten.*" %>
```

Tenemos la JSP

```
<%@ page import="com.aurumsol.banco.*" %>
<% Libreta lib = new Libreta();
lib.iniciar(12345); lib.depositar(1000);
boolean exito = lib.retirar(500);%>
<HTML><HEAD><TITLE>Mov.</TITLE></HEAD><BODY>
<%if (exito){
     out.print("Operaciones efectuadas");
}else {
     out.print("Retiro no efectuado");
}%> </BODY></HTML>
```

Esta JSP usa la clase `Libreta` (1)

```
package com.aurumsol.banco;
public class Libreta {
  private int numero, saldo;
  public void iniciar(int nLibreta){
     numero = nLibreta;
     saldo = 0;
  }
  public int obtenerNumero(){
     return numero;
  }
  public int obtenerSaldo(){
     return saldo;
  }
```

Esta JSP usa la clase `Libreta` (2)

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

¿Cómo desplegamos esta aplicación Web?

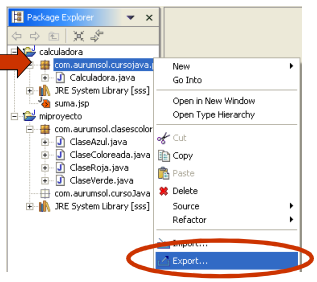
- La JSP lo tenemos claro. La colocamos en **`webapps/ROOT`**
- Para desplegar la clase hay diferentes formas. Utilizaremos la más usada y la más cómoda en grandes proyectos: los archivos JAR.

Archivos JAR

- Un archivo JAR es un archivo comprimido independiente de la plataforma que contiene clases Java.
- Es una clase especial de archivo ZIP.
- Normalmente contiene archivos **`.class`** (clases compiladas) junto con otros archivos necesarios para la ejecución de una aplicación: archivos de imágenes, etc.
- Tiene muchas utilidades: el despliegue de clases en servidores de servlets, la transferencia de archivos en un applet, la organización de paquetes, para comprimir aplicaciones, etc.

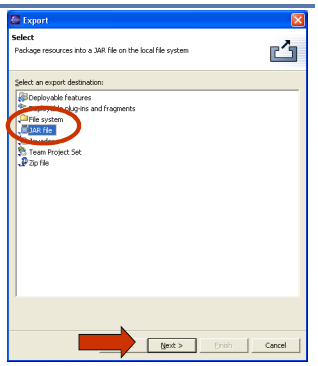
Para desplegar las clases

- Primero vemos que no hay errores en la clase.
- En el Package Explorer, hacemos clic derecho en el nombre del paquete que contiene las clases
- En el menú que aparece, seleccionamos la opción **Export**



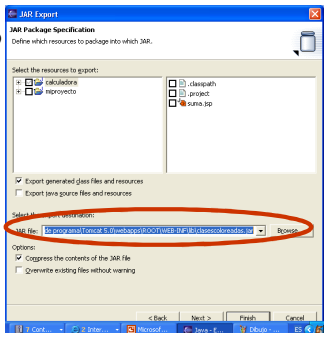
En la ventana que aparece, se selecciona JAR file.

- Y se hace clic en **Next**



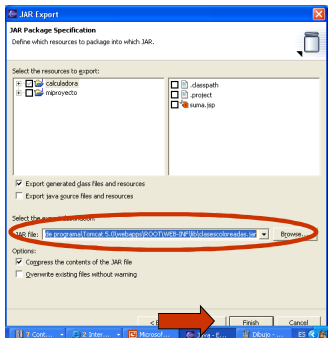
En la ventana que aparece, nos pedirá la ruta donde se creará el JAR

- La ruta es el directorio seguido del nombre. Es bueno poner un nombre orientativo. Por ejemplo, **`calculadora.jar`**



En cuanto al directorio, lo mejor es poner el de despliegue de los JAR

- De esta manera, crearemos el JAR y desplegaremos en un solo paso
- Hacemos clic en **Finish**.



¿Cuál es el directorio de despliegue de los archivos JAR?

- Es **`webapps\ROOT\WEB-INF\lib`** de la instalación de Tomcat.
- Si este directorio no existe deberá crearse.
- Recordemos que los JAR son una especie de archivo ZIP con clases.
- Desplegando el archivo JAR desplegamos las clases.

Resumen

- Para desplegar una aplicación Web con clases, se debe
- Desplegar las JSP de la aplicación Web en **`webapps\ROOT`**
- Empaquetar las clases Java en archivos JAR y desplegarlas en **`webapps\ROOT\WEB-INF\lib`**

Nota importante

- Eclipse nos oculta un poco esto, pero:
- Las clases Java antes de compilar se guardan en archivos con extensión **`.java`**.
- Las clases compiladas se guardan en archivos **`.class`**
- Los archivos JAR son una clase de archivos ZIP que contienen clases compiladas **`(.class)`**. Compruébenlo (con un programa de tipo Winzip).

Ejercicio

- Desplegar la aplicación anterior de la libreta.
- Modificarla para que el número de libreta y los montos del depósito y del retiro se introduzcan desde un formulario HTML.

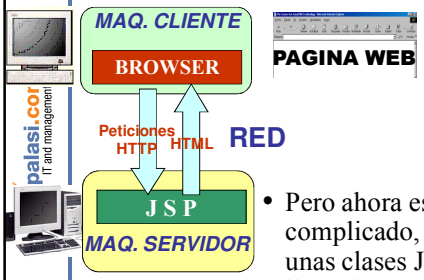
Solución (1)

```
<%@ page import="com.aurumsol.banco.*" %>
<%
int nLibreta =
Integer.parseInt(request.getParameter("nlib"));
int mDeposito =
Integer.parseInt(request.getParameter("depos"));
int mRetiro =
Integer.parseInt(request.getParameter("retiro"));
Libreta lib = new Libreta();
lib.iniciar(nLibreta); lib.depositar(mDeposito);
boolean exito = lib.retirar(mRetiro);
%>
```

Solución (2)

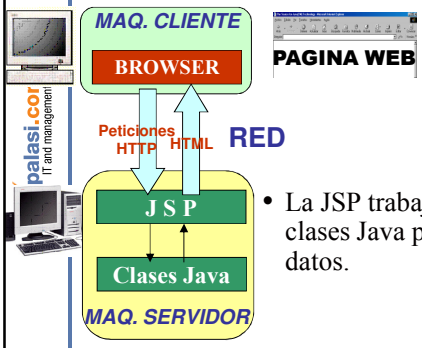
```
<HTML><HEAD><TITLE>Mov.</TITLE></HEAD><BODY>
<%if (exito){
     out.print("Operaciones efectuadas");
}else {
     out.print("Retiro no efectuado");
}%>
</BODY>
</HTML>
```

Recordemos que en el primer tema teníamos esta estructura del programa



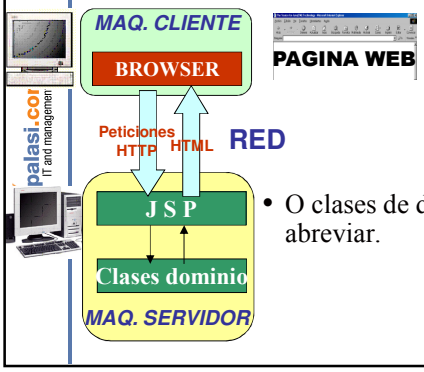
- Pero ahora es más complicado, pues tenemos unas clases Java que son llamadas por la JSP.

Ahora tenemos las clases Java



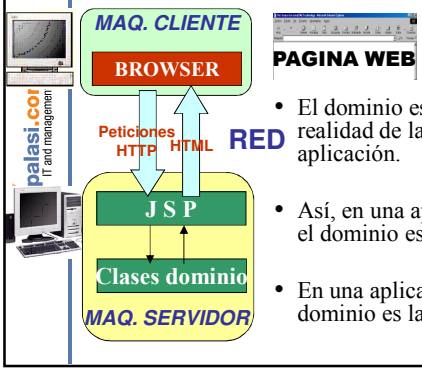
- La JSP trabaja y llama a las clases Java para manejar los datos.

Estas clases se les llama clases de modelo de dominio



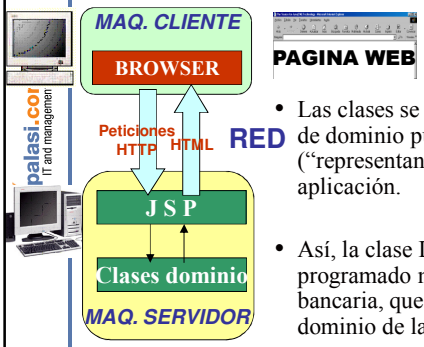
- O clases de dominio, para abreviar.

¿De dónde viene el nombre "modelo de dominio"?



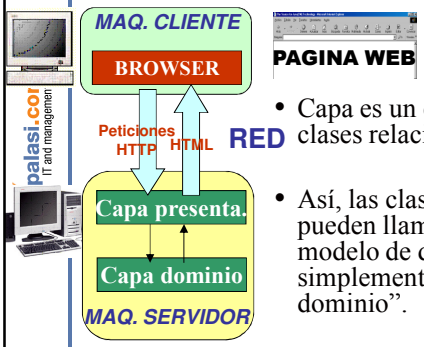
- El dominio es la parte de la realidad de la que trata nuestra aplicación.
- Así, en una aplicación bancaria, el dominio es el banco.
- En una aplicación contable, el dominio es la contabilidad.

¿De dónde viene el nombre "modelo de dominio"?



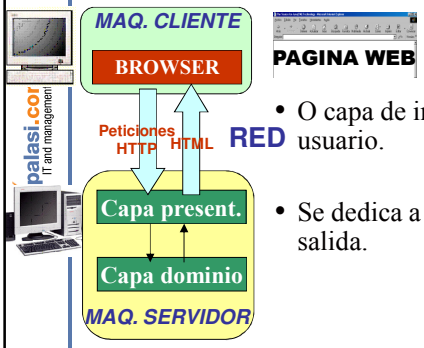
- Las clases se llaman de modelo de dominio pues modelan ("representan") el dominio de la aplicación.
- Así, la clase Libreta, que hemos programado modela una libreta bancaria, que es parte del dominio de la aplicación.

También se le puede llamar "capa de dominio"



- Capa es un conjunto de clases relacionados.
- Así, las clases que usamos se pueden llamar "Capa de modelo de dominio" o simplemente "capa de dominio".

A las JSP se les puede llamar "capa de presentación"



- O capa de interfaz de usuario.
- Se dedica a la entrada y salida.

Ejercicio

- Despliegue la aplicación Web anterior y comprueben que funciona.

Ejercicio

Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

Supongamos una clase `Cliente`

