



DESENVOLUPAMENT WEB ENTORN SERVIDOR

2019/20

Tema 1: Programació orientada a objectes.....	5
Què és la POO?	6
Especificació formal UML: Diagrama de classes	11
Cicle de vida dels objectes	30
Encapsulació	36
Herència (modificador d'accés i composició)	41
Polimorfisme	44
Sobrescritura de hashCode, equals i toString.....	49
Ús de static (variables, mètodes i classes)	50
Objectes immutables	52
Classes i mètodes abstractes	53
Ús de final.....	55
Classes internes, locals i anònimes	56
Enumeracions	58
Classes genèriques.....	59
Interfícies.....	63
Classes abstractes	66
Interfaces Comparable i Comparator	67
Interfícies Funcionals	69

Tema 2: Patrons i Arquitectura Client-Servidor.....	73
GoF.....	74
Singleton	77
Prototype	80
IoC i DI	86
MVC (Model-Vista-Controlador).....	93
DAO	94
Antipatterns.....	96
Java EE	97
Requisits per emprar Java EE.....	98
Servlets	107
JSP	112
JSTL (JSP Standard Tag Library)	116
JavaBeans.....	117
PHP	118
Tema 3: Frameworks	119
Spring 5: Spring Core i Spring MVC.....	119
Tècniques d'accés a dades	119

Tema 5: Servers Stateless: OAuth i Serveis WEB (SOAP i API REST).....	198
Bibliografia.....	199

TEMA 1: PROGRAMACIÓ ORIENTADA A OBJECTES

QUÈ ÉS LA POO?

L'orientació a objectes no és un tipus de llenguatge de programació. És una **metodologia de treball** per crear programes. Permet dissenyar l'aplicació de manera **totalment independent** del llenguatge de programació que s'utilitzarà. Un programa s'entén com una simulació d'un escenari del món real, en què un conjunt d'elements, els **objectes**, **interactuen entre ells** per dur a terme la tasca que es vol resoldre.

Bases de la POO

1. Tot és un objecte, amb una identitat pròpia.
2. Un programa és un conjunt d'objectes que interactuen entre ells.
3. Un objecte pot estar format per altres objectes més simples.
4. Cada objecte pertany a un tipus concret: una classe.
5. Objectes del mateix tipus tenen un comportament idèntic.

Tot és un objecte, amb una identitat pròpia.

La POO es basa en crear la simulació d'un escenari que podríem tenir en el **món real**, però **dins de l'ordinador**. És una interacció d'un conjunt d'elements, cadascun del quals té unes propietats i un comportament concret que intenten imitar les de l'element del món real. Amb les propietats d'un objecte ens referim a les qualitats que es consideren importants quantificar i que defineixen l'aspecte o l'estat **de l'objecte**. S'anomenen formalment els seus atributs.

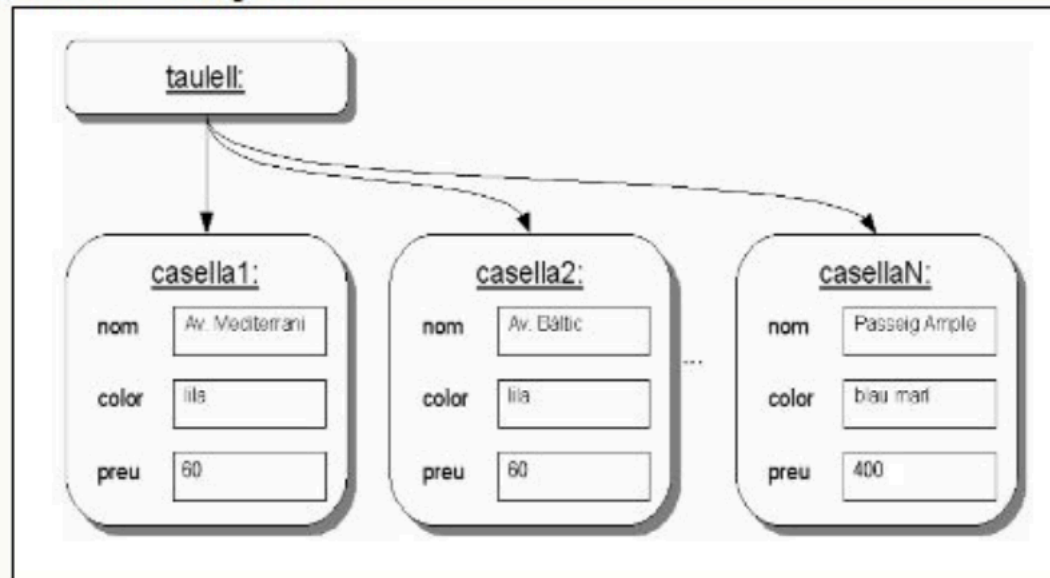
Un objecte es pot compondre d'altres objectes més simples

Quan es vol resoldre un problema, la manera més senzilla de tractar amb la complexitat és mitjançant la **descomposició en problemes més simples**.

El motor d'un cotxe es descompon en elements més senzills i molt més fàcils d'analitzar, els quals, a la vegada, es poden descompondre en un altre conjunt d'elements encara més simples. Per exemple: El moviment del motor depèn del cigonyal, les valvules, els pistons, ... i la seva interacció.

Això ens permet: poder generar elements complexos a partir d'altres de més simples, i tenir la possibilitat d'interactuar directament tant amb l'objecte complex com amb qualsevol dels seus subelements → **Afavoreix la reutilització de codi**.

Figura 1.2. Un objecte es pot compondre d'altres objectes



Cada objecte pertany a un tipus concret: una classe.

Hi ha objectes que es poden considerar del mateix tipus: tenen exactament **les mateixes propietats**, tot i que el **valor** de cada una pot ser **diferent** per a cada objecte concret → aquests objectes pertanyen a **la mateixa classe**.

Una classe és l'especificació formal de les propietats (els atributs) i el comportament esperat (la llista d'operacions) d'un conjunt d'objectes del mateix tipus, i que actua com una plantilla per generar cadascun d'ells.

Es diu que un objecte és una instància d'una classe i que un objecte és instanciat quan es crea dins l'aplicació.

Els objectes del mateix tipus tenen un comportament idèntic

De la mateixa manera que una classe defineix els atributs d'objectes del mateix tipus, també n'especifica el comportament: la seva llista d'operacions. Per cada interacció que un objecte de la classe A pot rebre (sense que importi el possible objecte origen), caldrà definir una operació associada a aquesta en especificar la classe A.

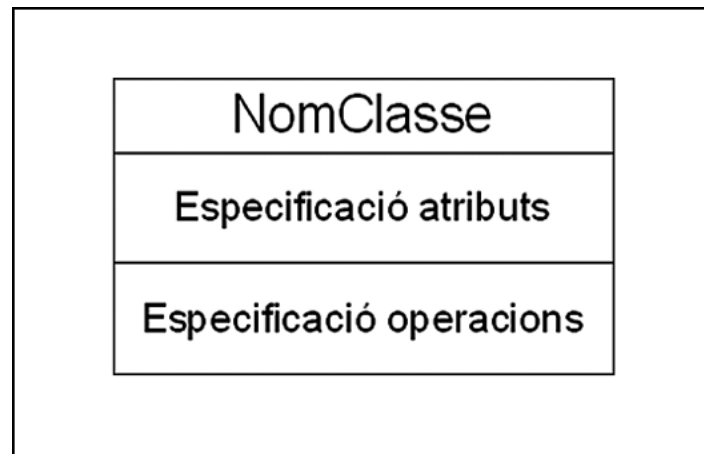
Pases a seguir per solucionar un problema amb POO

1. Plantejar l'escenari descriptivament, amb **llenguatge humà**.
2. **Localitzar**, dins la descripció de l'escenari, els **elements** que es consideren **més importants**: els que realment interactuen amb vista a resoldre el problema. Aquests seran els objectes del programa.

3. Considerar quins elements són d'una certa **complexitat**. Redefinir-los com a agrupacions d'objectes més simples. **Descomposició**.
4. Agrupar els diferents objectes segons el tipus: quins objectes veiem que tenen propietats o comportaments idèntics.
5. Identificar i enumerar les característiques dels objectes de cada classe.
6. Establir les **relacions** que hi ha **entre els objectes** de les diferents classes a partir del paper que interpreten en el problema general.
7. La descomposició inicial d'un problema mitjançant l'orientació a objectes no ha d'explicitar mai la interfície d'usuari a emprar. Només es defineix la lògica interna del sistema i com s'estructura la informació a processar. Això és el que s'anomena el model de l'aplicació.

ESPECIFICACIÓ FORMAL UML: DIAGRAMA DE CLASSES

L'UML és un llenguatge estàndard que permet especificar amb notació gràfica programari orientat a objectes.



En UML, una classe es representa en format complet mitjançant una caixa dividida horitzontalment en tres parts. La part superior compleix exactament la mateixa funció i té el mateix format que en el format simplificat, i s'estableix el nom de la classe. En la part del mig es defineixen els atributs que tindran les seves instàncies. Finalment, en la part inferior, es defineixen les operacions que es poden cridar sobre qualsevol de les seves instàncies. L'aspecte és el que es mostra en l'exemple de la figura anterior.

Atributs

- Un **atribut públic** s'identifica amb el símbol "+". En aquest cas, si una instància a: està enllaçada amb una instància b:, a: pot accedir lliurement als valors emmagatzemats en els atributs de b:.
- Un **atribut privat** s'identifica amb el símbol "-". No es pot accedir a aquest atribut des d'altres objectes, independentment del fet que existeixi un enllaç o no. A efectes pràctics, és com si no existís fora de l'especificació de la classe i, en conseqüència, només es pot utilitzar en les operacions dins de la mateixa classe en què s'ha definit.

Enllaços entre objectes

A partir dels mapes d'objectes s'han detectat enllaços entre objectes. De fet, aquests són molt importants, ja que un objecte només pot cridar l'operació d'un altre objecte si existeix aquest enllaç. Per tant, també cal poder indicar aquests enllaços al especificar la classe.

Depenent de si un objecte ha de gestionar un o molts enllaços, es pot usar un únic atribut o una llista (List). Per exemple, una agenda gestiona moltes pàgines, pel que es pot especificar l'atribut:

```
-pagines: List<Pagina>
```

D'altra banda, si ens interessa controlar la pàgina actual, que només és una, es pot fer:

```
-paginaActual: Pagina
```

Atributs de classe

A part dels atributs que defineixen les propietats de cada instància d'una classe, hi ha un tipus especial d'atributs, anomenats **atributs** de classe. A l'hora de definir-los, es diferencien subratllant-los, si bé la sintaxi és idèntica als atributs genèrics:

```
visibilitat nomAtributClasse: tipus [= valor inicial]
```

Per exemple:

```
+pi: real
```

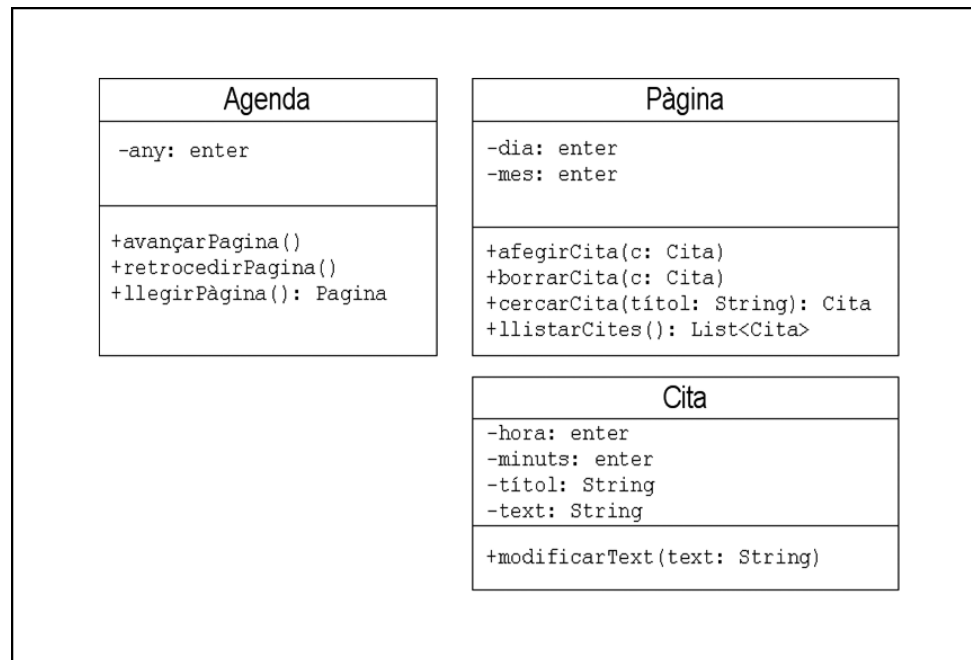
Mètodes

Dins la definició d'una classe, les operacions disponibles s'especifiquen de la manera següent:

```
visibilitat nomOperació (llistaParàmetres): tipusRetorn
```

El camp “llistaParàmetres” té el format següent:

```
nomParàmetre1: tipus, ... , nomParàmetreN: tipus
```

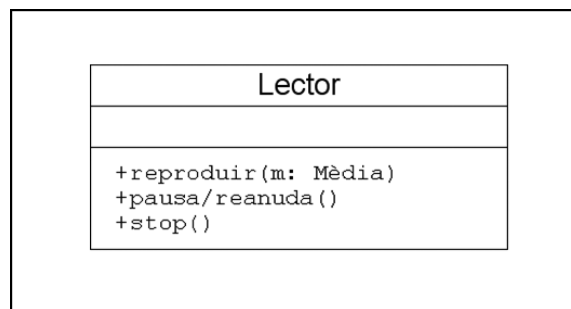


Associacions

El tipus de relació més freqüent és l'associació.

Per tant, perquè dos objectes puguin interactuar hi ha d'haver una associació entre les seves classes al diagrama estàtic UML, de manera que es considera que estan **enllaçats**. En cas contrari, la crida d'operacions no és possible. Quan per mitjà d'un enllaç un objecte **objecteA:** crida una operació sobre un objecte **objecteB:**, les transformacions fetes per l'operació únicament afectaran l'objecte **objecteB:**. No n'afectaran cap de la resta d'instàncies que hi hagi en aquell moment que pertanyin a la mateixa classe que l'**objecteB:**.

El diagrama estàtic UML estarà format en la seva totalitat per la representació gràfica de totes les classes identificades i les relacions que hi ha entre totes elles.



Eines CASE

Normalment, els diagrames estàtics UML es generen mitjançant l'ajut d'eines CASE (*computer aided software engineering*, ,enginyeria de programari assistida per ordinador').

Es considera que hi ha una **associació** entre dues classes quan es vol indicar que els objectes d'una classe poden cridar operacions sobre els objectes d'una altra.

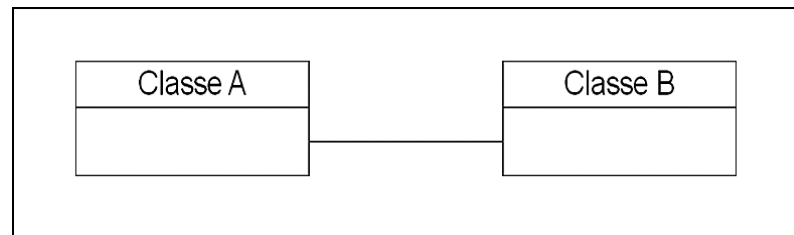
Tot i que les associacions indiquen que hi ha enllaços entre objectes, es representen gràficament en el diagrama respecte a les seves classes. Quan dues classes es representen relacionades, les seves instàncies poden estar enllaçades en algun moment de l'execució de l'aplicació.

Donada una associació, hi ha un conjunt de descriptors addicionals que es poden especificar:

- En el centre, entre les dues classes implicades, s'especifica el **nom de l'associació**. Aquest nom indica què representa l'associació a nivell conceptual.
- En cada extrem de l'associació s'especifica quines són les funcions de les classes implicades. El nom indica, de manera entenedora, el paper que tenen els objectes de cada classe en la relació.
- Amb una fletxa se n'especifica la **navegabilitat**. Partint del nom de l'associació i les funcions de les classes, s'ha de poder establir quina és la classe origen i quina la destinació. A efectes pràctics, la navegabilitat indica el sentit de les interaccions entre objectes: a quina classe pertanyen els objectes que

poden cridar operacions i a quina els objectes que reben aquestes crides.

- Juntament amb la funció, s'especifica la **cardinalitat**, o multiplicitat, de la relació per a cada extrem. La cardinalitat especifica amb quantes instàncies d'una de les classes pot estar enllaçada una instància de l'altra classe.



Si cal, res no impedeix que una associació sigui **navegable** en ambdós sentits, tot i que no és el cas més freqüent. En aquest cas, no cal posar cap fletxa.

La **cardinalitat** defineix quantes instàncies diferents d'una classe ClasseA es poden associar amb una instància de la classe ClasseB en un moment determinat de l'execució del programa.