```
package com.aurumsol.clientes.dominio;
public class Cliente {
  private String nombre;
  private int DUI
  public void fijarNombre(String nueNombre){
     nombre = nueNombre;
  }
  public void fijarDUI(int nuevoDUI){
     DUI = nuevoDUI;
  }
}
```

Imaginemos una JSP que crea un nuevo cliente

```
<%@ page
  import="com.aurumsol.clientes.dominio.*"%>

<% //Se obtienen los datos del formulario
String nombre =
  request.getParameter("nombre");
int DUI = Integer.parseInt(
  request.getParameter("DUI"));

//Se crea un nuevo cliente
Cliente cliente1 = new Cliente();
cliente1.fijarNombre(nombre);
cliente1.fijarDUI(DUI);
%>
<HTML><HEAD><TITLE>Crear
  cliente</TITLE></HEAD>
<BODY>Se creo el nuevo cliente</BODY></HTML>
```

Esta es la solución que hemos visto hasta ahora

- Pero es una mala solución.
- Mezcla lo que son aspectos de proceso de datos con lo que son aspectos de presentación (entrada/salida).

Lo que está en rojo es proceso de datos, en verde presentación (entrada/salida)

```
<%@ page
  import="com.aurumsol.clientes.dominio.*"%>

<% //Se obtienen los datos del formulario
String nombre =
  request.getParameter("nombre");
int DUI = Integer.parseInt(
  request.getParameter("DUI"));
```

ENTRADA

```
//Se crea un nuevo cliente
Cliente cliente1 = new Cliente();
cliente1.fijarNombre(nombre);
cliente1.fijarDUI(DUI);
```

PROCESO

```
%>
<HTML><HEAD><TITLE>Crear
  cliente</TITLE></HEAD>
```

SALIDA

```
<BODY>Se creo el nuevo cliente</BODY></HTML>
```

No es buena idea mezclar el proceso de datos con la entrada/salida

- El código que aparece es complicado, enredado y difícil de mantener.
- Además, si queremos cambiar el diseño de la página Web, es difícil separar el código que procesa los datos del código que hace el diseño (los diseñadores Web no suelen saber programación).
- Si queremos cambiar la tecnología de generación de páginas Web (por ejemplo, usar Tapestry, PHP en vez de JSP) también resulta difícil de cambiar.
- Si queremos cambiar a otro tipo de interfaz (por ejemplo, de escritorio también es un problema).

Es mejor idea hacer el proceso de datos fuera de la JSP

- La JSP debería usarse sólo para entrada/salida.
- El proceso de datos debería programarse en un método de una clase, fuera de la JSP.

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

Por ejemplo, en la clase NegocioCliente

```
package com.aurumsol.clientes.negocio;
import com.aurumsol.clientes.dominio.Cliente;
public class NegocioCliente {
  public void creaNuevoCliente(String nombre,
  int DUI){
     Cliente cliente1 = new Cliente();
     cliente1.fijarNombre(nombre);
     cliente1.fijarDUI(DUI);
  }
}
```

- Hay un método que tiene el proceso de datos.
- (No se fijen por ahora en el **import**: lo veremos después).

Las clases en las que se hace el proceso de datos se llaman clases de negocio

- Suelen trabajar las clases de dominio.

Ahora la JSP queda

```
<%@ page
  import="com.aurumsol.clientes.negocio.*"%>

<% //Se obtienen los datos del formulario
String nombre =
  request.getParameter("nombre");
int DUI = Integer.parseInt(
  request.getParameter("DUI"));

//Se llama a la clase de negocio
NegocioCliente clase= new NegocioCliente();
clase.creaNuevoCliente(nombre,DUI);
%>
<HTML><HEAD><TITLE>Crear
  cliente</TITLE></HEAD>
<BODY>Se creo el nuevo cli></HTML>
```

ENTRADA

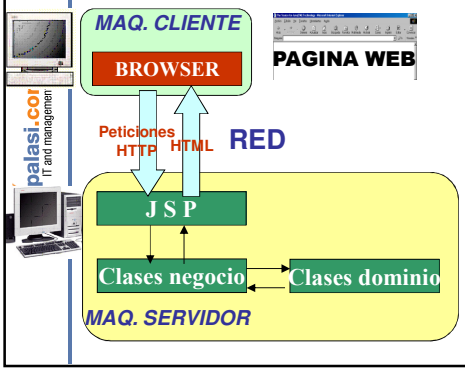
LLAMAR NEGOCIO

SALIDA

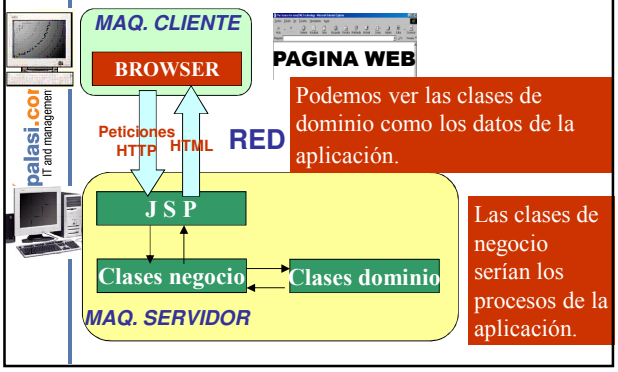
Ventajas de esta solución

- Podemos cambiar la JSP sin que cambiemos el procedimiento de crear un nuevo cliente (así, los diseñadores Web no deben ocuparse de la programación).
- Podemos cambiar el procedimiento de crear un nuevo cliente sin cambiar el diseño Web (así, los programadores no deben ocuparse del diseño Web).
- El código es más sencillo, ordenado y mantenible (esto no se ve mucho aquí porque sólo son tres líneas: es un caso real sería más evidente).

Ahora tenemos las siguiente estructura



¿Qué diferencia hay entre las clases de negocio y las de dominio?



Podemos ver las clases de dominio como los datos de la aplicación.

Las clases de negocio serían los procesos de la aplicación.

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

Por ejemplo, miremos las clases `NegocioCliente` y `Cliente`

```
public class Cliente {
  private String nombre; private int DUI;
  public void fijarNombre(String nueNombre){
     nombre = nueNombre; }
  public void fijarDUI(int nuevoDUI){
     DUI = nuevoDUI;}}
public class NegocioCliente {
  public void creaNuevoCliente(String nombre, int
  DUI){
     Cliente cliente1 = new Cliente();
     cliente1.fijarNombre(nombre);
     cliente1.fijarDUI(DUI);}}
```

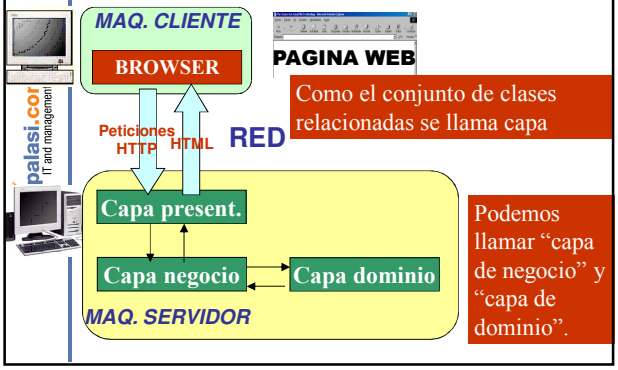
- La clase **`Cliente`** (dominio), contiene los datos de cada cliente.
- La clase **`NegocioCliente`** realiza los procesos (crear un cliente, inicializarlo). Para ello, usa la clase **`Cliente`**.

Dicho de otra manera