El nombre d'instàncies per a cada cas s'indica amb una llista de nombres enters, ubicada en l'extrem oposat de l'associació. Per exemple, la cardinalitat que indica quantes instàncies de la ClasseB pot enllaçar una

instància de la ClasseA s'ubica en l'extrem de l'associació en què està representada la classe ClasseB. En cas que es vulgui indicar un nombre indeterminat, sense fita superior en el nombre d'enllaços, s'utilitza el símbol *. Per establir rangs de valors possibles, s'usen les fites inferior i superior separades amb tres punts.

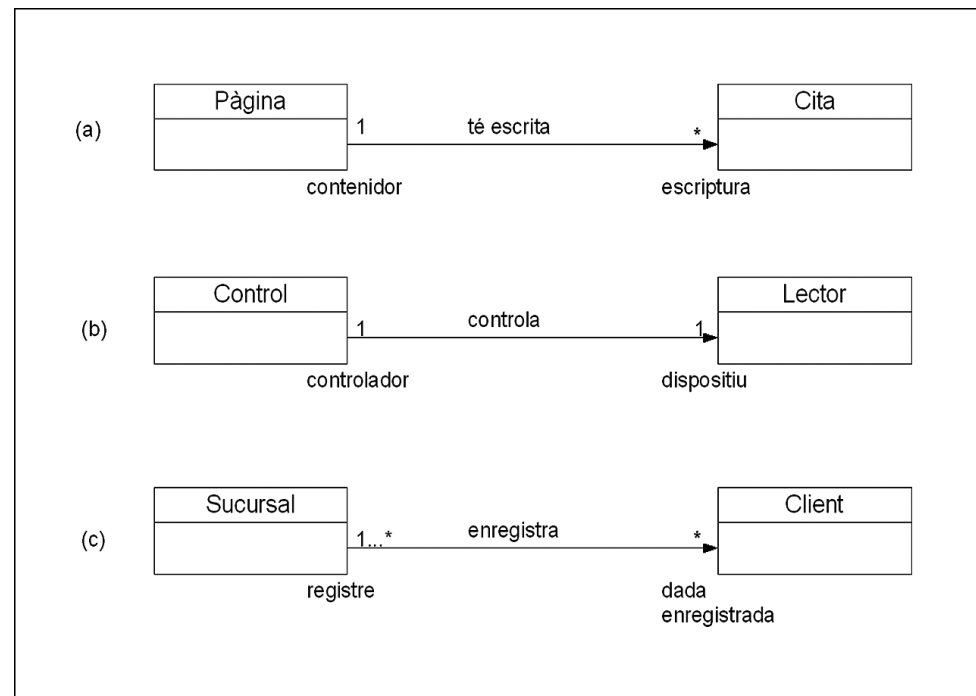
Diferents cardinalitats i el seu significat

- 1: només un enllaç
- 0..1: Cap o un enllaç
- 2,4: dos o 4 enllaços
- 1..*: entre 1 i infinit
- 1..5: entre 1 i 5 enllaços (1, 2, 3, 4 o 5)

Perquè el programa es consideri correcte a l'hora d'implementar el disseny, s'ha de fer el necessari perquè les condicions que expressen les cardinalitats sempre siguin certes. Si una cardinalitat és, per exemple, 1, això vol dir que tot objecte d'aquesta classe sempre està enllaçat amb un, i només un, objecte de l'altra classe. Mai no hi pot haver dins el programa en execució un cas en què no es compleixi aquesta condició.

De tots aquests descriptors, només la navegabilitat i la cardinalitat són imprescindibles (i de les dues, la cardinalitat és la més important), ja que la decisió que es prengui en aquests aspectes dins l'etapa de disseny tindrà implicacions directes sobre la implementació. Les funcions i el nom són opcionals, si bé molt útils amb vista a la comprensió del diagrama estàtic i com a mètode de reflexió per al dissenyador.

En la figura següent es mostra una representació completa d'una associació, amb tota la informació que cal especificar.



A partir del que expressa la figura, aquestes són algunes de les conclusions a les quals arribaria un observador aliè al procés de disseny:

- **A partir d'(a).** Una instància qualsevol de la classe Pàgina pot enllaçar fins a un nombre indeterminat d'objectes diferents de la classe Cita. Això inclou no tenir-ne cap (una pàgina en blanc). Així, doncs, hi ha pàgines que tenen tres cites, d'altres deu, d'altres cap, etc.

A partir d'(a). Recíprocament, donat un objecte qualsevol de la classe Cita, només estarà enllaçat a un únic objecte Pàgina. Per tant, no es pot tenir una mateixa cita en dues pàgines diferents (però sí tenir dues citacions diferents i de contingut idèntic, amb els mateixos valors per als atributs, cadascuna a una pàgina diferent). Tampoc no hi pot haver citacions que, tot i ser en l'aplicació, no estiguin escrites a cap pàgina.

A partir de (b). Un objecte de la classe Control sempre té un objecte de la classe Lector enllaçat. Per tant, un tauler de control sempre controla un únic lector de mèdia. No es pot donar el cas que un tauler de control no controli cap lector. La inversa també es certa; tot lector és controlat per algun tauler de control.

A partir de (b). Donada la navegabilitat especificada, s'està dient que el tauler de control pot cridar operacions sobre el lector, però no a l'inrevés. Això té sentit, ja que és el tauler de control qui controla el lector.

A partir de (c). Donat un client, aquest pot estar enregistrat en més d'una sucursal, però almenys sempre ho estarà en una. Mai no es pot donar el cas d'un client donat d'alta en el sistema però que no estigui enregistrat en cap sucursal.



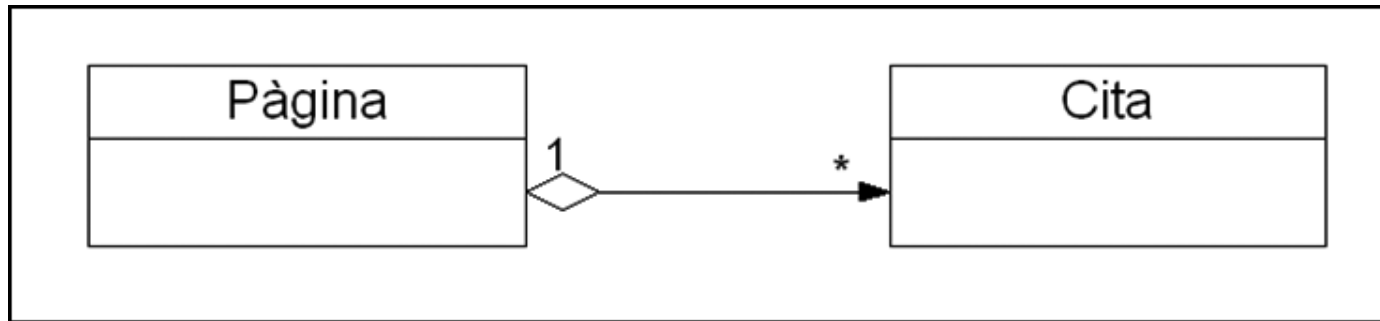
Tot i que no és habitual, res no impedeix que dues classes tinguin més d'una associació entre elles. Normalment, aquest cas s'aplica quan es vol fer una diferenciació explícita de diferents tipus de relació o de funcions entre instàncies de cada classe.

Atès que un transportista pot adoptar diferents papers en la seva relació amb una sucursal, això es pot representar especificant que hi ha diferents tipus d'associació entre sucursals i transportistes.

Tot i la versatilitat dels diagrames estàtics UML, aquests no sempre són capaços de reflectir totes les restriccions necessàries en els enllaços entre objectes de diferents classes. Les associacions indiquen els possibles enllaços, però no especifiquen condicions concretes per a la seva presència. En aquests casos s'utilitza una **restricció textual**. Aquesta no és més que una frase addicional al peu del diagrama, escrita en llenguatge humà, en què s'explica breument en què consisteix aquesta restricció.

Agregacions

Una **agregació** és un tipus d'associació especial que especifica explícitament que hi ha una relació de tot-part entre els objectes de l'agregat (el tot) i el component (la part).

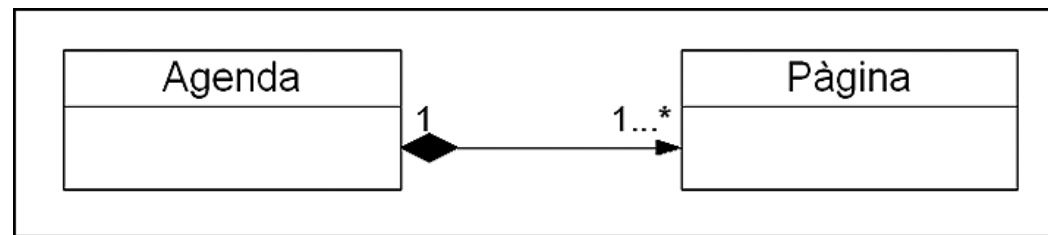


Hi ha un tipus d'associació especial mitjançant la qual el dissenyador vol donar un sentit més específic a l'enllaç, com ara que els objectes de certa classe formen part dels objectes d'una altra. Aquest subtipus d'associació s'anomena *agregació*.

En el diagrama estàtic UML, això es representa gràficament afegint un rombe blanc en l'extrem de l'associació on hi ha la classe que representa el tot. Com que amb aquest símbol ja es diu quina és la relació entre els objectes d'ambdues classes, es poden ometre el nom i la funció en els descriptors de l'associació.

Composicions

Una **composició** és una forma d'agregació que requereix que els objectes components només pertanyin a un únic objecte agregat i que, a més a més, aquest darrer no té sentit que existeixi si no n'existeixen els components.



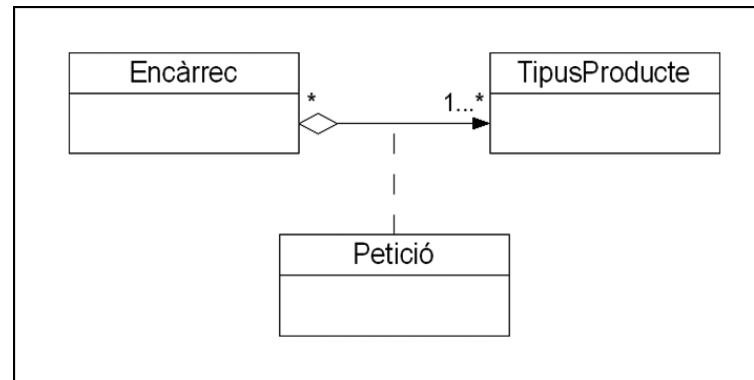
En aquest cas, en usar una composició per representar aquesta associació, el dissenyador expressa que no té sentit una agenda sense pàgines. De fet, conceptualment, el propi conjunt de pàgines és l'agenda en si.

Classes associatives

Hi ha circumstàncies que fan que el dissenyador consideri necessari afegir propietats addicionals a una associació, el valor de les quals pot variar segons quines siguin les instàncies enllaçades. En definitiva, el dissenyador vol especificar atributs en una associació. Amb aquesta finalitat s'usen les *classes associatives*.

Una classe associativa es representa amb el mateix format que una classe: és una nova classe que cal especificar dins la descomposició del problema.

Les **classes associatives** representen associacions que es poden considerar classes.



En la classe associativa **Petició** hi hauria les propietats vinculades a la petició d'un tipus de producte concret dins un encàrrec (per exemple, el nombre de productes que cal enviar o el seu color). Cap d'aquestes propietats no correspon a l'encàrrec ni al tipus de producte. Tot això es plasmarà en forma d'atributs dins d'aquesta classe.

Totes les classes associatives que apareguin en el diagrama estàtic UML s'hauran d'especificar completament, tant pel que fa als atributs com a les operacions. Tot i representar una associació i no un element identificable dins del problema, a efectes pràctics són una classe més, com qualsevol altra.

Una particularitat de les classes associatives és que es poden representar amb classes i associacions normals. Aquest fet és important, ja que és l'única manera de representar-les en un mapa d'objectes i la majoria de llenguatges de programació no suporten les associacions amb aquest tipus de classes vinculades.

Seguint el sentit de la navegabilitat (classe origen - classe destinació) el desenvolupament és el següent:

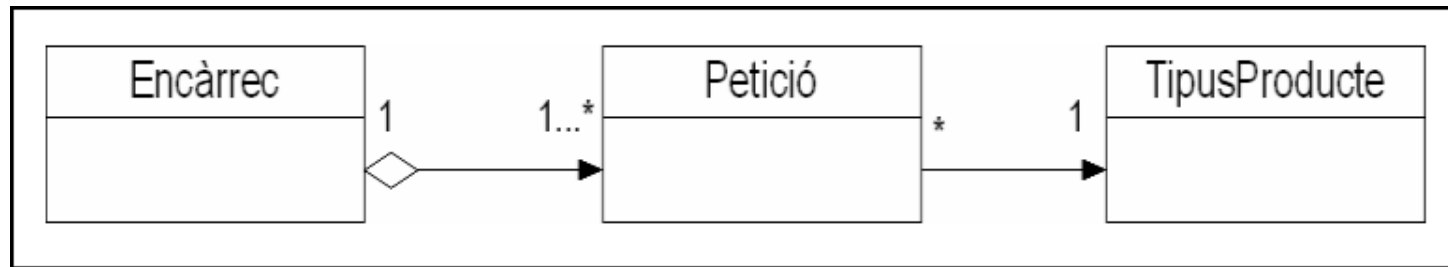
1. Eliminar l'associació original.
2. Generar una associació de la classe origen a la classe associativa. El tipus de la nova associació i la navegabilitat són idèntics a l'original.
3. Generar una altra associació de la classe associativa a la destinació. La navegabilitat és de classe associativa a destinació.

Ara la classe associativa ja és una classe normal dins el diagrama estàtic UML, però atès que ara hi ha dues associacions, cal adaptar-hi les cardinalitats:

- La cardinalitat en els extrems oposats a l'antiga classe associativa sempre és 1.
- La cardinalitat en l'extrem oposat de la classe origen, en què ara hi ha la classe associativa, és la que hi havia respecte a l'antiga classe destinació.

- La cardinalitat en l'extrem oposat de la classe destinació, en què ara hi ha la classe associativa, és la que hi havia respecte a l'antiga classe origen.

El resultat d'aplicar aquesta traducció:



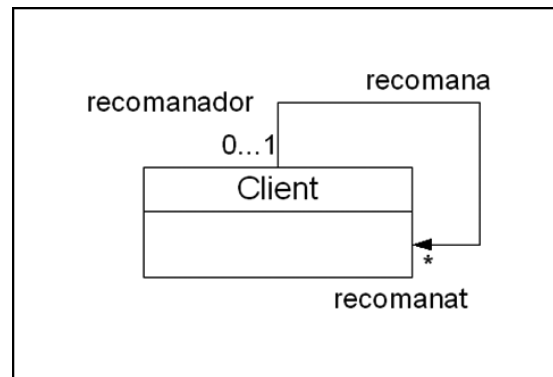
De fet, depenent de les interpretacions que ha fet el dissenyador respecte a la descripció del problema, es pot arribar a aquest diagrama directament des del pas d'identificació de classes. De totes maneres, en fer un diagrama estàtic UML, és recomanable usar classes associatives sempre que apliqui i només fer la traducció en el moment d'implementar.

Associacions reflexives

Atès que una associació indica enllaços entre instàncies d'una classe, res no impedeix que un objecte estigui enllaçat amb objectes del mateix tipus. Quan això passa, es representa mitjançant una associació reflexiva.

Un exemple d'aquest cas es mostra en la figura següent, en què els clients de l'aplicació de gestió recomanen altres clients. Es tracta d'una associació reflexiva, ja que tant qui recomana com qui és recomanat, un client, pertanyen a la mateixa classe.

Una **associació reflexiva** és aquella en què la classe origen i la destinació són la mateixa.