```
public class Cliente {
  private String nombre; private int DUI;
  public void fijarNombre(String nueNombre){
     nombre = nueNombre; }
  public void fijarDUI(int nuevoDUI){
     DUI = nuevoDUI;}}
public class NegocioCliente {
  public void creaNuevoCliente(String nombre, int
  DUI){
     Cliente cliente1 = new Cliente();
     cliente1.fijarNombre(nombre);
     cliente1.fijarDUI(DUI);}}
```

- **`Cliente`** = datos, **`NegocioCliente`** = procesos.
- En general, las clases de dominio almacenan los datos y las de negocio ejecutan los procesos.

Clases de negocio contienen procesos, Clases de dominio contienen datos



Fíjense que cada capa se suele colocar en un paquete diferente

```
package com.aurumsol.clientes.dominio;
public class Cliente {
  //Aquí más código que no ponemos
}
package com.aurumsol.clientes.negocio;
public class NegocioCliente {
  //Aquí más código que no ponemos
  }
}
```

- Normalmente una aplicación tiene un paquete (en nuestro caso, **`com.aurumsol.clientes`**) y cada capa tiene un subpaquete de éste (en nuestro caso, **`com.aurumsol.clientes.dominio`** y **`com.aurumsol.clientes.negocio`**).
- Así se les pedirá en el proyecto.

Un detalle. ¿Qué es ese `import`?

```
package com.aurumsol.clientes.negocio;
import com.aurumsol.clientes.dominio.Cliente;
public class NegocioCliente {
  public void creaNuevoCliente(String nombre,
  int DUI){
     Cliente cliente1 = new Cliente();
     cliente1.fijarNombre(nombre);
     cliente1.fijarDUI(DUI);
  }
}
```

- La clase **`NegocioCliente`** usa una clase (**`Cliente`**), que no es de su mismo paquete.
- Cuando una clase usa otra que es de paquete diferente, debe importarla (decir que va a usarla).

Importando una clase

- Para importar una clase de un paquete diferente del que estamos se hace.
  `import` ***`paquete.Clase;`***
- También se puede hacer lo siguiente:
  `import` ***`paquete.*;`***
  y se importarán todas las clases del paquete.
- Las sentencias **`import`** deben ir detrás de la instrucción package.
- La instrucción **`package`** debe ser siempre la primera de un archivo fuente Java.

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

Esto es lo mismo que importar clases en una JSP

- Recordemos que, para importar clases en una JSP, hacíamos:

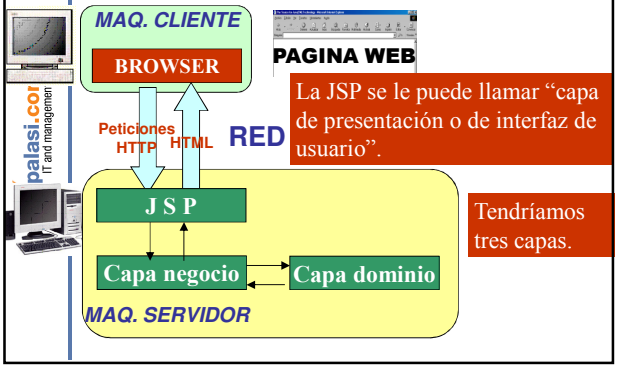
```
<%@ page import="paquete.Clase" %>
```

o bien

```
<%@ page import="paquete.*" %>
```

- En las clases es lo mismo pero
  - La sintaxis es diferente.
  - Sólo se importan las clases que son de diferente paquete.

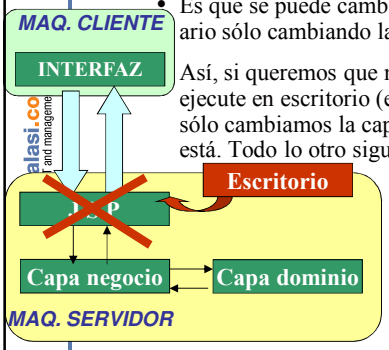
La JSP trata la E/S La capa de negocio los procesos y dominio los datos



La JSP se le puede llamar "capa de presentación o de interfaz de usuario".

Tendríamos tres capas.

Una ventaja de separar la presentación (JSP) de los procesos

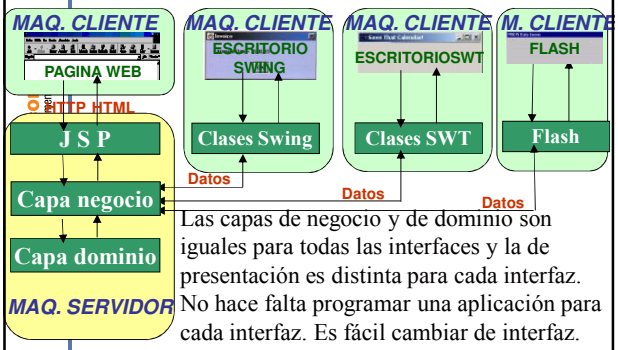


- Es que se puede cambiar la interfaz de usuario sólo cambiando la capa de presentación

Así, si queremos que nuestra aplicación se ejecute en escritorio (en vez de en el Web), sólo cambiamos la capa de presentación y ya está. Todo lo otro sigue igual.

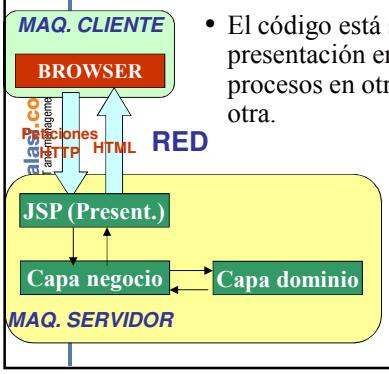
Incluso podemos tener un programa que pueda operarse con diferentes interfaces de usuario (escritorio, Web, applets).

Ejemplo: una aplicación con diferentes interfaces de usuario



Las capas de negocio y de dominio son iguales para todas las interfaces y la de presentación es distinta para cada interfaz. No hace falta programar una aplicación para cada interfaz. Es fácil cambiar de interfaz.

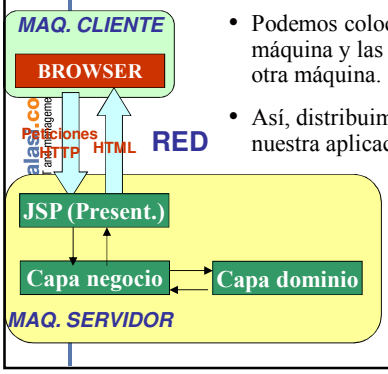
Más ventajas de esto



- El código está simple y claro: la presentación en una parte, los procesos en otra y los datos en otra.

Esto hace más fácil leer, depurar y mantener el programa.

Más ventajas de esto

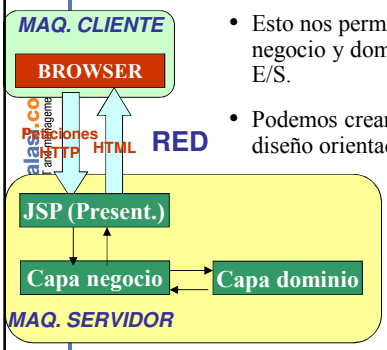


- Podemos colocar la JSP en una máquina y las clases de cálculo en otra máquina.
- Así, distribuimos la carga en nuestra aplicación.

En este caso, podemos poner un firewall entre las dos máquinas con lo cual aumentamos la seguridad

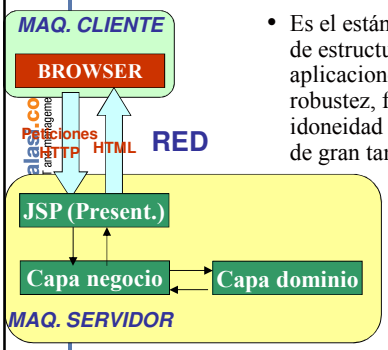
Mail: palasi@palasi.com Web: www.palasi.com

Más ventajas de esto



- Esto nos permite que las capas de negocio y dominio no contengan E/S.
- Podemos crearla con análisis y diseño orientada a objetos.

Más ventajas de esto



- Es el estándar internacional de estructurar las aplicaciones, por su robustez, flexibilidad y la idoneidad para aplicaciones de gran tamaño.

En general

- La estructura de programación que aprendemos aquí es muy complicada para lo sencillos que son los programas.
- Es como tener un camión para transportar una única piedra.
- Se hace para que sepan organizar aplicaciones reales, que son como éstas pero con más clases, métodos más elaborados.

Conclusiones

- A partir de ahora lo haremos así.
- Separaremos siempre los datos en la capa de dominio, los procesos en la capa de negocio y la presentación en la JSP.
- Aprenderemos a hacer las cosas bien desde el principio.

Pregunta

- Supongamos una aplicación que gestiona el Registro de Comercio. Digan en qué capas se colocarán estos fragmentos de código:
- El código que modela un comerciante.
- El código que crea un nuevo comerciante y le asigna una matrícula de comercio.
- El código que representa una sociedad.
- El código que escribe por pantalla los datos del comerciante.
- El código con los datos de la matrícula.
- El código que registra el pago de la matrícula de un comerciante para un año.

Ejercicio

- Recuerden la aplicación Web que, usando la clase **`Libreta`** que hemos programado anteriormente, que crea una nueva libreta y hace un depósito y un retiro, debiendo escribir el saldo a la salida.
- Modifíquenla de forma que contenga la estructura que acabamos de ver (con tres capas: JSP, negocio y dominio).

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

La clase Libreta forma la capa de dominio (1)