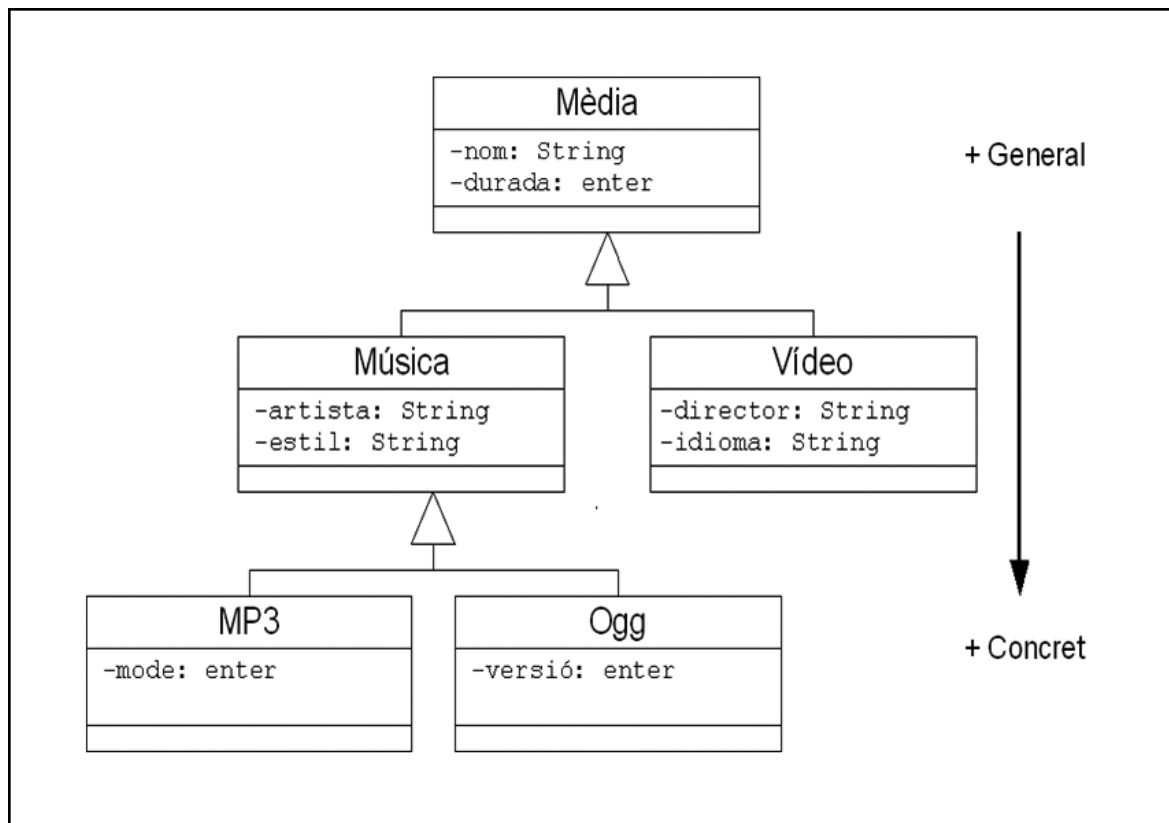
Un dels moments en què el dissenyador ha d'anar amb més compte en aquest tipus d'associacions és quan n'especifica la cardinalitat. La cardinalitat a l'origen de la relació sempre ha de preveure el cas 0. En cas contrari, és impossible generar un mapa d'objectes que la satisfaci, ja que s'hi genera un bucle infinit. Això s'ha de tenir en compte, independentment del fet que, conceptualment, afegir-hi cardinalitat 0 tingui sentit o no.

Exemple de bucle infinit: un arbre genealògic (bé, gairebé infinit)

Herència

Finalment, dins un diagrama estàtic UML també s'estableixen les relacions d'herència entre els objectes de diferents classes. Això permet definir relacions d'especialització/generalització, on la classe a la part superior de la relació representa un concepte més general que el de la part inferior, més específic. Aquest tipus de relació normalment també indica que la classe més específica és idèntica a una altra, excepte en alguns petits aspectes on, o bé és diferent, o bé se suposen propietats addicionals.

Per exemple, diferents tipus de fitxers de mèdia es poden vincular entre si d'acord a una relació d'herència tal com mostra la figura següent. A la dreta es veu quin concepte es considera més general i quin més específic.



CICLE DE VIDA DELS OBJECTES

Creació d'objectes

Per poder manipular-lo, primer cal **crear-lo d'alguna manera** dins el programa. Un cop creat, es poden invocar operacions sobre ell. Però per tal de poder cridar operacions, **és imprescindible un enllaç a l'objecte**.

La creació d'un objecte la realitza sempre una operació especial de la classe, anomenada **constructor**, que es distingeix perquè té el mateix nom que la classe. Els constructors **poden incorporar paràmetres** i això permet que hi pugui haver diferents constructors, que es distingeixen pel nombre i/o els tipus dels seus paràmetres.

Per exemple, si volem crear un objecte de la classe Date proporcionada per Java, en consultarem la documentació:

Constructor Summary	
<code>Date(int year, int month, int day)</code>	Deprecated. <i>instead use the constructor Date(long date)</i>
<code>Date(long date)</code>	Constructs a Date object using the given milliseconds time value.

La classe *Date* incorpora dos constructors, un dels quals ens diu que és obsolet (*deprecated*).

Tota classe té, com a mínim, un constructor i no totes les classes tenen més d'un constructor.

En el llenguatge Java, tot objecte es crea obligatòriament amb l'operador **new** acompanyat de la crida al constructor que correspongui.

```
new Date (109,0,1); // Objecte amb 1-1-2009 a les 00:00:00  
new Date (0); // Objecte amb 1-1-1970 a les 00:00:00
```

L'operador **new** crea un objecte assignant la memòria necessària de manera automàtica.

Fent referència als objectes

Per **accedir a un objecte** es fa declarant una variable per **fer referència** a objectes de la classe concreta i assignant a aquesta variable el resultat de l'execució de l'operador **new**, el qual **retorna una referència** (adreça de memòria) a l'objecte creat.

La sintaxi per declarar una variable de nom “*obj*” per fer referència a objectes de la classe X és:

X obj; → Alerta, però, ja que en aquesta variable no hi ha cap objecte!!

```
// Declaració de variable de referència no inicialitzada

X obj1;
// Creació d'objecte al que es podrà accedir via la variable de
// referència obj1
obj1 = new X(...);
// Declaració de variable de referència i creació d'objecte al
// que es podrà accedir via la variable de referència obj2
X obj2 = new X(...);

// Declaració de variable de referència no inicialitzada
X obj3;
// La variable obj3 fa referència al mateix objecte que fa
// referència la variable obj1
obj3 = obj1;

// Declaració de variable de referència que fa referència al
// mateix objecte que fa referència la variable obj2
X obj4 = obj2;
```

Inicialització d'objectes

La inicialització dels objectes és una tasca que s'efectua durant el procés de construcció dels objectes. Així doncs **les operacions d' inicialització es realitzaran als constructor(s).**

Vegem diverses construccions d'objectes de la classe Date que permeten diferents maneres d'inicialitzar els objectes creats:

```
Date d1 = new Date (109,0,1); //Objecte inicialitzat amb data 1-1-2009 a les 00:00:00
Date d2 = new Date (0);      //Objecte inicialitzat amb data 1-1-1970 a les 00:00:00
Date d3 = new Date ();       //Objecte inicialitzat amb la data i l'hora del sistema
```

Manipulació d'objectes

S'ha de fer per mitjà de les operacions que proporciona la pròpia classe, amb una sintaxi molt simple:

```
<variableQueFaReferènciaObjecte>.<nomMètode>(<paràmetres>);
```

Així, per canviar el dia, el mes o l'any d'objectes Date, el llenguatge Java ens proporciona setDate(), setMonth() i setYear() i podrem cridar-los sobre qualsevol objecte Date:

```
Date d = new Date (109,0,1); // Nou objecte amb valor 1-1-2009
d.setYear (100);             // Canviem valor a 1-1-2000
d.setMonth (5);              // Canviem valor a 1-6-2000
d.setDate (40);              // Canviem valor a 10-7-2000
```

Destrucció dels objectes

Els objectes, en el moment de la seva creació, **ocupen un espai de memòria** i, per tant, cal ser conscients que **cal destruir-los** quan ja no es necessitin.

En alguns llenguatges de programació orientats a objectes (C++ entre ells) és responsabilitat del programador **tenir sempre present les dades dinàmiques generades per tal d'eliminar-les** de la memòria quan ja no siguin necessàries (provoca molts errors).

Java ens estalvia aquesta feina, de manera que ens permet crear tants objectes com es vulgui, els quals mai han de ser destruïts, ja que **és l'entorn d'execució de Java el que elimina els objectes** quan determina que no s'utilitzaran més (GC).

El Garbage Collector (recuperador de memòria) és un procés automàtic de la màquina virtual Java que **periòdicament s'encarrega de recollir els objectes** que ja no es necessiten i els destrueix tot alliberant la memòria que ocupaven.

Escaneja tots els objectes i totes les variables de referències a objectes que hi ha en la memòria de manera que els objectes pels quals no hi ha cap variable de referència que hi apunti són objectes que ja no s'utilitzaran més i, per tant, són recol·lectats per ser destruïts.

Les referències a objectes es perden en els casos següents:

- Quan la **variable** que conté la referència **deixa d'existir** perquè el flux d'execució del programa **abandona definitivament l'àmbit** en què havia estat creada.
- Quan la variable que conté la referència passa a contenir la referència en d'un altre objecte o **passa a valer null**.
- **L'execució** del recuperador de memòria és **automàtica**, però un programa pot demanar al recuperador de memòria que s'executi immediatament mitjançant una crida al mètode **System.gc()**. Ara bé, l'execució d'aquesta crida **no garanteix que la recol·lecció s'efectui**; **depèn**drà de l'estat d'execució de la **màquina virtual**.

Abans que es reculli un objecte, el recuperador de memòria li dóna la possibilitat d'executar unes darreres voluntats, les quals han d'estar recollides en una operació de nom **finalize()** dins la classe a què pertany l'objecte. Aquesta possibilitat pot ser necessària en diverses situacions:

- Quan calgui alliberar recursos del sistema gestionats per l'objecte que és a punt de desaparèixer (arxius oberts, connexions amb bases de dades...).
- Quan calgui alliberar referències a altres objectes per fer-los candidats a ser tractats pel recuperador de memòria.

ENCAPSULACIÓ

Encapsular significa programar com a caixes negres, és a dir, només sabent l'entrada i la sortida sense importar com es faci.

Els objectes només coneixen la seva estructura, no la dels demés. La interacció entre objectes es realitza a través de mètodes. Normalment, els atributs d'un objecte s'han d'obultar o editar a través de mètodes.

Encapsulació de classes

Classes disenyades com a capsas negres: Algú que no ha participat en el seu procés de disseny o de desenvolupament ha de poder usar-les **sense que li calgui saber-ne l'estructura interna:**

- com s'emmagatzema l'estat dels objectes (els atributs), o
- de quina manera s'ha decidit implementar cert mètode.

Per a qui l'està usant, la complexitat necessària per dur a terme la tasca o accedir al que hi ha a dins de la classe són qüestions irrellevants, sempre que la classe acabi realitzant el procés

Quan es defineix una classe, només s'ha de mostrar **què poden fer** els seus objectes, però **no com ho fan**, o mitjançant quines dades. **Això fa que:**

- **Es minimitzin les implicacions de qualsevol modificació** posterior a una classe, fent que aquest canvi es propagui el mínim possible dins de tot el programari desenvolupat.
- Poguem aplicar **programació defensiva**: forçar un major control d'errors dins la capsa negra, de manera que s'eviti que un objecte estigui en un estat que es pugui considerar invàlid o inconsistent (**contenció** dels efectes d'un possible error).

Un dels **mecanismes bàsics** per obtenir classes encapsulades correctament és **establir una interfície** estable en cada classe i no exposar cap camp de dades de manera que sigui directament accessible → per defecte, tots els atributs d'una classe **cal que siguin sempre privats**.

Només ha de ser possible accedir als atributs per mitjà d'operacions accessoras, tant per a la seva lectura com per modificar-ne el valor.

Això afavoreix la minimització dels efectes provocats pels canvis!

Les classes es poden organitzar en paquets i aquesta possibilitat s'acostuma a utilitzar quan tenim un **conjunt de classes relacionades entre elles**.

La definició d'una dada i/o un mètode pot incloure un modificador que indiqui el tipus d'accés que es permet a la dada i/o mètode:

- **public**, que dóna accés a tothom;

- **private**, que prohibeix l'accés a tothom menys pels mètodes de la pròpia classe;
- **protected**, que es comporta com a public per a les classes derivades de la classe i com a private per a la resta de classes;
- **sense modificador**, que es comporta com a public per a les classes del mateix paquet i com a private per a la resta de classes.

Així doncs:

- La majoria de classes seran **públiques**.
- Cada fitxer .java tindrà només una sola Classe pública, amb el mateix nom del fitxer.
- La majoria d'atributs d'una classe seran **privats**.
- Només algunes constants, o casos molt particulars tindran un altre modificador d'accés.

Tipus de classe

Encara que a Java només tenim una sola manera de crear classes, els patrons de disseny ens diuen que podem classificar-les segons la seva funcionalitat:

- **Model:** representen els objectes del problema a resoldre (persona, alumne, cotxe...). El seu nucli solen ser els atributs i els getters i setters d'aquests. Solen tenir a més sobreescrits els mètodes equals, hashCode i toString.
- **Serveis:** implementen la lògica de negoci. El seu nucli són els mètodes públics que ofereixen als consumidors del servei.
- **Auxiliars:** Serveixen per a realitzar operacions auxiliars de càlcul o transformació de dades (patró Adapter per exemple). La majoria estan compostes de mètodes estàtics.
- **Main:** és el punt d'entrada a l'aplicació. La majoria de vegades només serveix per iniciar el procés principal.
- **Test:** classes orientades a fer proves a la nostra aplicació. En Java, els tests unitaris és solen fer amb JUnit.
- **Altres (Controladors...)**

Encapsulació de Mètodes

Si una classe té atributs, segurament tindrà mètodes públics.

Pot tenir sentit l'ocultació de mètodes?

La resposta és afirmativa, ja que en el disseny d'una classe pot interessar **desenvolupar un mètode intern per ser cridat** en el disseny d'**altres mètodes de la classe** i no es vol donar a conèixer a la comunitat de programadors que utilitzaran la classe. Els mètodes privats són interessants per càlculs auxiliars o parcials ja que només es poden invocar desde la pròpia classe.

HERÈNCIA (MODIFICADOR D'ACCÉS I COMPOSICIÓ)

Herencia de classes

Una classe que exten (hereda) d'una altra hereda els seus atributs i els seus mètodes però NO els seus constructors encara que siguin públics. Es poden afegir atributs i mètodes nous. Es tracta d'una associació "es-un" ja que la classe filla és un subtipus de la classe pare. Per exemple: Cotxe és un Vehicle o Lleó és un Animal.

Si emprem **protected** a la classe base, tendrem accés directe als seus atributs (generalment no està recomenat).

```
public class Base {  
  
    private String nom;  
    protected String cognoms;  
  
    //...  
}  
  
public class Filla extends Base {  
  
    public void metode() {  
        //this.nom = "Joan"; //Impossible accedir.  
        this.setNom("Joan"); //Funciona perfectament  
        this.cognoms = "Galmés";  
    }  
}
```

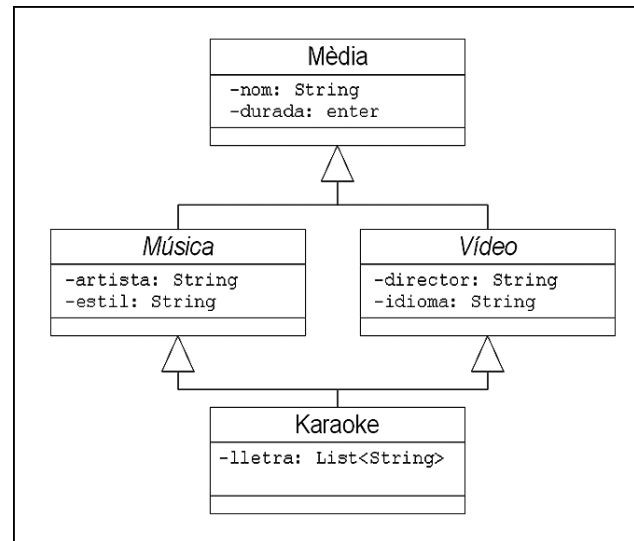
```
}  
}
```

Herència d'interfícies

També podem tenir herència entre interfícies. Les regles són les mateixes que a les classes.

Herencia Múltiple

L'herència múltiple es dona quan una classe hereda directament de més d'una classe.



En Java NO és possible heretar de més d'una classe. Doncs, com ho resol Java? Java sí pot implementar més d'una interfície, així Java resol la casuística de l'herència múltiple.

Composició o Herència?

Hi ha casos que és difícil elegir entre composició o herència. Aquí teniu un enllaç on ho explica molt bé:

[Composition vs Inheritance](#)

POLIMORFISME

El **polimorfisme** consisteix en la possibilitat d'aplicar una mateixa operació a objectes de diferents classes, però cridant una implementació diferent segons la classe a la qual pertanyi l'objecte sobre el qual es crida.

Operacions polimòrfiques

Les subclasses poden accedir als mètodes de la classe pare (públics i protected).

Java ens permet crear instàncies d'objectes i que aquests siguin referenciats per alguna de les classes pare o alguna de les interfícies que implementa.

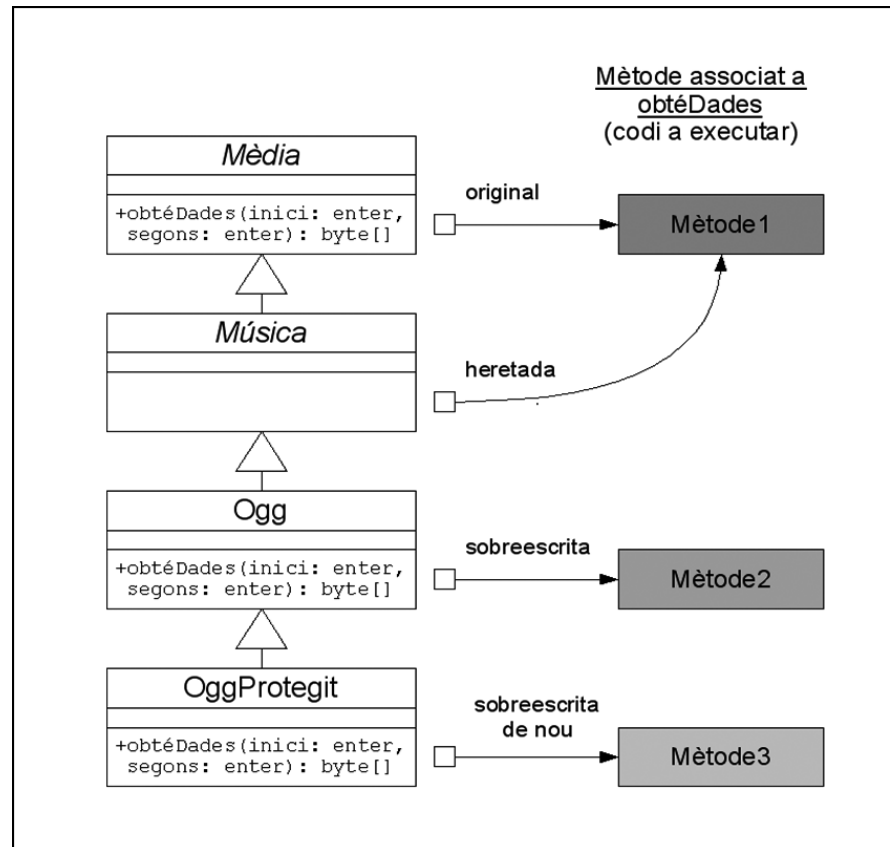
```
Rectangle r = new Quadrat();
```

També permet l'ocultació o sobreescritura de mètodes de les classes derivades

Què passa quan hi ha referències a la classe base però la instanciació és d'un objecte derivat? Doncs que Java tria en temps d'execució el tipus d'objecte. Si aquest tipus ha ocultat un mètode de la superclasse, aleshores es crida a la concreció (**polimorfisme!**), en altre cas, es crida al mètode de la classe base. Això és coneix com operacions polimòrfiques.

És a dir, quan sobre un objecte donat es crida una operació que la seva classe ha sobreescrit, el mètode que s'executa sempre és el definit en la classe a què pertany l'objecte, independentment del tipus de la variable on estigui contingut. Aquest comportament és el que es coneix com a **polimorfisme** dins l'orientació a objectes.

La propietat polimòrfica de les operacions és especialment rellevant quan un objecte ha perdut la



identitat: ha estat assignat a una variable definida amb el tipus d'una superclasse, de manera que només és possible cridar operacions especificades en aquesta superclasse. Tot i aquesta circumstància, quan es crida una

operació polimòrfica, el codi que s'acaba executant és el relatiu al tipus real de l'objecte, no el de la superclasse.

En especificar una **operació** com a **abstracta**, s'indica que no té cap mètode associat, és a dir, que a l'hora d'implementar-la no es codifica. El codi a executar s'obtindrà a partir de la sobreescritura duta a terme per alguna de les seves subclasses.

Una operació abstracta s'especifica escrivint-la en cursiva:

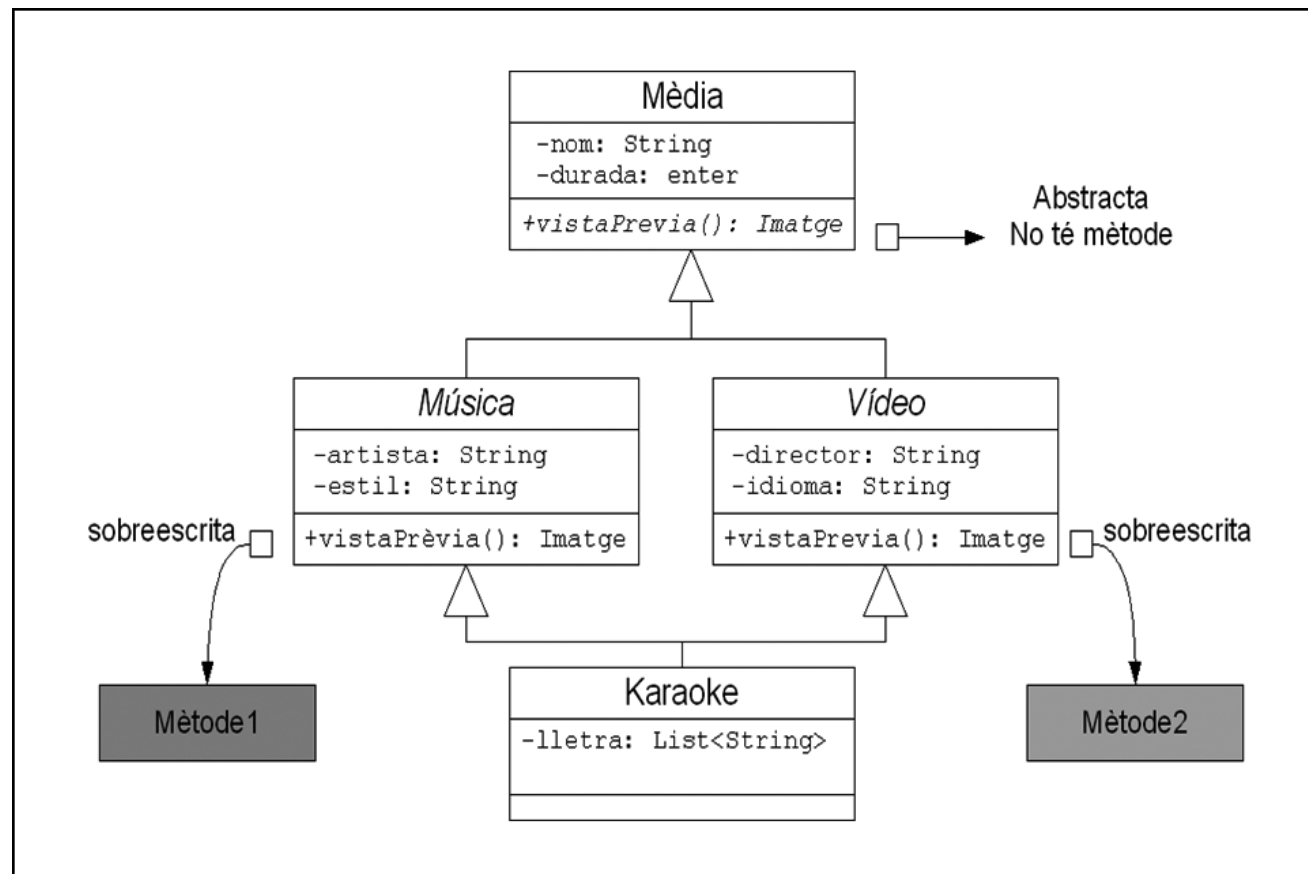
```
+obtéDades(inici: enter, segons: enter): byte[]
```

Mai no es pot donar el cas que sigui possible instanciar una classe que no té un mètode associat per a cadascuna de les seves operacions.

Polimorfisme i herència múltiple

Tot i que l'herència múltiple pot servir per multiplicar la capacitat d'estendre codi i reaprofitar feina, hi ha un problema greu en usar-la quan es permeten mètodes polimòrfics: el problema del diamant. Aquest problema consisteix en la incapacitat de determinar quin mètode cal executar realment quan hi ha duplictat de noms en una operació redefinida dins una jerarquia de classes.

En la figura següent es pot apreciar aquesta problemàtica. Tant en la classe Música com en la classe Vídeo l'operació vistaPrèvia especificada originalment en Mèdia ha esta sobreescrita. Per a cada cas el mètode associat és diferent, ja que cal fer una acció diferent per generar la imatge resultant (per exemple, la informació sobreimpresa depèn dels atributs de cadascú). Per herència, és possible cridar aquesta operació sobre els objectes de la classe Karaoke. El problema és que, en aquesta situació, resulta impossible decidir quin dels dos mètodes cal executar realment.



SOBRESCRITURA DE HASHCODE, EQUALS I TOSTRING

Tots els objectes, de manera directe o indirecta hereden de l'objecte **Object**. Object té una sèrie de mètodes que, seguint les regles d'herència que hem vist, podem sobreescriure. Si no ho sobreescrivim, ens retornarà els valors per defecte de la classe Object. Els principals mètodes que podem sobreescriure són:

- **equals**: ens permet indicar la igualtat entre dos objectes. Per comparar **objectes no es pot fer servir "=="**!
- **hashCode**: retorna un valor únic associat a la instància de la classe.
- **toString**: retorna una representació de l'objecte en format de text (String).

ÚS DE STATIC (VARIABLES, MÈTODES I CLASSES)

Atributs d'objectes i classe

Podem crear totes les instàncies (objectes) que volguem d'una mateixa classe. Aquestes instàncies tenen una còpia pròpia dels atributs. Si volem que totes les instàncies comparteixin un mateix atribut ho hem de fer amb **static**.

```
public static Integer nombreInstancies;
```

Mètodes estàtics

Un mètode estàtic pot ser cridat sense haver d'instanciar la classe. Normalment a les classes amb altun mètode estàtic, tots els altres mètodes també ho són (classes auxiliars). Ús:

```
NomClasse.metodeEstatic();
```

Constants

Apart d'estàtiques també solen ser "final"

```
public static final String PI = 3.1416;
```

Classes estàtiques

Les classes poden ser estàtiques, però només tenen sentit quan treballem amb classes internes.

OBJECTES IMMUTABLES

Els objectes immutables són objectes l'estat dels quals no pot ser modificat una vegada inicialitzats. No són constants ja que es defineixen en temps d'execució i les constants en temps de compilació. L'exemple més comú és la classe **String**.

Recomenacions per a fer objectes immutables:

- Definir totes les propietats com **final private**
- No afegir mètodes *setter*
- Fer la classe final per tal d'evitar l'herència.

CLASSES I MÈTODES ABSTRACTES

Classes abstractes

Les classes abstractes són classes que no es poden instanciar. Poden tenir mètodes amb implementació i atributs però també **mètodes abstractes**, és a dir, sense implementació.

Les classes abstractes també poden implementar interfícies. Si ho fan, poden deixar els mètodes de la interfície sense implementar, però les subclasses tindran l'obligació de fer-ho.

Mètodes abstractes

Els mètodes abstractes han d'estar definits dins una classe abstracta. Defineixen la signatura del mètode, però sense la seva implementació i les seves subclasses (herència) es comprometen a implementar els mètodes abstractes. Si les subclasses no implementen els mètodes abstractes, aquests mètodes han de quedar com a abstractes dins una subclasse abstracta.

```
public interface Transformable {  
    public void rotar();  
    public void voltearHorizontal();  
    public void voltearVertical();  
}
```

```
}  
  
public abstract class ObjecteGrafic implements Transformable {  
  
    protected int x, y;  
  
    public void moureA(int novaX, int novaY) {  
        this.x = novaX;  
        this.y = novaY;  
    }  
  
    abstract public void dibuixar();  
  
    abstract public void canviarTambany(int factorAugment);  
  
}
```

ÚS DE FINAL

Classes

Les classes final són classes que **no es poden estendre**, és a dir, no admeten herència. En una jerarquia serien sempre la darrera fulla. Es poden instanciar i emprar com a classes normals.

Mètodes

Els mètodes final són mètodes que **no admeten sobreescritura** (Override). Si la classe d'un mètode final és extesa, aleshores la classe filla agafarà la implementació de la classe base sense possibilitat de sobre escriure el mètode.

Variables

Les variables final són aquelles que no es poden modificar (constants). Però alerta! Si declaram una **referència** (instància) d'un objecte com a final, vol dir que no el podem reassignar a un altre objecte, però sí que podrà canviar l'estat de l'objecte (per exemple, amb setters). Per exemple, els arrays.

CLASSES INTERNES, LOCALS I ANÒNIMES

Classes anidades

Java ens permet definir classes dins altres classes. Aquestes classes s'anomenen classes anidades. Poden ser de dos tipus, classes estàtiques i classes no estàtiques.

Alerta! L'anidament de classes **no** és composició.

Per què emprar classes anidades?

- Si només s'empra a un lloc és lògic definir-la com a classe anidada.
- Augment de l'encapsulació
- Codi més llegible i mantenible

Classes internes

Les classes internes són les classes anidades no estàtiques. Només poden existir dins el marc d'una instància de la classe externa i poden accedir als membres de la classe externa.

Si es defineix una variable dins una classe interna amb el mateix nom que la classe externa, la classe interna oculta a la classe externa (*shadowing*).

Classes locals

Les classes locals es defineixen dins un bloc, normalment a dins un mètode. Serveixen per a donar més cohesió al codi (si cal).

Classes anònimes

Les classes anònimes permeten **definir i instanciar** a la classe a la vegada. Són com les classes locals però sense nom, així que només es poden emprar una sola vegada.

Cal tenir en compte que es poden definir a partir d'una altra classe (o interfície).

Les classes anònimes les podem crear en el cos d'un mètode (com les locals) però també d'una classe o fins i tot com a paràmetre d'un mètode.

ENUMERACIONS

Les enumeracions en Java són com un tipus de classe que, a més, poden incloure mètodes i altres atributs.

```
public enum Direccio {  
    NORD, SUD, EST, OEST  
}
```

CLASSES GENÈRIQUES

Tradicionalment, per a crear una classe genèrica, s'empraven referències del tipus Object.

```
public class Box {  
    private Object object;  
  
    public void set(Object object) {  
        this.object = object;  
    }  
  
    public Object get() {  
        return object;  
    }  
}
```

Emperò, aquestes classes poden produir problemes (errors) en temps d'execució.

Amb Java 8 (de fet, desde Java 5), podem crear classes genèriques amb l'operador diamant.

```
public class Box<T> {  
    private T object;  
  
    public void set(T object) {  
        this.object = object;  
    }  
  
    public T get() {  
        return object;  
    }  
}
```

Podem emprar més d'un tipus a la vegada.

```
public class Parell<T, S> {  
    private T obj1;  
    private S obj2;  
  
    //Reste de la classe  
  
}
```

Encara que es pot emprar qualsevol valor com a paràmetre, hi ha un conveni per a saber el tipus de paràmetre esperat per la classe genèrica. Són aquests:

- E: element
- K: clau
- T: tipus
- V: valor
- S, U, V... (2n, 3r, 4t, ... tipus)

Per a instanciar una classe genèrica s'havia de fer així:

```
Parell<String, String> parella = new Parell<String, String>("Hola", "Mon");
```

amb Java 8 ho podem abreujar amb l'operador diamant:

```
Parell<String, String> parella = new Parell<>("Hola", "Mon");
```

A les classes genèriques també podem acotar que sigui herència d'una classe particular. Com que **no** podem heretar de més d'una classe (herència múltiple) només podem heretar d'una sola classe, si volem que sigui més d'una, la resta han de ser interfícies.

Exemple d'herència en classes genèriques.

```
public class NumericBox<T extends Number> {  
  
    private T object;  
  
    //reste de la classe  
}
```

Exemple de genèrics amb una classe i una interfície.

```
public class A {  
    //reste de la classe  
}  
  
public interface B {  
    //reste de la interfície  
}
```

```
public class StrangeBox <T extends A & B> {  
  
    //reste de la classe  
}
```

Les classes genèriques també permeten tipus comodí els quals ens permeten més flexibilitat

```
public static double sumOfList(List<? extends Number> list) {  
    double s = 0.0;  
    for (Number n : list)  
        s += n.doubleValue();  
    return s;  
}
```

INTERFÍCIES

Una interfície és un contracte de compromís que adquireix la classe que ho implementa, és a dir, un conjunt d'operacions que la classe es compromet a implementar (mètodes abstractes). Fins Java 7 les interfícies només definien els mètodes, a partir de Java 8 també poden incloure implementació per defecte.

Definició d'una interfície

```
public interface GroupedInterface extends Interface1, Interface2 {  
    // constant declarations  
    // base of natural logarithms  
    double E = 2.718282;  
    // method signatures  
    void doSomething (int i, double x);  
    int doSomethingElse(String s);  
}
```

Les interfícies poden heretar d'altres interfícies (una o més), és a dir, es permet l'herència múltiple entre interfícies.