```
package com.aurumsol.banco.dominio;
public class Libreta {
  private int numero, saldo;
  public void iniciar(int nLibreta){
     numero = nLibreta;
     saldo = 0;
  }
  public int obtenerNumero(){
     return numero;
  }
  public int obtenerSaldo(){
     return saldo;
  }
```

La clase Libreta forma la capa de dominio (2)

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

La capa de negocio sería así

```
package com.aurumsol.banco.negocio;
import com.aurumsol.banco.*;
public class NgcLibreta {
   public boolean creaDepositaYRetira(int
     numLibreta, int deposito, int retiro){
     Libreta lib = new Libreta();
     lib.iniciar(numLibreta);
     lib.depositar(deposito);
     boolean exito = lib.retirar(retiro);
     return exito
  }
}
```

Ahora la JSP sería (1)

```
<%@ page
  import="com.aurumsol.banco.negocio.*"%>
<% //Se obtienen los datos del formulario
int numLibreta = Integer.parseInt(
  request.getParameter("NLIB"));
int deposito = Integer.parseInt(
  request.getParameter("DEPOSITO"));
int retiro = Integer.parseInt(
  request.getParameter("RETIRO"));

//Se llama a la clase de negocio
NgcLibreta ngc = new NegocioLibreta();
boolean exito = ngc.creaDepositaYRetira
  (numLibreta,deposito,retiro)%>
```

ENTRADA

LLAMAR NEGOCIO

Ahora la JSP sería (2)

```
<HTML><HEAD><TITLE>Libreta</TITLE></HEAD>
<BODY>
<%if (exito){
  out.print("Operaciones efectuadas");
}else {
  out.print("Retiro no efectuado");
}%>
</BODY></HTML>
```

SALIDA

Ejercicio

- Queremos crear una aplicación Web para la matriculación de alumnos en una Universidad. Lo haremos por partes.
- Primero, crearemos la capa de dominio compuesta por la clase Asignatura, de las que nos interesa su nombre, número de alumnos por salón y número de alumnos matriculados en toda la asignatura.
- Tendrá tres métodos: uno inicia la asignatura con el nombre y el número de alumnos por salón. Otro matricula un número de alumnos en la asignatura. El tercero obtiene el número de aulas ocupadas por los alumnos.

Solución: la capa de dominio (1)

```
package com.aurumsol.alumnos.dominio;
public class Asignatura {
 private String nombre;
 private int numeroPorAula, numeroTotal;
 public void iniciar (String nueNombre,int
  nueNumeroAula){
   nombre = nueNombre;
   numeroPorAula = nueNumeroAula;
   numeroTotal = 0;
 }
 public void matricular (int numeroAlumnos){
   numeroTotal += numeroAlumnos;
 }
```

Solución: la capa de dominio (2)

```
public int consigueTotalAulas(){
  int cociente = (numeroTotal/numeroPorAula);
  int residuo = (numeroTotal%numeroPorAula);
  if (residuo == 0){
     return cociente;
  }else{
     return (cociente+1);
  }
 }
}
```

Ejercicio

- Ahora programen la capa de negocio que deberá contener un método que recibe un nombre de asignatura, un número de alumnos por aula y un número de alumnos inicial.
- Este método deberá crear la asignatura, iniciarla y matricular los alumnos iniciales. Devolverá las aulas ocupadas.

Solución: la capa de negocio

```
package com.aurumsol.alumnos.negocio;
import com.aurumsol.alumnos.dominio.*;
public class NgcAsignatura {
  public int iniciaAsignatura (String nombre
  int alumnosPorAula, int alumnosIniciales){
   Asignatura materia = new Asignatura();
   materia.iniciar(nombre,alumnosPorAula);
   materia.matricular(alumnosIniciales);
   return (materia.consigueTotalAulas());
 }
}
```

Ahora hagamos la capa de presentación

- Creemos un HTML con tres cuadros: nombre de asignatura, alumnos por aula y alumnos a matricular.
- La JSP obtiene estos datos y los pasa al método de la capa de negocio. Escribe el número de aulas por pantalla.

La JSP es

```
<%@ page
  import="com.aurumsol.alumnos.negocio.*"%>
<% String nomAsignatura =
  request.getParameter("NASIGNATURA");
int alumnosAula = Integer.parseInt(
  request.getParameter("ALUMAULA"));
int alumnosMatri = Integer.parseInt(
  request.getParameter("ALUMMATR"));

NgcAsignatura ngc = new NgcAsignatura();
int aulas = ngc.iniciaAsignatura
(nomAsignatura,alumnosAula,alumnosMatri)%>
<HTML><HEAD><TITLE>Materia</TITLE></HEAD>
<BODY>Las aulas ocupadas son <%=aulas%>
  </BODY></HTML>
```

EN-TRA-DA

LLA MA NEG

SA LI DA

Programa del tema 2

- 2.1. Qué es la programación orientada a objetos.
- 2.2. Los conceptos de objeto y clase.
- 2.3. Ciclo de vida de un objeto.
- 2.4. Programación de clases.
- 2.5. Despliegue de clases.
- 2.6. Arquitectura en n-capas.
- 2.7 Más aspectos de programación de clases.

Un hecho desconocido (hasta ahora)

- Lo de la programación orientada a objetos pareciera no ir con las JSPs.
- Las JSPs parecen más documentos HTML que clases Java.
- En realidad, las JSPs también son clases.

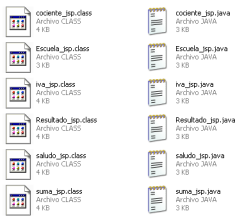
Las JSP son clases

- Dicho más propiamente, las JSPs se traducen a clases.
- Cada vez que accedemos por primera vez a una JSP que se acaba de desplegar
  - 1. La JSP se traduce a una clase.
  - 2. La clase generada se compila.
  - 3. La clase compilada se ejecuta.
- Es por eso que tarda tanto. En subsiguientes ejecuciones, sólo se sigue el paso 3.

¡No nos lo creemos!

- Sigan el método de Santo Tomás.
- Accedan a la carpeta `work\Catalina\localhost\_\org\apache\jsp` de la instalación de Tomcat.
- Verán unos archivos `.java` y unos `.class`

Estos archivos se corresponden a las JSPs que han desplegado



- Así, si han desplegado `suma.jsp`, aparecen dos archivos:
- `suma_jsp.java`.
- `suma_jsp.class`.

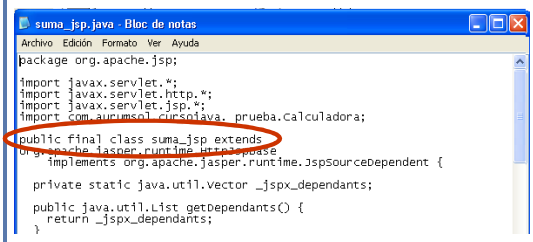
Estos archivos se corresponden a las clases fuentes y compiladas

- `suma_jsp.java`. **Corresponde a la clase en que se traduce la `suma.jsp` cuando es desplegada.**
- `suma_jsp.class`. **Es la clase anterior compilada. Es lo que se ejecuta cada vez que hacemos `http://dominio/suma.jsp`**

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

Abran `suma_jsp.java` (u otra similar)

- Es una clase Java. Muy rara para lo que sabemos, pero una clase.



Notas

- A las clases en las que se traducen las JSP se les llama servlets. De ahí la expresión servidor de servlets.
- Cada servlet tiene un método **`_jspservice`**. Búsquenlo.
- Cada vez que accedemos a la JSP, este método se ejecuta y hace toda la tarea que hasta ahora suponíamos que hacía la JSP.
- Es el método **`_jspservice`** el que ejecuta los resultados que hemos escrito en la JSP.

Ejemplo: `suma_jsp.java`