Implementació d'una interfície

Una classe pot implementar una o vàries interfícies:

```
public class RectanglePlus implements Relatable {  
    //...  
    public int isLargerThan(Relatable other) {  
        RectanglePlus otherRect = (RectanglePlus)other;  
        if (this.getArea() < otherRect.getArea())  
            return -1;  
        else if (this.getArea() > otherRect.getArea())  
            return 1;  
        else  
            return 0;  
    }  
}
```

Podem emprar una interfície per emmagatzemar una referència a un objecte que implementa la interfície.

```
RectanglePlus rectangleOne = new RectanglePlus(10, 20);  
Relatable rectangleTwo = new RectanglePlus(20, 10);
```

Com hem dit abans, amb Java 8 podem escriure una implementació per defecte dels mètodes d'una interfície. Si ho fem, hem de marcar aquests mètodes com **default** o **static**. Aquests són els únics mètodes que no serà necessari implementar obligatòriament per les classes que implementen la interfície.

```
public interface Interficie {  
    default public void metodePerDefecte() {
```



```
        System.out.println("Nou mètode per defecte");  
    }  
  
}
```

CLASSES ABSTRACTES

Les classes abstractes són classes definides amb la paraula reservada **abstract**. Aquestes classes **no es poden instanciar** i poden tenir atributs i mètodes amb implementació o mètodes sense implementació (abstractes). Els mètodes abstractes s'ha de definir obligatòriament a les subclasses a no ser que la subclasse sigui també abstracte, aleshores el mètode es pot deixar sense implementació.

```
public abstract class Abstracta {  
  
    public abstract void salutacio(String s);  
  
    public void saluda() {  
        System.out.println("Hola món!!!");  
    }  
  
}
```

En aquest punt, hom es pot demanar quan emprar una classe abstracta i quan una interfície, ja que ambdues s'assemblen molt. Anem a veure quan emprar una o altra:

Interfícies	Classes abstractes
Classes no relacionades podran implementar els mètodes	Compartir codi amb classes molt relacionades
Si es vol indicar que existeix un tipus de comportament però no sabem qui ho implementa.	Les classes derivades empraran mètodes <code>protected</code> o <code>private</code>
Si necessitem herència múltiple	Volem definir atributs que no siguin estàtics o constants

INTERFACES COMPARABLE I COMPARATOR

Les interfícies (funcionals) Comparable i Comparator serveixen per comparar objectes que no són primitius. Anem a veure'ls en més detall:

Comparable

Comparable és una interfície definida de la següent manera:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

El valor de retorn de compareTo serà un número el qual equivaldrà a:

- 0 si ambdós objectes són igual
- Valor negatiu si l'objecte és menor (típicament -1)
- Valor positiu si és major (típicament 1)

Comparator

La interfície (funcional) Comparator és pareguda però la seva definició és aquesta:

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

En aquest cas es reben dos objectes i el valor de retorn és igual a la interfície Comparable. Comparable ens serveix per indicar un ordre diferent a l'ordre natural definit per Comparable que, encara que no sigui obligatori, si que és recomanable definir Comparable.

INTERFÍCIES FUNCIONALS

Una interfície funcional és una interfície que només té la definició d'un sol mètode abstracte. Estrictament parlant, pot tenir varis mètodes abstractes sempre i quan tots menys un sobreescriuin a un mètode públic de la classe Object. A més, poden tenir un o varis mètodes per defecte o estàtics (com hem vist a les interfícies).

```
Collections.sort(llista, new Comparator<String>() {
```

```
    //Ordenamos la cadena por su longitud
    @Override
    public int compare(String str1, String str2) {
        return str1.length()-str2.length();
    }

```

```
});
```

Java 8 incorpora també l'anotació `@FunctionalInterface` que permet al compilador comprovar si una interfície compleix amb les característiques de ser funcional o no.

Expressions lambda

Una expressió lambda és una classe anonima que implementa interfície funcional.

Les interfícies funcionals i les expressions lambda estan relacionades, de manera que allà on s'espera una instància d'una classe que implementa una interfície funcional, podrem emprar una expressió lambda.

```
Collections.sort(llista, (str1, str2)-> str1.length()-str2.length());
```

Apart de Comparable i Comparator, existeixen moltes altres classes que defineixen interfícies funcionals i que són de gran utilitat. En aquest apartat anem a veure quatre exemples:

Predicate<T>

El mètode abstracte és

```
boolean test(T t);
```

Comprova si es compleix o no una condició. S'empra molt a l'hora de filtrar.

A més es pot combinar amb els mètodes **and**, **or**, i **negate**.

Consumer<T>

El mètode abstracte és:

```
void accept(T t);
```

Serveix per consumir objectes. Un dels exemples més típics és imprimir:

```
//...  
    .forEach(System.out::println)
```

Adicionalment té el mètode **andThen** que permet compondre consumidors per encadenar operacions.

Function<T, R>

El mètode abstracte és:

```
R apply(T t);
```

Serveix per aplicar una transformació a un objecte. L'exemple més clar és el mapeig d'objectes en altres. Adicionalment té altres mètodes:

- andThen: permet compondre funcions
- compose: compon dues funcions (inversa que andThen).

Supplier<T>

El mètode abstracte és:

```
T get();
```

Aquesta funció serveix per a retornar un valor.

Hi ha algunes interfícies especialitzades, és a dir, interfícies funcionals però sense ser una classe genèrica. Algunes classes són:

- IntSupplier
- Long Supplier
- DoubleSupplier
- BooleanSupplier

TEMA 2: PATRONS I ARQUITECTURA CLIENT-SERVIDOR

G O F

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides són coneguts mundialment com "La banda dels quatre" (Gang of Four) i van ser els primers que definiren, a través del seu llibre Design Patterns, el concepte de patró (pattern) i van establir una sèrie de patrons que, després, se'n van definir més.

Què és un patró?

Un patró de disseny és una solució a un problema concret l'efectivitat del qual ha estat demostrada. A més, aquesta solució ha de ser reutilitzable en problemes similars, és a dir, un patró és una solució reutilitzable a un problema comú en el desenvolupament d'un component software.

A més, un patró proporciona una descripció o plantilla de com afrontar la solució final del problema i mostren les relacions entre objectes o classes, sense especificar les classes que finalment solucionaran el problema.

Classificació dels patrons

Segons el nivell d'abstracció, podem classificar els patrons en:

Patrons d'arquitectura: Aquells que expressen un **esquema organitzatiu** estructural fonamental per a sistemes de programari.

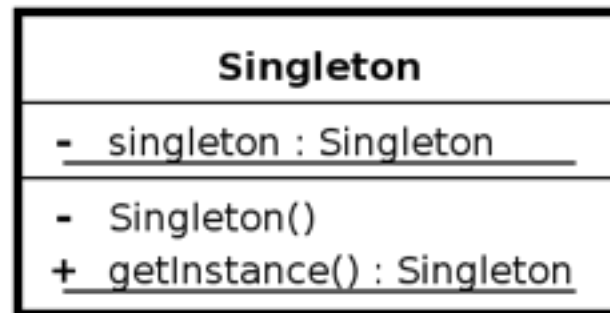
Patrons de disseny: Aquells que expressen esquemes per definir estructures de disseny (o les seves relacions) amb les quals construir sistemes de programari.

Dialectes: Patrons de baix nivell específics per a un llenguatge de programació o entorn concret.

De la classificació anterior, la que ens interessa més en aquest mòdul és la de Patrons de disseny i aquests els podem classificar en:

		PROPÒSIT		
		Creacional	Estructural	Comportament
ÀMBIT	Classe	Factory Method	Adapter	Interpreter Template Method
	Objecte	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

SINGLETON



Singleton és un dels patrons de disseny proposat per *Gang of Four*, el qual serveix per a poder tenir una classe de la qual només volem un única instància (services, managers, controllers...).

Per a implementar-la hem de:

- Definir un únic constructor com a privat per evitar la instanciació.
- Definir un mètode estàtic el qual retorni la instància si ja està creada, o la crei el primer cop.

```
public class ServeiSingleton {  
    //Una instància del objeto que va a existir  
    private static ServeiSingleton instance = null;  
  
    //Evitam les instàncies directes
```

```

private ServeiSingleton() { }

public static ServeiSingleton getInstance() {
    if (instance == null)
        instance = new ServeiSingleton();

    return instance;
}
}

```

Un altre exemple més sofisticat:

```

public class Singleton {

    private static Singleton INSTANCE = null;

    // Feim el constructor privat per tal de que ningú pugui instanciar la classe

    private Singleton() {}

    // mètode sincronitzat que crea la instància de l'objecte per evitar que dos fils
    // concurrents obtinguin diferents instàncies

    private synchronized static void createInstance() {

        if (INSTANCE == null) {

```

```
        INSTANCE = new Singleton();  
    }  
}  
  
public static Singleton getInstance() {  
    if (INSTANCE == null) createInstance();  
    return INSTANCE;  
}  
}
```

Arribat aquest punt, hom pot pensar que Singleton es podria resoldre amb una classe estàtica, però no és així. Aleshores, quina diferència hi ha entre una classe estàtica i una classe que segueixi el patró Singleton? La resposta com sempre, a [Stack Overflow](#)

PROTOTYPE

El patró *Prototype* és un patró de disseny **enfocat a la creació d'objectes**. S'utilitza quan el tipus d'objectes a crear està determinat per **una instància prototípica**, que es clona per produir nous objectes. Aquest patró s'utilitza per a:

- decidir en temps d'execució de forma dinàmica els objectes que s'han d'emprar dintre d'una jerarquia d'objectes possibles, i
- **evitar el cost inherent a la creació d'un nou objecte** en la forma estàndard (per exemple, l'ús de la paraula clau "*new*" que invoca al constructor de l'objecte) essent aquesta creació enormement costosa.



Java ens proporciona un mecanisme que ens permet clonar objectes de forma automàtica. No obstant això, per comprendre aquest mecanisme necessitem entendre la signatura del mètode clone() de Object:

```
protected native Object clone() throws CloneNotSupportedException;
```

I en concret hem d'entendre dues qüestions:

1. Què vol dir la paraula *native*, i
2. Com pot esser que un mètode d'una classe NO abstracta no tengui implementació.

La resposta a les dues qüestions és la mateixa: El mètode clone() realment està implementat, però amb codi nadiu de la màquina en lloc d'estar implementat amb el Bytecode de la màquina virtual de Java (veure).

Java proporciona un mecanisme per la crida a funcions C i C++ des del codi font Java. Per definir mètodes com a funcions C o C++ s'utilitza la paraula clau native.

```
protected native Object clone() throws CloneNotSupportedException;
```

Java proporciona un mecanisme per la crida a funcions C i C++ des del codi font Java. Per definir mètodes com a funcions C o C++ s'utilitza la paraula clau native.

```
public class CLS_NativeData {  
  
    private native String getValue(int i);  
  
    public CLS_NativeData() {  
        super();  
    }  
  
    static {  
        System.loadLibrary("NativeData");  
    }  
  
    public String getData(int i) {
```

```
    return getValue(i);  
}  
}
```

Com es pot veure, és una classe senzilla, no té res complex per entendre. Es pot observar 2 coses importants. El mètode "GetValue" té un atribut anomenat "native" la qual cosa indica que és un mètode natiu. També el mètode "System.loadLibrary" a qual passant el nom de la llibreria que es vol enllaçar amb l'aplicació. El mètode "getData" servirà per accedir al mètode natiu "GetValue" ja que aquesta, està declarada com privada, encara que això és només en aquest cas, vosaltres podeu declarar-ho com públic i evitar de posar un altre mètode "accessor".

En tot cas, el mecanisme de clonació Java es força particular i ens obliga a seguir aquest el procés següent. La classe `java.lang.Object` conté una implementació native i protected del mètode `clone`. Aquesta implementació (que depèn de la màquina sobre la qual s'executi el codi) determina quanta memòria està sent usada per l'objecte a ser clonat, reserva la mateixa quantitat de memòria per a l'objecte clon, i copia els valors de memòria de la vella direcció de memòria a la nova. I al final es torna un `java.lang.Object` el qual és la referència al nou objecte (el clon).

Per implementar la clonació en una classe, s'han de fer dues coses:

1. La classe ha d'implementar la interfície `Cloneable`, aquesta interfície no té mètodes d'implementar. El propòsit de `Cloneable` és indicar al mètode `clone` de `java.lang.Object` que el programador ha donat permís explícit a la classe per permetre que els objectes instanciats a partir d'ella siguin clonats.
2. El mètode `clone` de la classe `java.lang.Object` ha de ser sobreescrit amb un accés de tipus public en comptes de protected. És en aquest mètode que s'implementarà el codi que clona de l'objecte. Un exemple d'una implementació senzilla de `clone` és:

```
public Object clone ()
{
    Object clone = null;
    try
    {
        clone = super.clone ();
    }
    catch (CloneNotSupportedException e)
    {
        // No hauria succeir
    }
    return clone;
}
```

L'excepció `CloneNotSupportedException` és llançada pel mètode `clone` de la classe `java.lang.Object` per prevenir que l'operació de clonació s'executi si no s'ha atorgat el permís per fer-ho (és a dir, s'implementi la interfície `Cloneable`).

En termes senzills el mètode `clone` de la classe `java.lang.Object` crea un nou objecte mitjançant la còpia exacta dels bytes de memòria i retornant una referència, d'això es té que els objectes membres d'un clon apunten als mateixos objectes que els objectes membres de l'objecte original. Per exemple, considerem la següent classe:

```
public class CloneTest () implements Cloneable
{
    public String objetoInterno;

    public Object clone ()
    {
        Object clone = null;
        try
        {
            clone = super.clone ();
        }
        catch (CloneNotSupportedException e)
        {
            // No hauria ocórrer
        }
        return clone;
    }
}
```

IOC I DI

El patró de la IOC es la injeccio de dependències. IOC és un paradigma

Un paradigma és un “exemple o model per realitzar alguna cosa”.

En aquest cas,és un **mètode de programació** que es basa en **invertir el flux dels programes** implementats de forma tradicional, en els que:

La **interacció** entre components de programari es fa de forma **imperativa**, invocant procediments o funcions.

La interacció entre components es fa de forma **secuencial**.

Amb IoC, s'especifiquen respostes desitjades a successos o sol·licituds de dades. **El procés es realitzat per una entitat externa.**

Tradicionalment el programador especifica la **seqüència de decisions i procediments** que poden donar-se durant el cicle de vida d'un programa mitjançant trucades a funcions:

```
text ← Llegir_port_entrada;  
  
si text <> buit fer  
  
valor_xml ← cercar_valor_xml;
```

```
final_fer;  
  
valor_a_respondre ← Procesar_valor_xml(valor_xml);  
  
xml_resposta ← formar_xml_resposta(valor_a_respondre);  
  
escriure_port_sortida(xml_resposta);
```

En la inversió de control s'especifiquen respostes desitjades a successos o sol·licituds de dades concretes, deixant (**delegant!!**) que algun tipus d'entitat o arquitectura externa dugui a terme les **accions de control** que es requereixin en l'**ordre necessari** i per al conjunt de successos que hagin de passar.

```
Framework.missatge_xml_peticio ← classe_missatge_peticio;  
  
Framework.procesar_missatge ← funció_procesa_misatge;  
  
Framework.misatge_xml_resposta ← classe_missatge_resposta;  
  
Framework.escoltar_a_url(url_del_servei)
```

Exemple de IoC

Suposem que els membres d'una aplicació de xarxes socials estan representats per la classe *Persona* següent.

Suposem també que els membres de l' aplicació de xarxa social s'emmagatzemen en una instància de `List<Person>`.

Volem crear un mètode per cercar membres que concordin amb una característica determinada.

```
public class Person {  
  
    public enum Sex {  
        MALE, FEMALE  
    }  
  
    String name;  
    LocalDate birthday;  
    Sex gender;  
    String emailAddress;  
  
    public int getAge() {  
        // ...  
    }  
  
    public void printPerson(){  
        // ...  
    }  
}
```

Un enfocament simplista és crear diversos mètodes; cada mètode busca els membres que concordin amb una característica determinada, com el gènere o l'edat :


```

public static void printPersonsOlderThan(List<Person> people, int age) {
    for (Person p : people) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}

```

El següent mètode és més genèric que *printPersonsOlder*, imprimeix els membres dins d'un determinat rang d'edats:

```

public static void printPersonsWithinAgeRange(List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}

```

Següent possible solució:

```

public static void printPersons(List<Person> people, CheckPerson tester) {
    for (Person p : people) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}

```

```
interface CheckPerson {
    boolean test(Person p);
}
```

```
class CheckPersonService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

Comencem a delegar
responsabilitats

```
printPersons(people, new CheckPersonEligibleForSelectiveService());
```

Un dels arguments de la següent invocació del mètode *printPersons* és una classe anònima que filtra els membres que són elegibles per al servei selectiu: els que són homes i entre les edats de 18 i 25:

```
printPersons(people,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

Dependency Injection

Implementa el paradigma de la Inversió de Control (IoC).

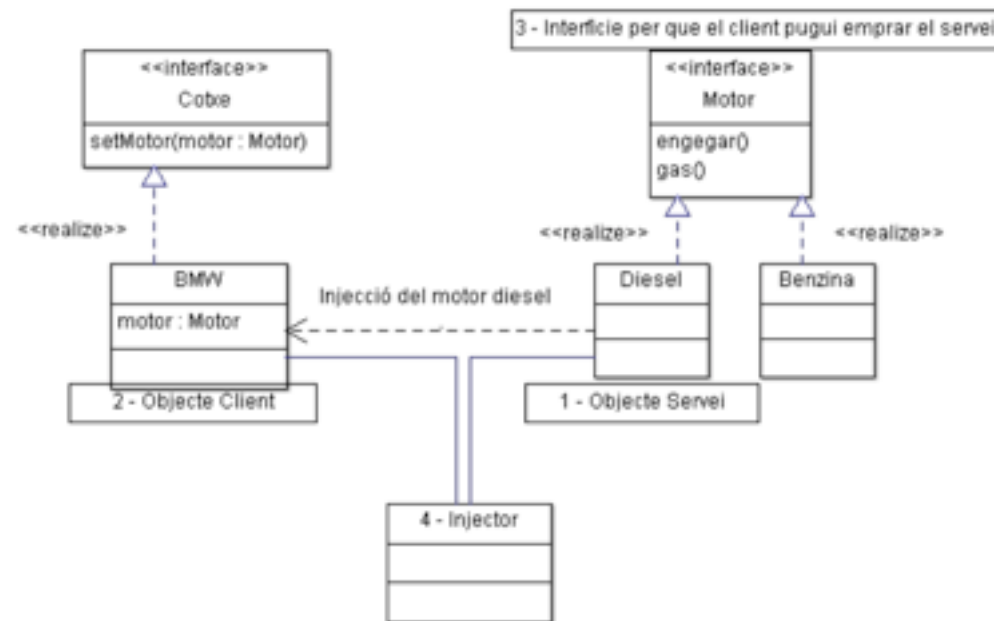
OBJECTIU : Disminuir l'acoblament entre mòduls de programari.

El mòdul depenent es injectat i passa a formar part de l'estat del mòdul principal, en comptes d'instanciar-lo.

Una injecció és el pas d'una dependència d'un component o mòdul a un objecte dependent amb l'objectiu de disminuir l'acoblament entre mòduls.

El patró separa la creació de dependències d'un client del seu propi comportament. Això permet als dissenys dels programes a ser dèbilment acoblats. La injecció de dependència implica quatre elements:

- l'aplicació d'un objecte de servei;
- l'objecte de client en funció del servei;
- la interfície que utilitza el client per comunicar-se amb el servei;
- i l'objecte de l'injector, que és responsable de injectar el servei en el client.



Il·lustració 6: Injecció de dependències

Article Martin Fowler Injecció de Dependències

MVC (MODEL-VISTA-CONTROLADOR)

És un patró arquitectònic que divideix la problemàtica de la creació d'interfícies d'usuari en tres parts:

El Model, el Controlador, i la Vista.

OBJECTIU: Separar la lògica de la creació de les interfícies d'usuari de les dades i la lògica de negoci.

El Model

És la representació de la informació amb la qual el sistema opera, per tant gestiona:

- tots els accessos a aquesta informació,
- tant consultes com actualitzacions,
- privilegis d'accés que s'hagin descrit en la lògica de negoci.

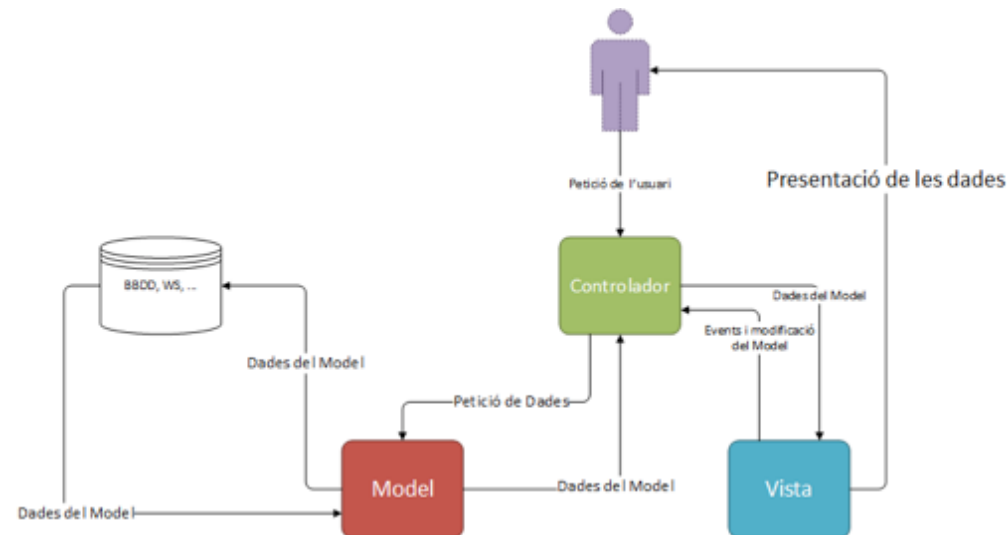
Envia a la 'vista' aquella part de la informació que en cada moment se li sol·licita perquè sigui mostrada (típicament a un usuari). Les peticions d'accés o manipulació d'informació arriben al 'model' a través del Controlador.

El Controlador

Respon a esdeveniments (usualment accions de l'usuari) i invoca peticions al 'model' quan es fa alguna sol·licitud sobre la informació (per exemple, editar un document o un registre en una base de dades). Es podria dir que el 'controlador' fa d'intermediari entre la 'vista' i el 'model' (vegeu Middleware).

La Vista

Presenta el 'model' (informació i lògica de negoci) en un format adequat per interactuar (usualment la interfície d'usuari) per tant requereix d'aquest 'model' la informació que ha de representar com a sortida.



ANTIPATTERNS

Un antipatró (antipatter) és una solució a un problema que amb forma semblant a la d'un patró, **intenta prevenir contra errors comuns de disseny en el programari.**

La idea dels antipatrons és donar a conèixer els problemes que impliquen certs dissenys molt freqüents, per intentar evitar que diferents sistemes acabin cop i un altre en el mateix carreró sense sortida per haver comès els mateixos errors.

Per exemple, un antipatró molt freqüent és emprar el concepte d'herència per a reutilitzar codi.

JAVA EE

Java Platform, Enterprise Edition o Java EE (anteriorment conegut com Java 2 Platform, Enterprise Edition o J2EE fins a la versió 1.4), és una plataforma de programació per desenvolupar i executar aplicacions per a servidor en llenguatge de programació JAVA.

Permet emprar una arquitectura de N capes distribuïdes i components de programari modulars executant-los damunt el servidor d'aplicacions. Aquesta tecnologia inclou Enterprise JavaBeans (EJB), JavaServer Pages (JSP), Servlets, Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Java Message Service (JMS), Java Remote Method Invocation (RMI) i especificacions per a interactuar amb sistemes d'administració de l'empresa.

REQUISITS PER EMPRAR JAVA EE

Apache Tomcat

Apache Tomcat (abans baix el projecte Apache Jakarta) és un contenidor de servlets desenvolupat a l'**Apache Software Foundation**. Tomcat implementa les especificacions de **servlet** i de **JavaServer Pages** (JSP) de Oracle (originalment fou desenvolupat per Sun Microsystems), proporcionant un entorn per al codi Java a executar en cooperació amb un servidor web. Aquest afegeix eines per a la configuració i el manteniment, però també pot ser configurat editant els fitxers de configuració que normalment són en format XML. Tomcat inclou el seu propi servidor HTTP, per això també se'l considera un servidor web independent.

Els components principals d'Apache Tomcat són:

- Catalina: contenidor de Servlets
- Jasper: Motor de JSP
- Coyote: Connector HTTP com a servidor web

L'estructura de directoris (no entrarem en detalls perquè ho veureu a desplegament d'aplicacions web) de Apache Tomcat és:

Directori	Descripció
bin	executables per a iniciar i apagar el servidor Tomcat
conf	configuració del Tomcat
lib	Arxius .jar disponibles per a les aplicacions web
logs	arxius de log. El més important és catalina.out on surten les connexions i errors de l'aplicació web.
temp	arxius temporals emprats per la JVM
webapps	directori on es desen les aplicacions web.
work	directori on es generen els .class per emprar els servlets.

Estructura de directoris d'un projecte Java EE

Directori	Descripció
src	Directori amb els arxius del projecte (servlets, JSP, html, CSS...)
/WEB-INF	Directori amb la metainformació del projecte
/WEB-INF/classes	Servlets i altres classes Java de l'aplicació. Cada subdirector correspon als paquets de classe Java
/WEB-INF/lib	Arxius .jar importats a l'aplicació i accessibles només a dins aquella aplicació.
/WEB-INF/web.xml	Indica l'estructura de directoris que emprarà l'aplicació web. Els principals usos són: definir un arxiu inicial (amb l'etiqueta <welcome-file>) i mapejar els distints Servlets cap a una URL associada a un arxiu html (o JSP) definits.
/META-INF	conté l'arxiu context.xml emprat per a configurar el context de l'aplicació. (Opcional)

Maven

Utilitzareu Maven per administrar els projectes que creareu al llarg d'aquest mòdul. Maven és una eina d'administració de projectes que engloba tot el cicle de vida d'una aplicació, des de la seva creació fins als binaris amb els quals distribuir el projecte.

Un projecte Maven es defineix mitjançant un fitxer anomenat **POM** (*Project Object Model*). Aquest fitxer s'utilitza per definir les instruccions per compilar el projecte, les dependències del projecte (llibreria), etc. En

Maven, l'execució d'un fitxer POM sempre genera un **artefacte**. Aquest artefacte pot ser qualsevol cosa: un fitxer *jar*, un fitxer *swf* de Flash, un fitxer *zip* o el mateix fitxer *pom*.

Les etiquetes més importants del fitxer POM són:

- *groupId*: és com el paquet del projecte. Normalment es posa el nom de l'empresa, ja que tots els projectes amb un *groupId* pertanyen a una sola empresa.
- *artifactId*: és el nom del projecte.
- *version*: nombre de versió del projecte.
- *packaging*: tipus de fitxer generat en compilar el projecte Maven.

Protocol HTTP (i HTTPS)

El protocol de transferència d'hipertext o **HTTP** (*HyperText Transfer Protocol*) estableix el protocol per a l'intercanvi de documents d'hipertext i multimèdia al web.

El propòsit del protocol HTTP és permetre la transferència d'arxius (principalment, en format HTML) entre un navegador (el client) i un servidor web. La comunicació entre el navegador i el servidor es duu a terme en dues etapes:

- El navegador realitza una sol·licitud HTTP.
- El servidor processa la sol·licitud i després envia una resposta HTTP.

Una **sol·licitud HTTP** és un conjunt de línies que el navegador envia al servidor. Inclou:

- Una línia de sol·licitud: és una línia que especifica el tipus de document sol·licitat, el mètode que s'aplicarà i la versió del protocol utilitzada. La línia està formada per tres elements que han d'estar separats per un espai:
 - el mètode
 - l'adreça URL
 - la versió del protocol utilitzada pel client (en general, HTTP/1.1)
 - Els camps de l'encapçalat de sol·licitud: és un conjunt de línies opcionals que permeten aportar informació addicional sobre la sol·licitud i/o el client (navegador, sistema operatiu, etc.). Cadascuna d'aquestes línies està formada per un nom que descriu el tipus d'encapçalat, seguit de dos punts (:) i el valor de l'encapçalat.
 - El cos de la sol·licitud: és un conjunt de línies opcionals que han d'estar separades de les línies precedents per una línia en blanc i, per exemple, permeten que s'enviïn dades per una comanda POST durant la transmissió de dades al servidor utilitzant un formulari.
- Per tant, una sol·licitud HTTP posseeix la següent sintaxi:

Les comandes HTTP més importants per fer una sol·licitud són:

- **GET:** s'utilitza per recollir qualsevol tipus d'informació del servidor. S'empra sempre que es prem sobre un enllaç o es tecleja directament a una URL. Com a resultat, el servidor HTTP envia el document corresponent a la URL seleccionada.
- **HEAD:** sol·licita informació sobre un objecte (fitxer): grandària, tipus, data de modificació... És utilitzat pels gestors de *cache* (memòria cau) de pàgines o els servidors *proxy* per conèixer quan és necessari actualitzar la còpia que es manté d'un fitxer.
- **POST:** serveix per enviar informació al servidor, per exemple les dades contingudes en un formulari. El servidor passarà aquesta informació a un procés encarregat del seu tractament (generalment una aplicació CGI/PHP/ASP...). L'operació que es realitza amb la informació proporci- onada depèn de la URL utilitzada. S'utilitza sobretot en els formularis.

Una **resposta HTTP** és un conjunt de línies que el servidor envia al navegador. Està constituïda per:

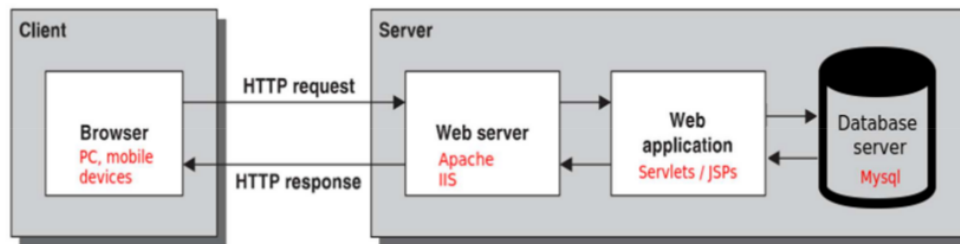
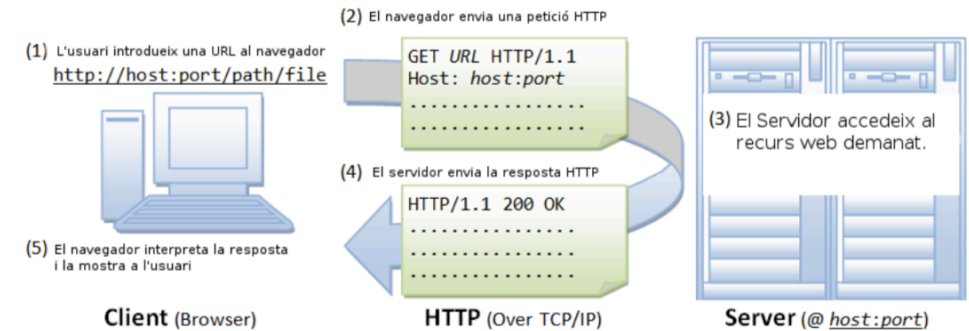
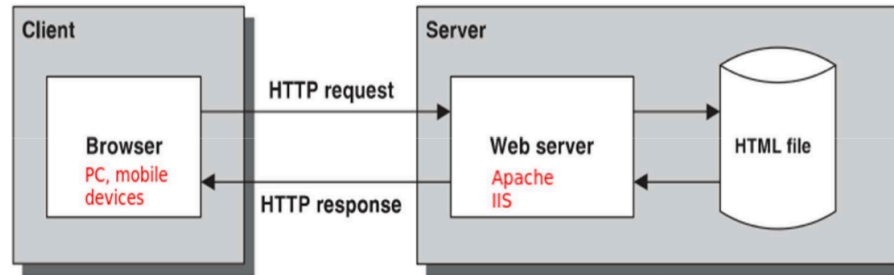
- Una línia d'estat: és una línia que especifica la versió del protocol utilitzada i l'estat de la sol·licitud en procés mitjançant un text explicatiu i un codi. La línia està composta de tres elements que han d'estar separats per un espai:
 - la versió del protocol utilitzada

- el codi d'estat
- el significat del codi
- Els camps de l'encapçalat de resposta: és un conjunt de línies opcionals que permeten aportar informació addicional sobre la resposta i/o el servidor. Cadascuna d'aquestes línies està composta d'un nom que qualifica el tipus d'encapçalat, seguit de dos punts (:) i del valor de l'encapçalat.
- El cos de la resposta: conté el document sol·licitat.