```
out.write('\n');
out.write('\n');
Calculadora calc = new Calculadora();int num1 = Integer.parseInt(
request.getParameter("num1"));int num2 = Integer.parseInt( request.getParameter("num2"));
out.write("\n");
out.write("<!DOCTYPE HTML PUBLIC \"-//W3C//dtd html 4.0 transitional//en\">\n");
out.write("<HTML><HEAD><TITLE>Calcular suma</TITLE></HEAD>\n");
out.write("<BODY>\n");
out.write("La suma es ");
out.print(calc.suma(num1,num2));
out.write("\n");
out.write("</BODY>\n");
out.write("</HTML>\n");
```

- Veamos un fragmento del método **`_jspservice`**

2.7. Más aspectos de programación de clases

- 2.7.1. Acceso a miembros de la misma clase.
- 2.7.2. La programación O-O en la práctica.
- 2.7.3. Javabeans.
- 2.7.4. Interfaz e implementación.
- 2.7.5. Usar otras clases del JDK

2.7. Más aspectos de programación de clases

- 2.7.1. Acceso a miembros de la misma clase.
- 2.7.2. La programación O-O en la práctica.
- 2.7.3. Javabeans.
- 2.7.4. Interfaz e implementación.
- 2.7.5. Usar otras clases del JDK

Accediendo a los miembros de la misma clase

- Sabemos que la forma de acceder a los miembros de una clase es con la notación del punto:

objeto.nombreatributo
objeto.nombremétodo(valorespar)

- Pero frecuentemente, desde un método de una clase, queremos acceder a los miembros de esta misma clase. Esto permite que los objetos de esa clase puedan acceder a sus propios miembros.

Accediendo a los miembros de la misma clase

- En este caso, puede prescindirse del punto y de la referencia a objeto

```
nombreatributo
nombremétodo(valorespar)
```

- Otra opción es usar la palabra reservada **this**

```
this.nombreatributo
this.nombremétodo(valorespar)
```

Ejemplo

- Completar la clase **Libreta** para que tenga un método que nos diga si dos libretas son iguales. Concretamente, si la libreta actual es igual a otra.

Solución

```
package com.aurumsol.cursojava.tema2.ejemplos;
public class Libreta {
  private int numero, saldo;
  public void iniciar(int nLibreta){
     numero = nLibreta;
     saldo = 0;
  }
  public int obtenerNumero(){
     return numero;
  }
  public int obtenerSaldo(){
     return saldo;
  }
```

Solución

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

Ejemplo

```
public boolean iguales (Libreta otra){
 return (this.obtenerNumero()==
   otra.obtenerNumero())
}
}
```

¿Cómo se utiliza `iguales`?

- Si tenemos dos libretas **l1,l2**, para saber si son iguales

```
l1.iguales(l2)
```

- O también

```
l2.iguales(l1)
```

2.7. Más aspectos de programación de clases

- 2.7.1. Acceso a miembros de la misma clase.
- 2.7.2. La programación O-O en la práctica.
- 2.7.3. Javabeans.
- 2.7.4. Interfaz e implementación.
- 2.7.5. Usar otras clases del JDK

Ejercicio

- Crear una clase que modele un banco. Para simplificar, supondremos que un banco tiene 3 libretas como máximo.
- Las operaciones que nos interesan de un banco son añadir una nueva libreta, ingresar en una libreta (dado su número), retirar de una libreta, obtener el saldo de una libreta, obtener el número de libretas.
- Deben usar la clase de libreta que han programado anteriormente.
- Consideren todos los errores que se puedan dar e intenten que sea el formulario el que los genere.

Solución (1)

```
package com.aurumsol.cursojava.tema2.ejemplos;
public class Banco {
  private Libreta cuenta1, cuenta2, cuenta3;
  public boolean agregarLibreta(Libreta nLibreta){
    if (cuenta1==null){
     cuenta1 = nLibreta;
     return true;
    } else if (cuenta2==null){
     cuenta2 = nLibreta;
     return true;
    } else if (cuenta3==null){
     cuenta3 = nLibreta
     return true;
    } else {
     return false;
  }
```

Solución (2)

```
public Libreta obtenerLibreta(int numero){
    if ((cuenta1 != null) &&
       (cuenta1.obtenerNumero()==numero)){
        return cuenta1;
    } else if ((cuenta2 != null) &&
       (cuenta2.obtenerNumero()==numero)){
        return cuenta2;
    } else if ((cuenta3 !=null) &&
       (cuenta3.obtenerNumero()==numero)){
        return cuenta3;
    } else {
        return null;
    }
}
```

Solución (3)

```
public int numLibretas(){
    int numeroLib = 0;
    if (cuenta1 != null) {
        numeroLib++;
    } else if (cuenta2 != null){
        numeroLib++;
    } else if (cuenta3 !=null) {
        numeroLib++;
    }
    return numeroLib;
}
}
```

Ejemplo de uso de la clase `Banco` (1)

- Añadimos una nueva libreta con código 123 al banco que está en la variable **`banco1`**.

```
Libreta nuevaCuenta = new Libreta()
nuevaCuenta.iniciar(123);
banco1.agregarLibreta(nuevaCuenta)
```

- Como vemos utilizamos la clase **`Libreta`** para crear la libreta antes de agregarla al Banco.

Ejemplo de uso de la clase `Banco` (2)

```
public class NegocioBanco {
  public boolean retirar(int nLibreta, int
  monto){
   Libreta cuenta =
     banco1.obtenerLibreta(nLibreta);
   if (cuenta==null){
      return false; //No existe la libreta
   } else {
      boolean correcto = cuenta.retirar(monto);
      return correcto;
   }

}
```

- Devuelve **`true`** si el retiro no se efectuó y **`false`** en caso contrario.

Como se ve en estos ejemplos

- Para hacer una determinada tarea, colaboran objetos de la clase **`Banco`** y objetos de la clase **`Libreta`**.
- La clase **`Banco`** no tiene operaciones para retirar, depositar o crear libretas ya que esto corresponde a la clase **`Libreta`**.
- De esta manera, evitamos la repetición de código, lo que es bueno para el mantenimiento.
- Esta es la forma orientada a objetos de programar: cada clase tiene sus responsabilidades y no se mezcla con las otras.

2.7. Más aspectos de programación de clases

- 2.7.1. Acceso a miembros de la misma clase.
- 2.7.2. La programación O-O en la práctica.
- 2.7.3. Javabeans.
- 2.7.4. Interfaz e implementación.
- 2.7.5. Usar otras clases del JDK

Supongamos ahora que queremos hacer una clase que modele un círculo

- No nos interesa la representación gráfica del círculo, sino sus datos.

Posible solución

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  public float radio;
  public void asignarRadio(float
  nuevoRadio){
     radio = nuevoRadio;
  }
  public float consultarRadio(){
     return this.radio;
  }
}
```

Supongamos que tenemos un objeto `circulo1` de esta clase

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  public float radio;
  public void asignarRadio(float nuevoRadio){
     radio = nuevoRadio;
  }
  public float consultarRadio(){
     return this.radio;
  }
}
```

Da el mismo resultado poner

```
circulo1.radio
circulo1.consultarRadio()
```

Supongamos que tenemos un objeto `circulo1` de esta clase

```
package com.aurumsol.cursoJava.tema2.beans;
public class Circulo {
  public float radio;
  void asignarRadio(float nuevoRadio){
     radio = nuevoRadio;
  }
  float consultarRadio(){
     return this.radio;
  }
}
```

Da el mismo resultado poner

```
circulo1.radio = expresion
circulo1.asignarRadio(expresion)
```

Podríamos prohibir el acceso a atributos desde fuera de la clase

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void asignarRadio(float nuevoRadio){
     radio = nuevoRadio;
  }
  public float consultarRadio(){
     return this.radio;
  }}
```

Ahora sólo se puede acceder al radio a través de métodos

```
circulo1.consultarRadio
circulo1.asignarRadio(expresion)
```

Podríamos prohibir el acceso a atributos desde fuera de la clase

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void asignarRadio(int nuevoRadio){
     radio = nuevoRadio;
  }
  public float consultarRadio(){
     return this.radio;
  }}
```

no se puede acceder a los atributos desde fuera

```
circulo1.radio
circulo1.radio = expresion
```

¡¡PROHIBIDO!!

Se dice que la clase tiene el estado encapsulado

- La ventaja es que podemos implementar controles a la modificación de los datos.
- Por ejemplo, se puede comprobar que no se entre ningún radio negativo.
- De la otra manera, otra clase podría asignarnos un valor negativo y la clase no se podría proteger.

Por ejemplo

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void asignarRadio(int nuevoRadio){
     if (radio>=0){
          radio = nuevoRadio;
     }
  }
  public float consultarRadio(){
     return this.radio;
  }
}
```

Cuando el estado está encapsulado, suele haber 2 métodos por atributo

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void asignarRadio(int nuevoRadio){
     radio = nuevoRadio;
  }
  public float consultarRadio(){
     return this.radio;
  }
}
```

Un método para consultar el atributo y otro para modificarlo

Cuando estado está encapsulado, hay 2 métodos por cada atributo

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void asignarRadio(int nuevoRadio){
     radio = nuevoRadio;
  }
  public float consultarRadio(){
     return this.radio;
  }
}
```

En este caso, los métodos son `asignarRadio` y `consultarRadio`.