Arquitectura Client-Servidor

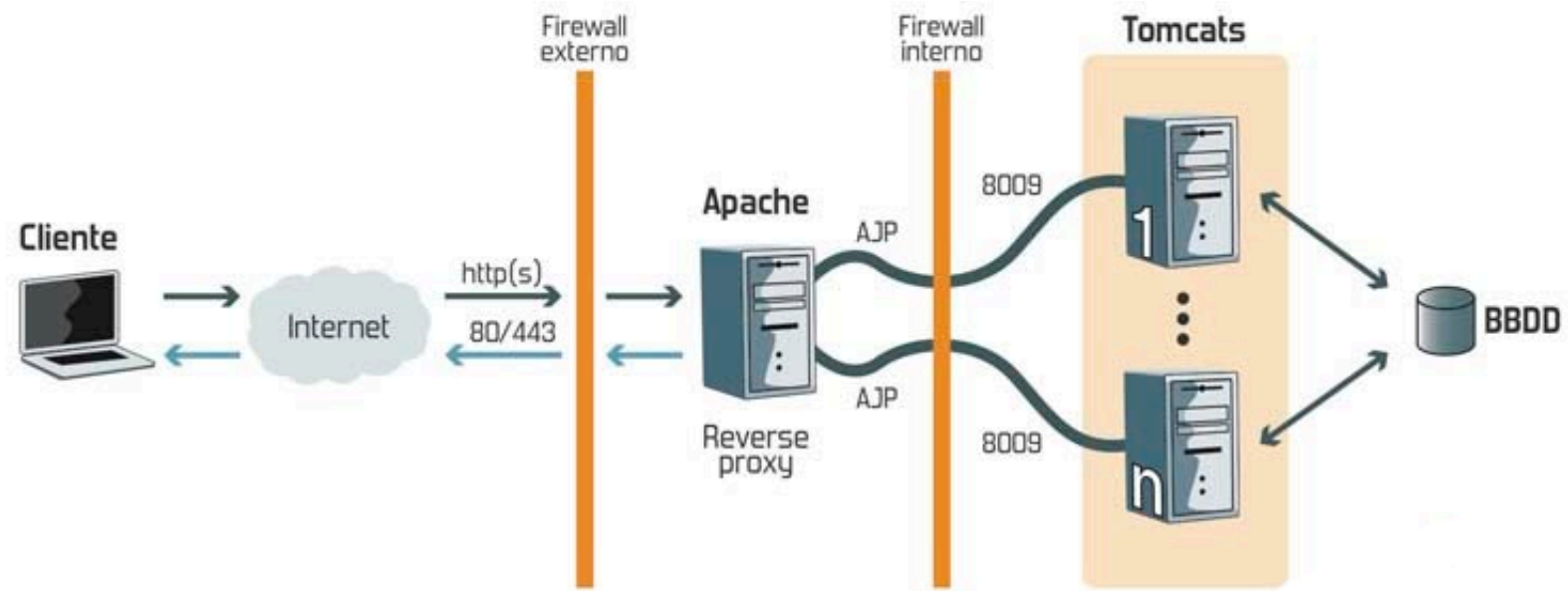
La manera de funcionar que des d'un navegador us permet accedir a la informació que existeix en un servidor es diu **arquitectura client-servidor**.

Un **servidor web** serveix contingut estàtic a un navegador, carrega un arxiu i el serveix a través de la xarxa al navegador d'un usuari. Aquest intercanvi d'informació entre el navegador i el servidor es produeix perquè parlen l'un amb l'altre mitjançant HTTP. Per exemple: Apache



Un **servidor d'aplicacions** és un servidor web i un *framework* de programari on es poden executar aplicacions. És a dir, és un servidor web que permet l'execució d'un programa en el mateix servidor amb les dades proporcionades per una aplicació client. Per exemple: Tomcat

Concretament, el fluxe que hi hauria amb un servidor Tomcat seria el següent:



SERVLETS

Un Servlet és una classe java que permet processar peticions web HTTP (HTTP Request) i retornar una resposta (HTTP Response). Els Servlets són classes que s'executen en el servidor i serveixen per a generar contingut dinàmic d'una aplicació web.

En el patró de disseny MVC els Servlets tenen el paper de **CONTROLADOR**.

Els Servlets poden rebre paràmetres, ja sigui per GET (via URL) o per POST (via formulari o petició AJAX). GET i POST són les peticions tradicionals al servidor, avui dia també trobem PUT i DELETE (es veuran més endavant).

Els Servlets poden retornar al response una resposta que no cal, necessàriament, sigui HTML. També pot ser XML, JSON, un PDF o una imatge.

Els Servlets són la base de molts frameworks com Struts, Spring, JSF i els empren per a gestionar el funcionament intern.

El mapeig dels servlets a l'arxiu web.xml es pot fer de dues formes:

- Tradicional

```
<servlet>  
  <servlet-name>HelloWorld</servlet-name>  
  <servlet-class>servlets.HelloWorldServlet</servlet-class>
```

```
</servlet>
<servlet-mapping>
  <servlet-name>HelloWorld</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

- Ús d'anotacions

```
@WebServlet(name="HelloWorld", urlPatterns="/hello")
public class HelloWorldServlet extends HttpServlet {}
```

Paràmetres

Els paràmetres, independentment del mètode utilitzat, es rebran amb

```
request.getParameter("nom_del_paràmetre");
```

Si el mètode utilitzat és GET, els paràmetres es concatenen a la URL després del símbol "?". Per exemple:
<http://www.esliceu.com?id=10&nom=Joan>

A l'exemple anterior, tindríem 2 paràmetres els quals podríem recuperar així:

```
String identificador = request.getParameter("id");
```

```
String nom = request.getParameter("nom");
```

Si el mètode utilitzat és POST, els paràmetres ens arriben a dins el body del header. Tradicionalment són paràmetres que venen d'un formulari, i rebríem com a nom de paràmetre el *name* del formulari:

```
<input type="text" name="nom">
```

```
String nom = request.getParameter("nom");
```

Tipus MIME

Els tipus MIME (MIME types), són els diferents tipus d'arxiu que pot retornar el servidor. Típicament és un HTML però també podem retornar un altre tipus. El tipus retornat es configura amb `setContentType` dins el Servlet.

Tipus MIME	Descripció
text/html	HTML
text/xml	XML
video/mpeg	Vídeo MPEG
application/pdf	Arxiu PDF
application/json	Arxiu en format JSON

Cal destacar que els tipus MIME han d'estar definits dins el SERVIDOR WEB i també saber que es poden afegir els diferents tipus que hom necessiti.

Response

L'objecte implícit `response` és l'encarregat de retornar la resposta cap a la vista. De moment, destacarem 3 maneres de fer-ho:

`response.sendRedirect("index.html")` --> Redirecció a una altra web. També podem posar una URL mapejada de `web.xml` o una URL externa.

`response.sendError(HttpServletResponse.SC_UNAUTHORIZED);` --> Envia un error HTTP

`Writer` --> Mitjançant l'objecte `PrintWriter` es pot pintar l'HTML a retornar

Upload File

Per a pujar un fitxer al servidor, necessitem enviar les dades mitjançant la codificació "multipart/form-data", així al tag del formulari, hem de dir-li:

```
<form action="..." method="..." enctype="multipart/form-data">
```

La segona cosa que hem de fer, és configurar el Servlet perquè accepti arxiu per multipart mitjançant l'anotació `@MultipartConfig`, per exemple:

```
@WebServlet(name = "Upload", urlPatterns = {"/upload"})
```

```
@MultipartConfig(location = "/tmp",
    fileSizeThreshold = 1048576, // 1mb
    maxFileSize = 1048576, // 1mb
    maxRequestSize = 5242880) // 5mb
```

Aleshores, dins el Servlet, podrem rebre l'arxiu mitjançant: `request.getPart("parametre_arxiu");`

```
// gets absolute path of the web application
String applicationPath = request.getServletContext().getRealPath("");
// constructs path of the directory to save uploaded file
String uploadFilePath = applicationPath + File.separator + UPLOAD_DIR;
```

```
String name = request.getParameter("name");
System.out.println("Name: " + name);
String email = request.getParameter("email");
System.out.println("email: " + email);
```

```
Part arxiu = request.getPart("photo");
```

```
System.out.println("part.getContentType : " + arxiu.getContentType());
System.out.println("part.getSize : " + arxiu.getSize());
System.out.println("part.getName : " + arxiu.getName());
System.out.println("part.getSubmittedFileName : " + arxiu.getSubmittedFileName());

String nomArxiu = arxiu.getSubmittedFileName();
```

```
//Write to server
arxiu.write(uploadFilePath + File.separator + nomArxiu);
request.setAttribute("message", "File uploaded successfully!");
```

JSP

Els Java Server Pages (JSP) són arxius executats en el servidor (Tomcat) amb la finalitat de manejar codi HTML permetent inserir codi natiu Java mitjançant etiquetes JSP.

En una aplicació MVC juguen el paper de **VISTA**, permetent enviar i rebre paràmetres als seus respectius controladors (Servlets). Com que es tracta de la vista, NO ES RECOMANA fer la lògica de negoci a dins la vista, sinó que aquesta permeti només la representació de la informació. També passa el cas contrari, NO ES RECOMANA crear res de la vista al Servlet (per exemple, amb PrintWriter).

JSP disposa d'unes etiquetes especials a través d'un llenguatge conegut com EL (Expression Language), el qual és molt útil a l'hora de recuperar informació de paràmetres o resoldre petites expressions (lògiques i aritmètiques).

A més, a dins JSP podem executar codi Java com hem vist, aquest codi Java a dins JSP s'introdueix utilitzant les etiquetes: `<% -- codi JSP -- %>`. Dintre d'aquestes etiquetes podem utilitzar codi Java. Recordeu que aquest codi s'executa en el servidor abans d'enviar la pàgina al navegador. En aquest cas, `<%@page` és una **directiva** on es defineixen uns paràmetres que s'han de complir a tota la pàgina. En concret, es defineix el tipus de contingut de la pàgina JSP i la seva codificació.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```


També podem fer altres coses, com importar classes o bé indicar si volem activar la sessió de l'usuari.
Exemple:

```
<%@page session="true" import="java.io.*, miPackage.miClase"%>
```

Resum d'etiquetes JSP

EL (Expression Language): permet la resolució dinàmica dels objectes i els mètodes de Java en pàgines JSP i Facelets. EL només permet la lectura de variables (getters) i no els setters. A més, no permeten la iteració de llistes ni arrays. Les expressions *EL* tenen la forma de $\${foo}$ i $\#{bar}$. També pot resoldre expressions lògiques o aritmètiques. Per exemple: $\${2 + 3}$ o $\${x + 2}$

Directives(`<%@..%>`): indiquen informació general de la pàgina, com pot ser importació de classes, pàgina a invocar davant errors, si la pàgina forma part d'una sessió, incloure una altre pàgina o fitxer, etc.

Exemple:

```
<%@page session="true" import="java.util.ArrayList"%>
```

```
<%@include file="titulo.txt"%>
```

Declaracions (`<%! .. %>`): serveixen per a funcions mètodes o variables. Aquestes variables conservaran el seu valor entre successives execucions. A més a més, les variables declarades dintre d'aquestes etiquetes esdevenen variables globals i poden ser utilitzades des de qualsevol lloc de la pàgina JSP.

```
<%-- Exemple --%>
<%!
private String ahora() { return ""+new java.util.Date(); }

%>
```

Scriptlets (<%...%>): codi Java incrustat (embedded).

```
<table>

<% for (int i=0;i<10;i++) { %>

<tr><td> <%=i%> </td></tr>

<% } %>

</table>
```

Expressions (<%=...%>): expressions Java que s'avaluen i s'envien a la sortida.

```
<%= new java.util.Date() %>
```

Objectes implícits dins JSP

JSP té una sèrie d'objectes implícits que podem utilitzar. Aquests són:

Objecte	Tipus
out	JspWriter
request	HttpServletRequest
response	HttpServletResponse
application	ServletContext
session	HttpSession

Scopes

Scope	Description
page	The bean is stored in the implicit PageContext object.
request	The bean is stored in the HttpServletRequest object.
session	The bean is stored in the HttpSession object.
application	The bean is stored in the ServletContext object.

JSTL (JSP STANDARD TAG LIBRARY)

La llibreria estàndard d'etiquetes **JSTL** és una col·lecció d'etiquetes JSP que encapsula les funcionalitats principals de moltes aplicacions JSP.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

JAVABEANS

Els JavaBean són classes Java que representen el MODEL dintre del patró MVC. Per a crear un JavaBean a dins la nostra arquitectura de Servlets + JSP la classe que pertany al model ha de complir que:

- Tingui un constructor sense arguments.
- Proporcioni getters i setters per accedir a les variables. Les variables booleanes canvien i enlloc de ser `getHabilitat()` es fa amb `isHabilitat()`.
- Implementar la interfície `Serializable`

TEMA 3: FRAMEWORKS

SPRING 5: SPRING CORE I SPRING MVC

TÈCNiques D'ACCÉS A DADES

Hibernate y JPA

JPA: especificació que NO té implementació. L'especificació de JPA defineix el següent:

- Especifica com es relacionen els objectes de Java amb l'esquema de Base de dades (XML o anotacions).
- API per realitzar operacions CRUD emprant `javax.persistence.EntityManager`
- Consultes a BBDD amb JPQL
- Motor de persistència que interacciona amb les instàncies transaccionals i les optimitza.

Hibernate: Framework que implementa les funcionalitats de JPA a través d'un dels seus mòduls, Hibernate ORM. Inclou més funcionalitats que les que especifica JPA.

3.3.4 Definición de nuestras clases modelo.

En este ejemplo, tan solo tendremos una clase modelo, cuyo contenido es el siguiente:

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class User {

    @Id
    private int id;

    @Column
    private String userName;

    @Column
    private String userMessage;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }
}
```



```

    public void setUsername(String userName) {
        this.userName = userName;
    }

    public String getUserMessage() {
        return userMessage;
    }

    public void setUserMessage(String userMessage) {
        this.userMessage = userMessage;
    }
}

```

Aunque estamos trabajando con Hibernate nativo, usamos las anotaciones de JPA (entre otras cosas, algunas anotaciones de Hiberante ya están deprecadas).

- **@Entity** indica que esta clase es una *entidad* que deberá ser gestionada por el motor de persistencia de Hibernate.
- **@Id** indica que, de todos los atributos, ese será tratado como clave primaria.
- **@Column** indica que ese atributo es una columna de la tabla resultante.

3.3.5 Clase de aplicación

Por último, nos falta implementar el código necesario para cargar la configuración del fichero **hibernate.cfg.xml** y realizar las operaciones necesarias. En nuestro caso, insertar dos nuevas entidades en la base de datos.

El código es el siguiente. Que no le preocupe al lector si no lo comprende completamente, ya que se irá desgranando a lo largo de las siguientes lecciones:

```

import org.hibernate.Session;
import org.hibernate.SessionFactory;

```

```

import org.hibernate.cfg.Configuration;

/**
 * Primer ejemplo con Hibernate Openwebinars.net
 *
 */
public class App {
    public static void main(String[] args) {
        // Inicializamos un objeto SessionFactory con la configuración
        // del fichero hibernate.cfg.xml
        SessionFactory sf = new Configuration().configure().buildSessionFactory();

        // Iniciamos una sesión
        Session session = sf.openSession();

        // Construimos un objeto de tipo User
        User user1 = new User();
        user1.setId(1);
        user1.setUserName("Pepe");
        user1.setUserMessage("Hello world from Pepe");

        // Construimos otro objeto de tipo User
        User user2 = new User();
        user2.setId(2);
        user2.setUserName("Juan");
        user2.setUserMessage("Hello world from Juan");

        // Iniciamos una transacción dentro de la sesión
        session.beginTransaction();

        // Almacenamos los objetos
        session.save(user1);
        session.save(user2);
    }
}

```

```
// Commiteamos la transacción  
session.getTransaction().commit();
```

```
// Cerramos todos los objetos  
session.close();  
sf.close();
```

```
}
```

```
}
```

Si comprobamos a través de Mysql Workbench, en nuestro esquema **hibernate** tendremos una nueva tabla, llamada **user** y que tendrá dos filas insertadas.

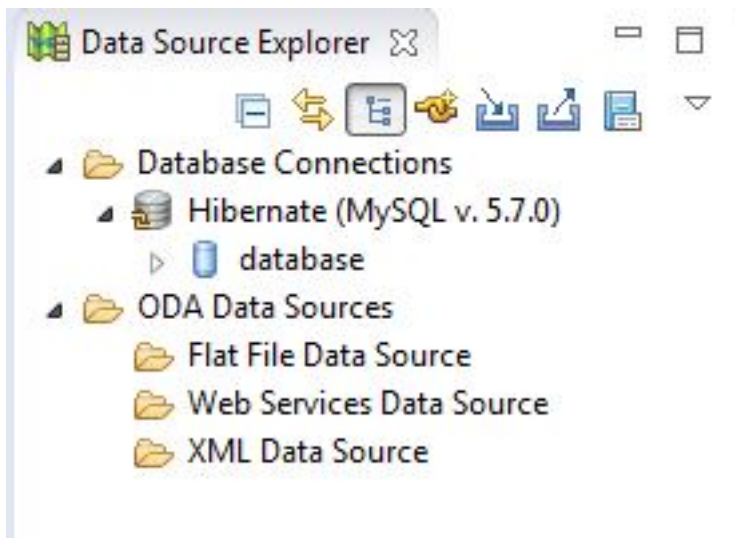
4.5 Configuración de la unidad de persistencia

4.5.1 Clases gestionadas

A través del asistente, añadimos nuestra clase *User*, y podemos marcar la opción de *Exclude unlisted classes*

4.5.2 Conexión

En la segunda pestaña, vamos a definir nuestra conexión. Lo hacemos como un *Resource Local* y, bien podemos establecer los datos a mano, o dar de alta una nueva conexión a base de datos, a través de la vista *Data Source Explorer*.