Cuando estado está encapsulado, hay 2 métodos por cada atributo

- Uno permite recuperar a un atributo privado. Se le llama **accessor.**

```
public float consultarRadio(){
   return this.radio;
}
```

- Otro permite modificar el atributo privado. Se le llama **mutator.**

```
public void asignarRadio(int nuevoRadio){
   radio = nuevoRadio;
}
```

Por convención, se nombra los dos métodos como `get` y `set`

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void setRadio(int nuevoRadio){
     radio = nuevoRadio;
  }
  public float getRadio(){
     return this.radio;
  }
}
```

En este caso, los métodos serían `setRadio` y `getRadio`. Seguir la convención.

Si una clase

- tiene el estado encapsulado
- sus métodos accesores y mutadores comienzan con **`get`** y **`set`**.

SE LE LLAMA

- un **Javabean**

Javabean es una clase que cumple estas dos condiciones

- Tiene todos sus atributos privados.
- A sus atributos sólo se puede acceder a través de métodos que se comienzan con **`get`** y **`set`**.

Un Javabean

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void setRadio(int nuevoRadio){
     radio = nuevoRadio;
  }
  public float getRadio(){
     return this.radio;
  }
}
```


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

1 Javabean

- Los atributos son privados y sólo pueden accederse mediante métodos **get** y **set**

1 Javabean

- Los atributos son privados y sólo pueden accederse mediante métodos **get** y **set**

Un Javabean

```
package com.aurumsol.cursojava.tema2.beans;
public class Circulo {
  private float radio;
  public void setRadio(int nuevoRadio){
     if (nuevoRadio >= 0){
          radio = nuevoRadio;
     }
  }
  public float getRadio(){
  }
}
```

Controlándolo con **get** y **set** hacemos que los atributos se accedan de un modo seguro, en vez de que todo el mundo los manosee.

1 Javabean

- Controlándolo con **get** y **set** hacemos que los atributos se accedan de un modo seguro, en vez que todo el mundo los manosee.

Conclusión

- Es una buena práctica hacer los atributos privados y acceder mediante **get** y **set**.
- Es decir, el uso de Javabeans se recomienda.
- Hay muchos productos que obligan usar Javabeans: Hibernate, Spring, Kodo.

2.7. Más aspectos de programación de clases

- 2.7.1. Acceso a miembros de la misma clase.
- 2.7.2. La programación O-O en la práctica.
- 2.7.3. Javabeans.
- 2.7.4. Interfaz e implementación.
- 2.7.5. Usar otras clases del JDK

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

Interfaz e implementación

- Interfaz: Todo lo que puede accederse desde fuera del objeto.
- Implementación: todo aquello que NO puede accederse desde fuera del objeto.

Interfaz e implementación

- Interfaz: Atributos públicos y métodos públicos.
- Implementación: Atributos privados, métodos privados y el cuerpo de los métodos.

Ocultación de la implementación

- Utilizamos los objetos sin saber cómo están implementados (sabemos QUÉ hacen, pero no el CÓMO)
- Esto simplifica la programación.
- Se puede mejorar la implementación sin cambiar la interfaz del objeto: *refactoring*

2.7. Más aspectos de programación de clases

- 2.7.1. Acceso a miembros de la misma clase.
- 2.7.2. La programación O-O en la práctica.
- 2.7.3. Javabeans.
- 2.7.4. Interfaz e implementación.
- 2.7.5. Usar otras clases del JDK

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.

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

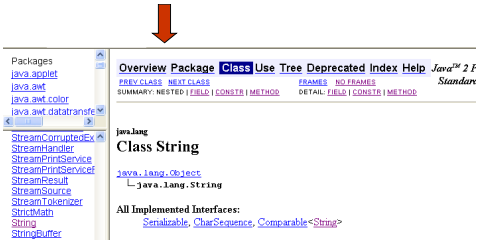
La pantalla aparece partida en tres partes.

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.



Sobre el profesor

Dr. Vicent-Ramon Palasí Lallana.

- **Consultas y dudas a:**
  Teléfono: 275-4254.
  Fax: 263-3948.
  E-mail: vpalasi@aurumsol.com
  vpalasi@ufg.edu.sv
  Web: www.aurumsol.com

2.7. Excepciones

- 2.7.1. Concepto de excepción.
- 2.7.2. Tratar una excepción en una JSP.
- 2.7.3. Tratar una excepción en una clase.
- 2.7.4. Tipos de excepciones.
- 2.7.5. Crear excepciones propias.

2.7. Excepciones

- 2.7.1. Concepto de excepción.
- 2.7.2. Tratar una excepción en una JSP.
- 2.7.3. Tratar una excepción en una clase.
- 2.7.4. Tipos de excepciones.
- 2.7.5. Crear excepciones propias.

Excepciones

- Son errores que se producen en la ejecución del programa.
- Normalmente, se interrumpe la ejecución del programa dando un mensaje de error.

Supongamos una JSP que calcula el cociente entre dos números

```
<% int num1 = Integer.parseInt(
  request.getParameter("num1"));
  int num2 = Integer.parseInt(
  request.getParameter("num2"));
  int coc= num1/num2;%>
<HTML><HEAD><TITLE>Cociente</TITLE><BODY>
El cociente de la divisi&oacute;n es
  <%=coc%> </HTML>
```

¿Qué pasa si ...?

- ¿Qué pasa si introducimos 4 y 0 como parámetros en el formulario?
- Despliegen y vean

Aparece un mensaje de error

excepción

```
org.apache.jasper.JasperException: / by zero
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:358)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

causa raíz

```
java.lang.ArithmeticException: / by zero
        org.apache.jsp.cociente_jsp._jspService(cociente_jsp.java:47)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:133)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:311)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

- Vean sólo el apartado "causa raíz".

El mensaje de error (apartado "causa raíz")

Lo que pone es:

```
java.lang.ArithmeticException: / by zero.
org.apache.jsp.cociente_jsp._jspService
(cociente_jsp.java:47)
```

causa raíz

```
java.lang.ArithmeticException: / by zero
        org.apache.jsp.cociente_jsp._jspService(cociente_jsp.java:47)
```

La aplicación Web "aborta" con una excepción de dividir por cero

```
java.lang.ArithmeticException: / by zero.
org.apache.jsp.cociente_jsp._jspService
(cociente_jsp.java:47)
```

El mensaje de la excepción indica:
- El tipo de excepción.
- En qué método, archivo y línea se produce.

Excepciones

- Son errores que se producen en la ejecución de un programa.
- Si no hacemos nada, interrumpen la ejecución del programa dando un mensaje de error.
- Pero es mejor que el programador las intercepte y les de un tratamiento adecuado

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

Fijémonos en `java.lang.ArithmeticException`

```
java.lang.ArithmeticException: / by zero.
org.apache.jsp.cociente_jsp._jspService
(cociente_jsp.java:47)
```

Las excepciones son objetos y tienen clases como todo en Java.

- **`ArithmeticException`** es la clase de excepción.
- **`java.lang`** es el paquete.

`java.lang`: Un paquete especial.

- Sabíamos que para usar unas clases de un paquete debíamos hacer un **`page import`**.
- Pero **`java.lang`** es un paquete especial. Es el paquete estándar de Java.
- Las clases de java.lang están disponibles para cualquier programa Java sin necesidad de importarlas.

`java.lang.Exception`

- Todas las excepciones son instancias de la clase **`Exception`**.
- **`Exception`** es la clase que engloba todas las excepciones: es la clase general sobre excepciones.

¿Qué pasa cuando se produce una excepción?

- Depende de cómo lo hayamos programado, pueden suceder dos cosas:
- Si no hemos programado nada, la máquina virtual y Tomcat enseñan un mensaje de error por pantalla.
- Pero podemos decir al programa que queremos tratar una excepción.

Tratar una excepción

- Es hacer algo adecuado con ella en vez de dejar que la máquina virtual y Tomcat acaben con un mensaje de error.
- Otro ejemplo: si queremos leer un archivo y este no se encuentra en disco se producirá una excepción **`FileNotFoundException`**.
- En vez de dejar que la máquina virtual aborte el programa con un mensaje críptico, podemos proveer un mensaje de error en español e indicar al usuario las acciones a seguir.

2.7. Excepciones

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

Tratando excepciones en una JSP. Se usa la estructura.

```
try {
//Código que puede producir excepción
} catch (ClaseExcepcion1 e1) {
  //Tratar exc. de ClaseExcepcion1
} catch (ClaseExcepcion2 e2) {
  //Tratar exc. de ClaseExcepcion2
...
} catch (ClaseExcepcionN en) {
  //Tratar exc. de ClaseExcepcionN
} finally {
  //Código que se ejecuta tanto si se
  produce una excepción como si no.
}
```

Notas sobre la estructura

- La cláusula **`finally`** es opcional y suele realizar alguna función "de limpieza" (por ejemplo, cerrar archivos), la cual debe ejecutarse tanto si se produce una excepción como si no.
- Las sentencias **`catch`** son como los condicionales anidados. La primera condición que se cumple es la que se ejecuta.
- Por ello, debemos diseñar las clases de excepción de más generales a más específicas.
- La última debe ser la más general, es decir, la clase **`Exception`**.

Solución (resumida)

```
<% int num1 = ...;  int num2 = ...;
  try{
    int coc= num1/num2;
  }catch (ArithmeticException p) {
    out.print("No se puede dividir"+
          "por cero");
  }catch (Exception e) {
    out.print("Algo ha fallado");
  }%>
<HTML> ...
```

Hubiera sido mejor idea y mejor diseño

```
<% int num1 = ...;  int num2 = ...;
  if (num2 =0){
    out.print("No se puede dividir"+
          "por cero");
  } else {
    int coc= num1/num2;
  }
%>
<HTML> ...
```

Moraleja

- Hay algunos casos en que se puede saber cuando se programa cuando se producirá una excepción.
- En este caso, es mejor usar un **`if`** que un **`try-catch`**.
- El **`try-catch`** es más lento y poco legible.
- De todas maneras, hay casos (no tan simples como éste) que no se puede saber si fallará hasta que se ejecute. Para ellos, es apropiado el **`try-catch`**.

2.7. Excepciones

Ahora bien.

- La JSP que hemos visto no sigue la arquitectura en n-capas.
- Debemos poner la presentación en la JSP y el cálculo en la clase Java.
- Es así cómo lo haremos.

Ejemplo: la clase `Calculadora`

```
package com.aurumsol.cursojava.tema2.excepciones;
public class Calculadora {
 public int cociente(int n1, int n2){
    return n1/n2;
 }
}
```

Tiene un método que calcula el cociente entero. La JSP siguiente la utiliza

La JSP llama a la clase anterior

```
<%@ page import="com.aurumsol.cursojava.
tema2.*" %>
<% int num1 = Integer.parseInt(
  request.getParameter("num1"));
  int num2 = Integer.parseInt(
  request.getParameter("num2"));
  Calculadora calc = new Calculadora();
  int coc= calc.cociente(num1,num2);%>
<HTML><HEAD><TITLE>Cociente</TITLE><BODY>
El cociente de la divisi&oacute;n es
  <%=coc%> </HTML>
```

¿Qué pasa si ...?

- ¿Qué pasa si introducimos 4 y 0 como parámetros en el formulario?
- Despliegen y vean

Se produce una excepción y, como no se trata, un mensaje de error

excepción

```
org.apache.jasper.JasperException: / by zero
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:358)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

causa raíz

```
java.lang.ArithmeticException: / by zero
        com.aurumsol.cursojava.tema2.Aritmetica.cociente(Aritmetica.java:5)
        org.apache.jsp.cociente_jsp._jspService(cociente_jsp.java:48)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:133)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:311)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

- Vean sólo el apartado “causa raíz”.

Ahora el mensaje de error indica que la excepción se produce en la clase

Lo que pone es:

```
java.lang.ArithmeticException: / by zero.
com.aurumsol.cursojava.tema2.Calculadora.cociente
(Calculadora.java:5)
```

causa raíz

```
java.lang.ArithmeticException: / by zero
        com.aurumsol.cursojava.tema2.Aritmetica.cociente(Aritmetica.java:5)
        org.apache.jsp.cociente_jsp._jspService(cociente_jsp.java:48)
```

Queremos tratar la excepción. ¿Cómo lo hacemos?

- Hay dos posibilidades:
- Primera: Tratarla con **`try-catch`**.
- Segunda: Propagarla con **`throws`**.

Queremos tratar la excepción. ¿Cómo lo hacemos?

- Hay dos posibilidades:
- Primera: Tratarla con **`try-catch`**.
- Segunda: Propagarla con **`throws`**.

Primera: tratarla con `try-catch`

```
package com.aurumsol.cursojava.tema2.excepciones;
public class Calculadora {
 public int cociente(int n1, int n2){
    try{
       return n1/n2
    }catch (ArithmeticException p) {
        out.print("No se puede dividir"+
          "por cero");
    }%>
}
```

CORRECTO, ¿NO?

¡¡¡¡NO, NO Y NO!!!!

- No podemos poner un **`out.print`** en una clase de negocio.
- Primero: No funciona.
- Segundo: Si funcionara, el mensaje saldría en el servidor (y no en el navegador del cliente).
- Tercero: Rompe el esquema de la arquitectura de n-capas.

Las clases de negocio sólo contienen cálculo

- No contienen operaciones de E/S.
- De esto se encargan las clases de presentación.
- ¿Qué hacemos, pues?

Supongamos que los números nunca son negativos.

```
package com.aurumsol.cursojava.tema2.excepciones;
public class Calculadora {
 public int cociente(int n1, int n2){
    try{
       return n1/n2
    }catch (ArithmeticException p) {
        return -1;
    }
    %>
}
```

Podemos retornar un –1 para indicar que se ha producido la excepción

La JSP escribirá el mensaje de error

```
<%@ page import=...
<% int num1 = ...;int num2 = ...
  Calculadora calc = new Calculadora();
  int coc= calc.cociente(num1,num2);%>
<HTML><HEAD><TITLE>Cociente</TITLE><BODY>
<% if (coc = -1){
      out.print("No se puede dividir"+
          "por cero");
   } else {
      out.print (El cociente es);
      out.print(coc);
   }%>
</HTML>
```

Como ven, siempre es la capa de presentación

- La que tiene que manejar la E/S.
- La capa de negocio sólo puede hacer cálculos.

Queremos tratar la excepción. ¿Cómo lo hacemos?

- Hay dos posibilidades:
- Primera: Tratarla con **`try-catch`**.
- Segunda: Propagarla con **`throws`**.

Propagar una excepción ("pasar la pelota")

- Un método propaga una excepción cuando
- No la trata el método donde se produce.
- El método pasa la responsabilidad de tratarla al código que lo llamó (a veces, a este código se le llama "código cliente").

Ejemplo: la clase `Calculadora`

```
package com.aurumsol.cursojava.tema2.excepciones;
public class Calculadora {
 public int cociente(int n1, int n2){
    return n1/n2;
 }
}
```

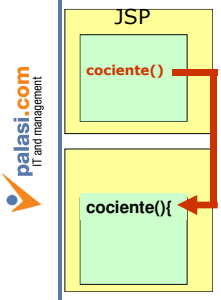
Esta clase la llama la JSP

La JSP llama a la clase anterior. La JSP es el "código cliente"

```
<%@ page import="com.aurumsol.cursojava.
tema2.*" %>
<% int num1 = Integer.parseInt(
  request.getParameter("num1"));
  int num2 = Integer.parseInt(
  request.getParameter("num2"));
  Calculadora calc = new Calculadora();
  int coc= calc.cociente(num1,num2);%>
<HTML><HEAD><TITLE>Cociente</TITLE><BODY>
El cociente de la divisi&oacute;n es
  <%=coc%> </HTML>
```


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

La JSP llama al método cociente



- La JSP es el código cliente.

Propagar la excepción significa pasarle la responsabilidad al código cliente

- A la JSP. Esto se haría de la siguiente manera:

```
package com.aurumsol.cursojava.tema2.excepciones;
public class Calculadora {
 public int cociente(int n1, int n2)
   throws ArithmeticException{
     return n1/n2;
 }
}
```

- Se pone **`throws`** ***ClaseExcepcion***

Propagar la excepción significa pasarle la responsabilidad al código cliente

- Si se produce la excepción, se pasará a la JSP. "Se le pasará la pelota".

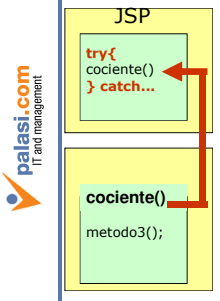
```
package com.aurumsol.cursojava.tema2.excepciones;
public class Calculadora {
 public int cociente(int n1, int n2)
   throws ArithmeticException{
     return n1/n2;
 }
}
```

- Al código que la llamó.

Ahora quien tiene que tratar la excepción es la JSP

```
<%@ page import=...
<% int num1 = ...;  int num2 = ...
  Calculadora calc = new Calculadora();
  try{
    int coc= calc.cociente(num1,num2)
  } catch (ArithmeticException ex){
    out.print("No se puede dividir"+
          "por cero");
  };%>
<HTML>...
```

Propagar la excepción

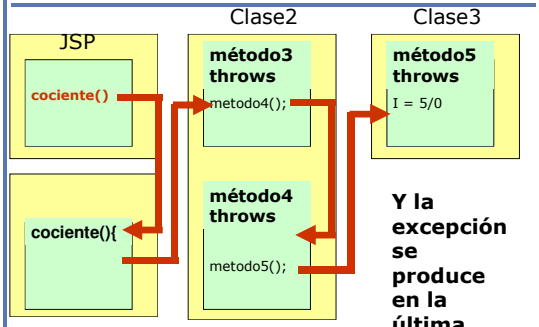


- Se propaga al código cliente, a la JSP.
- La JSP tiene que tratar la excepción. (Si no, Tomcat y la MV se encargan de recibir el tratamiento estándar).

Queremos tratar la excepción. ¿Cómo lo hacemos?

- Si se produce en una JSP, con **`try-catch`**.
- Si se produce en una clase, hay dos posibilidades:
  - Tratarla con **`try-catch`**.
  - Propagarla con **`throws`**.

¿Qué pasa si un método llama a otro y este a otro?

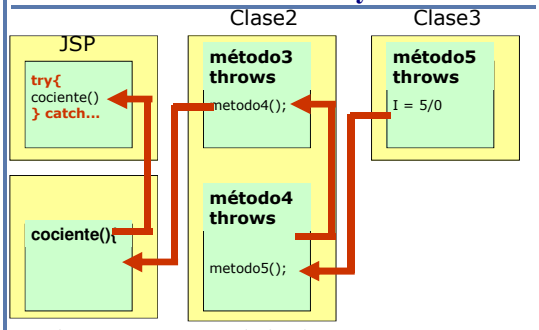


Y la excepción se produce en la última.

El arte de pasar la pelota

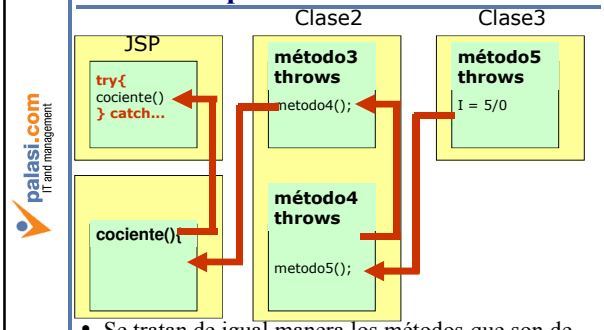
- El método donde se produce la excepción
  - O la trata (con try-catch) o la propaga (con throws).
- Si la propaga, el método que ha llamado al anterior.
  - O la trata o la propaga.
- Si la propaga, el método que ha llamado al anterior.
  - O la trata o la propaga.
- Y así sucesivamente. Si se ha propagado hasta la JSP.
  - O la JSP la trata (con try-catch) o la MV lo hará.

Imaginemos que se propaga hasta la JSP y ella lo trata



- El esquema sería el siguiente.

Para la propagación de error no importa la estructura de clases



- Se tratan de igual manera los métodos que son de la misma clase como los de otra.

2.7. Excepciones

Dos tipos de excepciones en Java

- Comprobadas ("Checked"). El compilador obliga a poner tratamiento de errores para estas excepciones.
- No comprobadas ("Unchecked" o "Runtime"). El compilador no te obliga a poner tratamiento de errores para las mismas.

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

La gran polémica

- Hay una gran polémica sobre si Java debe tener excepciones comprobadas. Otros lenguajes que copian a Java no las tienen.
- Esta es quizás la polémica más encarnizada en el mundo Java.
- Por razones que sería muy largo de explicar, mi opinión es que las excepciones comprobadas son la mayoría de las veces una mala idea, pues obligan a cambiar el diseño de todo el programa por excepciones que no son recuperables.

Mi consejo es

- Cada vez que encuentren una excepción comprobada, conviértanla en una no comprobada.
- Esto se hace con el código

```
try{
   Todo el código del método
}catch (ClaseExcepcion ex){
throw new RuntimeException(ex.getMessage(),
  ex);
}
```

Comprendamos el código

- Comprendiendo el código

```
try{
   Todo el código del método
}catch (ClaseExcepcion ex){
throw new RuntimeException(ex.getMessage(),
  ex);
}
```

- Lo que hacemos es capturar la excepción y lanzar otra que es no comprobada.

Comprendamos el código

```
new RuntimeException(ex.getMessage(), ex);
```

crea una excepción de tipo **RuntimeException** (este tipo es de excepciones no comprobadas), a la que se le pasa como datos la excepción original y el mensaje original.

- Esto lo veremos con más detalle cuando veamos constructores.
- Esta excepción es lanzada con **throws**

2.7. Excepciones

Crear nuestras propias excepciones

- Útil si queremos que ciertos errores específicos de nuestra aplicación se traten con el código de tratamiento de excepciones.
- Creamos una clase derivándola de **Exception**, con un código parecido al siguiente:

```
public class ImporteNegativoException
 extends Exception{
  public ImporteNegativoException
     (String txto){
     super (txto)
  }
}
```


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

Así podré usar la cláusula `throw` para lanzar la excepción

```
public double calculaIva(double importe)
  throws ImporteNegativoException{
  if (importe>=0) {
     return importe*1.13;
  } else{
     throw new ImporteNegativoException
      ("Importe negativo");
  }
}
```

La excepción nueva se propagará y se tratará con el código correspondiente (por ejemplo, en la JSP)

Pero hay que ir con cuidado al definir nuevas excepciones. En este caso, más adecuado no hacerlo.

El código de excepciones es relativamente lento

- Por ello, sólo se deben lanzar excepciones propias cuando sea estrictamente necesario.
- En caso contrario, deben manejarse, p. ej., devolviendo un código especial que será tratado con la JSP.

```
public double calculaIva(double importe){
  if (importe>=0) {
     return importe*1.13;
  } else{
     return -1.0;
  }
}
```

Notas importantes sobre excepciones (1)

- 1.Las variables que se definen dentro de un bloque **`try...catch`** sólo están definidas dentro dentro del **`try...catch`**.
- De la misma manera, las variables que se inicializan dentro del bloque **`try..catch`** no están inicializadas fuera.
- Dará error de compilación si intento acceder desde fuera del **`try...catch`** a estas variables.
- Si necesito que la variable esté disponible fuera del **`try ..catch`**, la defino e inicializo fuera de dicho bloque.

Notas importantes sobre excepciones (2)

- 2. Todas las excepciones tienen los métodos de instancia **`getMessage`** y **`printStrackTrace`**.
- El primero nos sirve respectivamente para obtener el mensaje de error que se generó (si es una excepción propia, lo generamos nosotros, si no la máquina virtual de Java).
- El segundo no sirve para imprimir por pantalla la pila de llamadas a métodos que nos indican en qué línea de código de cada método se genera la excepción.
- Los dos métodos son de suma importancia para el tratamiento de errores y la depuración.

Notas importantes sobre excepciones (3)

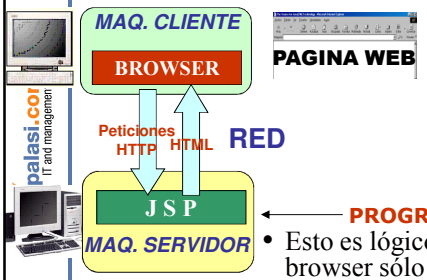
- 3. Mientras depuramos un programa es útil desactivar el tratamiento de excepciones, pues así vemos los mensajes de error tal como se producen.
- Pero en la versión final de producción que se entrega al cliente deben activarse las excepciones.

Algunas excepciones comunes del paquete *`java.lang`*

- **`ClassNotFoundException`** (“checked”). Se produce cuando queremos utilizar una clase en nuestro programa y ésta no se encuentra.
- **`NullPointerException`** (“unchecked”). Cuando intentamos ejecutar un método o acceder a un atributo de una variable y la variable tiene el valor **`null`**.
- **`ArithmeticException`**. (“unchecked”) Cuando se realiza alguna operación aritmética ilegal (dividir por cero, raíz cuadrada de un número negativo).
- **`ArrayIndexOutOfBoundsException`** (“unchecked”) Cuando intentamos acceder a un array en posiciones que están más allá del límite del array.

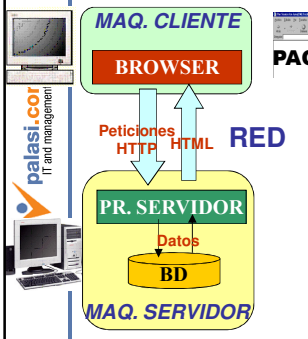
Política de excepciones: Generación, propagación y tratamiento

Es decir, lo que hace la JSP es generar una página HTML (dinámicamente)



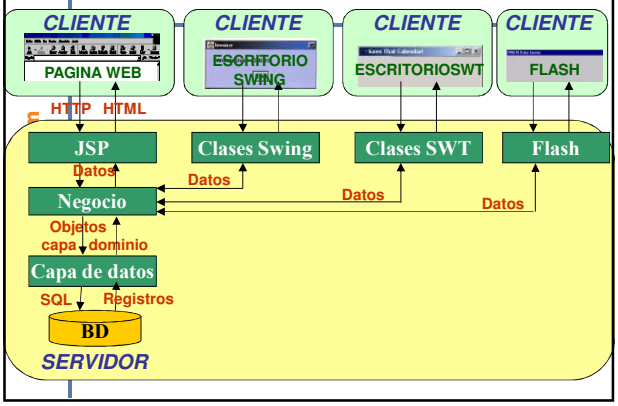
- Esto es lógico, pues el browser sólo acepta HTML. A partir del HTML, el browser desplegará una página

El servidor recibe peticiones HTTP y genera HTML



- El cliente es el navegador de Internet ("browser"). El servidor recibe peticiones HTTP y devuelve HTML que después el browser traduce en una página Web, que es la que ve el cliente.

Dibuix



Dibuix2

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