4.5.3 Opciones de Hibernate

Vamos a establecer ahora algunos parámetros propios de Hibernate, para que nuestro proyecto pueda funcionar. Son muy parecidos al proyecto anterior:

- `hibernate.dialect:org.hibernate.dialect.MySQL5InnoDBDialect`
- `hibernate.connection.driver:com.mysql.jdbc.Driver`
- `hibernate.hbm2ddl.auto:create`
- `hibernate.show_sql:true`
- `hibernate.format_sql:true`

Algunas se pueden añadir directamente desde la pestaña *Hibernate*, y otras desde la pestaña *Properties*.


```

public class App {
    public static void main(String[] args) {

        //Configuramos el EMF a través de la unidad de persistencia
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("PrimerEjemploHibernateJPA");

        //Generamos un EntityManager
        EntityManager em = emf.createEntityManager();

        //Iniciamos una transacción
        em.getTransaction().begin();

        // Construimos un objeto de tipo User
        User user1 = new User();
        user1.setId(1);
        user1.setUserName("Pepe");
        user1.setUserMessage("Hello world from JPA with Pepe");

        // Construimos otro objeto de tipo User
        User user2 = new User();
        user2.setId(2);
        user2.setUserName("Juan");
        user2.setUserMessage("Hello world from JPA with Juan");

        //Persistimos los objetos
        em.persist(user1);
        em.persist(user2);

        //Commiteamos la transacción
        em.getTransaction().commit();

        //Cerramos el EntityManager

```

```
em.close();
```

```
}  
}
```

A diferencia del proyecto anterior, en este caso tenemos que inicializar dos objetos **EntityManagerFactory** y **EntityManager**. El segundo será nuestra interfaz directa con la base de datos, teniendo los métodos necesarios para consultar, actualizar, insertar o borrar datos. El primero es la *factoría* que nos permite construir al segundo, cargando los datos de nuestra unidad de persistencia.

5.1 Y volvemos a comenzar

Vamos a realizar, por tercera vez, la misma operación. En este caso, el punto de partida será un proyecto Spring Boot. Animamos al lector a que al menos visite el **curso online de Spring MVC** para manejar bien algunos conceptos como la inyección de dependencias, el patrón MVC de Spring o el uso de Spring Boot.

5.2 Creación del proyecto

5.2.1 Datos iniciales

Creamos un nuevo proyecto. En este caso, se trata de un proyecto *Spring Starter Project*. Podemos dejar los elementos de configuración (Maven, Java 1.8, ...).

- groupId: **com.openwebinars.hibernate**
- artifactId: **PrimerEjemploSpringJPAHibernate**
- description: a gusto del programador
- package: **com.openwebinars.hibernate.primerejemplospringjpahibernate**

(Como siempre, estos datos pueden ser establecidos por el alumno siguiendo su propia nomenclatura).

5.2.2. Dependencias

Para aquellos que ya hayan creado algún proyecto de Spring Boot será conocido que este asistente nos permite seleccionar las dependencias que vamos a utilizar. Spring nos facilita mucho el trabajo, ya que podemos marcar aquello que sea necesario, y él se encargará de incluirlo en el `pom.xml`:

- *Web*: indicaremos que vamos a crear un proyecto Web MVC. Se encarga de insertar las dependencias, también, de *IoC container*.
- *SQL > MySQL*: incluirá el conector de Mysql para Java.
- *JPA*: nos permite utilizar en nuestro proyecto JPA, Spring Data JPA, Spring ORM e Hibernate. Esta dependencia nos ahorra tener que incluir manualmente las dependencias de JPA+Hibernate.

5.3 Configuración de la base de datos

Ya que estamos usando Spring Boot, haremos uso de la configuración vía Java, en lugar de a través de descriptores XML. Para configurar la base de datos, usaremos una clase con todos los elementos necesarios, y el fichero de *properties*.

Creamos una nueva clase, llamada DatabaseConfig:

```
import java.util.Properties;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.annotation.EnableTransactionManagement;
```



```
@Configuration
@EnableTransactionManagement
public class DatabaseConfig {
```

```
    /**
     * Definición del DataSource para la conexión a nuestra base de datos.
     * Las propiedades son establecidas desde el fichero de properties, y
     * asignadas usando el objeto env.
     */
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(env.getProperty("db.driver"));
        dataSource.setUrl(env.getProperty("db.url"));
        dataSource.setUsername(env.getProperty("db.username"));
        dataSource.setPassword(env.getProperty("db.password"));
        return dataSource;
    }
```

```
    /**
     *
     * Declaración del EntityManagerFactory de JPA
     */
    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean entityManagerFactory = new
LocalContainerEntityManagerFactoryBean();

        //Le asignamos el dataSource que acabamos de definir.
        entityManagerFactory.setDataSource(dataSource);
    }
```

```
// Le indicamos la ruta donde tiene que buscar las clases anotadas
```

```
entityManagerFactory.setPackagesToScan(env.getProperty("entityManager.packagesToScan"));
```

```
// Implementación de JPA a usar: Hibernate
```

```
HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();  
entityManagerFactory.setJpaVendorAdapter(vendorAdapter);
```

```
// Propiedades de Hiberante
```

```
Properties additionalProperties = new Properties();  
additionalProperties.put("hibernate.dialect", env.getProperty("hibernate.dialect"));  
additionalProperties.put("hibernate.show_sql",  
env.getProperty("hibernate.show_sql"));  
additionalProperties.put("hibernate.hbm2ddl.auto",  
env.getProperty("hibernate.hbm2ddl.auto"));  
entityManagerFactory.setJpaProperties(additionalProperties);
```

```
return entityManagerFactory;
```

```
}
```

```
/**
```

```
* Inicializa y declara el gestor de transacciones
```

```
*/
```

```
@Bean
```

```
public JpaTransactionManager transactionManager() {
```

```
    JpaTransactionManager transactionManager = new JpaTransactionManager();
```

```
    transactionManager.setEntityManagerFactory(entityManagerFactory.getObject());
```

```
    return transactionManager;
```

```
}
```

```
/**
```

```
*
```

```
* Este bean es un postprocessor que ayuda a relanzar las excepciones específicas
```

```

    * de cada plataforma en aquellas clases anotadas con @Repository
    *
    */
@Bean
public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {
    return new PersistenceExceptionTranslationPostProcessor();
}

```

```

@Autowired
private Environment env;

```

```

@Autowired
private DataSource dataSource;

```

```

@Autowired
private LocalContainerEntityManagerFactoryBean entityManagerFactory;

```

```

}

```

Y el fichero de properties:

Base de datos

```

db.driver: com.mysql.jdbc.Driver
db.url: jdbc:mysql://localhost/hibernate
db.username: openwebinars
db.password: 12345678

```

Hibernate

```

hibernate.dialect: org.hibernate.dialect.MySQL5InnoDBDialect
hibernate.show_sql: true
hibernate.hbm2ddl.auto: create
entitymanager.packagesToScan: com.openwebinars.hibernate.primerejemplospringjpa.hibernate

```

Los diferentes métodos anotados con **@Bean** sirven para construir esos beans *al vuelo* e inyectarlos directamente.

- El bean `dataSource` es el que crea dicho objeto con los datos definidos en el fichero de properties. Un `DataSource` es un objeto Java que nos permite generar conexiones con una base de datos.
- El bean `entityManagerFactory` es el que utilizaremos más adelante para construir instancias de `EntityManager`. Para crearlo, le tenemos que asignar el `dataSource` y un `vendorAdapter`, es decir, una implementación concreta de JPA. En nuestro caso es Hibernate.
- El bean `transactionManager` permitira utilizar la anotación `@Transactional`, que estudiaremos más adelante.
- Por último, el bean `exceptionTranslation` sirve para propagar las excepciones de bases de datos a las clases que implementemos y anotemos con `@Repository`.

5.4 Nuestro modelo y nuestra clase DAO (*Data Access Object*)

Nuestro modelo sigue siendo el mismo de los últimos proyectos:

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
```

```
@Entity
public class User {
```

```
    @Id
    private int id;
```

```
    @Column
    private String userName;
```

```
    @Column
    private String userMessage;
```

```
    public int getId() {
        return id;
    }
```

```
public void setId(int id) {  
    this.id = id;  
}
```

```
public String getUserName() {  
    return userName;  
}
```

```
public void setUserName(String userName) {  
    this.userName = userName;  
}
```

```
public String getUserMessage() {  
    return userMessage;  
}
```

```
public void setUserMessage(String userMessage) {  
    this.userMessage = userMessage;  
}
```

```
}
```

En este proyecto, vamos a añadir una nueva clase, haciendo uso del patrón *Data Access Object*, que nos invita a crear un objeto que encapsule las operaciones CRUD sobre una entidad.

```
import java.util.List;  
import javax.persistence.EntityManager;  
import javax.persistence.PersistenceContext;  
import javax.transaction.Transactional;  
import org.springframework.stereotype.Repository;
```

```
@Repository
@Transactional
public class UserDao {
```

```
    // A través de la anotación @PersistenceContext, se inyectará automáticamente
    // un EntityManager producido desde el entityManagerFactory definido en la clase
    // DatabaseConfig.
```

```
    @PersistenceContext
    private EntityManager entityManager;
```

```
    /**
     * Almacena el usuario en la base de datos
     */
    public void create(User user) {
        entityManager.persist(user);
        return;
    }
```

```
    /**
     * Elimina el usuario de la base de datos.
     */
    public void delete(User user) {
        if (entityManager.contains(user))
            entityManager.remove(user);
        else
            entityManager.remove(entityManager.merge(user));
        return;
    }
```

```
    /**
     * Devuelve todos los usuarios de la base de datos.
     */
```

```
@SuppressWarnings("unchecked")
public List<User> getAll() {
    return entityManager.createQuery("from User").getResultList();
}
```

```
/**
 * Devuelve un usuario en base a su Id
 */
public User getById(int id) {
    return entityManager.find(User.class, id);
}
```

```
/**
 * Actualiza el usuario proporcionado
 */
public void update(User user) {
    entityManager.merge(user);
    return;
}
```

```
}
```

Destaquemos algunos elementos:

- El estereotipo **@Repository** indica que esta clase es algo así como un almacén de datos.
- La anotación **@Transactional** permitirá, al tener definido un motor de transacciones en la clase DatabaseConfig, provocará que se invoquen los métodos begin() y commit() de forma “mágica” en el inicio y el fin cada método de esta clase.
- La anotación **@PersistenceContext** nos permite realizar dos operaciones en una: inyectar un **EntityManager** que será creado desde el bean **entityManagerFactory**.

6.3 Mapeo de entidades con anotaciones

Para convertir una clase Java en una **entidad**, tan solo tenemos que hacer dos cosas:

- Anotarla con **@Entity**.
- Incluir un atributo que esté anotado con **@Id**.

```
@Entity  
public class MyEntity {
```

```
    @Id  
    private long id;
```

```
    public long getId() {  
        return id;  
    }
```

```
}
```

Por defecto, esta entidad se mapea en la base de datos con una tabla llamada **MIENTIDAD**.

Como la anotación **@Id** es sobre un campo, por defecto, Hibernate habilitará por defecto todos los atributos de la clase como propiedades persistentes.

6.3.1 Eligiendo la clave primaria.

Si bien en el mundo de los diseñadores de bases de datos relacionales ha existido siempre una serie de procesos formales para la determinación de dependencias entre atributos y el cálculo de las claves primarias, es una práctica habitual entre los desarrolladores el utilizar un valor numérico (entero y potencialmente grande), como en el ejemplo anterior (**long id**). Además, para *tranquilidad* del programador, este valor puede ser autogenerado (por ejemplo, puede comenzar en 1, y asignar el siguiente valor a cada *fila* insertada).

Añadiendo la anotación `@GeneratedValue` a continuación de `@Id`, JPA asume que se va a generar un valor y a asignar el mismo antes de almacenar la instancia de la entidad. Existen diferentes estrategias de asignación:

- `GenerationType.AUTO`: Hiberante escoge la mejor estrategia en función del dialecto SQL configurado (es decir, dependiendo del RDBMS).
- `GenerationType.SEQUENCE`: Espera usar una secuencia SQL para generar los valores.
- `GenerationType.IDENTITY`: Hibernate utiliza una columna especial, autonumérica.
- `GenerationType.TABLE`: Hibernate usa una tabla extra en nuestra base de datos. Tiene una fila por cada tipo de entidad diferente, y almacena el siguiente valor a utilizar.

Generación de los ID antes o después de la inserción

Normalmente, un ORM optimizará las inserciones en una base de datos, agrupándolas por lotes. Esto supone que, en la realidad, es posible que no se realice la inserción al llamar al método `entityManager.persist(someEntity)`. Si esperamos a generar el ID a insertar los datos tiene como inconveniente que si llamamos a `someEntity.getId()`, obtendremos nulo. Por tanto, suele ser buena estrategia generar el ID antes de insertar, por ejemplo a través de una secuencia. Las columnas autoincrementales, los valores por defecto o las claves generadas por un trigger solo están disponibles después de la inserción.

6.3.2 Control de nombres

Por defecto, Hibernate toma una *estrategia de generación de nombres* para transformar el nombre de una clase (que en Java normalmente estará escrita en notación `UpperCamelCase`) al nombre de una tabla (por defecto, suele usar el mismo nombre en MAYÚSCULAS).

Se puede controlar el nombre de la tabla a través de la anotación `@Table(name= "NEWNAME")`.

```
@Entity
@Table(name= "MY_ENT")
public class MyEntity {
```

```
@Id
private long id;
```

```
public long getId() {
    return id;
}
```

```
}
```

6.4 Mapeo de valores

Cuando mapeamos una entidad, todos sus atributos son considerados persistentes por defecto. Las reglas por defecto son:

- Si la propiedad es un tipo primitivo, un envoltorio de un tipo primitivo (**Integer**, **Double**, ...), **String**, **BigInteger**, **BigDecimal**, **java.util.Date**, **java.util.Calendar**, **java.sql.Date**, **java.sql.Time**, **java.sql.Timestamp**, **byte[]**, **Byte[]**, **char[]**, o **Character[]** se persiste automáticamente con el tipo de dato SQL adecuado.
- Si es **java.io.Serializable**, se almacena con su representación serializada (esto no será lo que habitualmente deseemos).
- Si usamos **@Embeddable** (lo estudiaremos después), también lo persiste.

En otro caso, *lanzar  un error en la inicializaci n*. Hibernate escoge, dependiendo del dialecto configurado, la mejor correspondencia de tipos de dato en el RDBMS para los tipos Java que hayamos usado.

6.4.1 Anotaci n **@Column**

Esta anotaci n, sobre una propiedad, nos permitir  indicar algunas propiedades, entre las que se encuentran:

- **nullable**: nos permite indicar si la columna mapeada puede o no almacenar valores nulos. En la pr ctica, es como marcar el campo como requerido.

- **name**: permite modificar el nombre por defecto que tendrá la columna mapeada.
- **insertable**, **updatable**: podemos modificar si la entidad puede ser insertada, modificada, ...
- **length**: nos permite definir el número de caracteres de la columna.

¿Dónde anotar? ¿En las propiedades o en los getter?

Hibernate nos permite definir las anotaciones (**@Id**, **@Column**, ...) tanto sobre las propiedades como sobre los métodos getter (nunca sobre los setter). La pauta la marca la anotación **@Id**. Allá donde usemos esta anotación, marcaremos la estrategia a seguir.

6.4.2 Tipos temporales

Los tipos de datos temporales, de fecha y hora, tienen un tratamiento algo especial en Hibernate. Para un campo que contiene este tipo de información, se añade la propiedad **@Temporal**.

Esta anotación la podemos usar con los

tipos **java.util.Date**, **java.util.Calendar**, **java.sql.Date**, **java.sql.Time**, **java.util.Timestamp**. Hibernate también soporta los nuevos tipo de **java.time** disponibles en el JDK 8.

Como propiedad, podemos indicar que tipo de dato temporal vamos a querer usar a través del atributo **TemporalType**, teniendo disponibles **DATE**, **TIME**, **TIMESTAMP**.

```
@Entity
@Table(name= "MY_ENT")
public class MyEntity {
```

```
    @Id
    private long id;
```

```
    @Temporal(TemporalType.TIMESTAMP)
    private Date createdOn;
```

```
//...
```

```
}
```

Hibernate utiliza por defecto `TemporalType.TIMESTAMP` si no encuentra una anotación `@Temporal` sobre alguno de los tipos de datos definidos más arriba.

6.5 Tipos *embebidos*

En ocasiones, nos puede interesar tratar un grupo de atributos como si fueran uno solo. Un ejemplo clásico suele ser la dirección (nombre de la vía, número, código postal, ...). Para este tipo de situaciones tenemos la posibilidad de covertir una clases en `Embeddable`. Veamoslo con un ejemplo:

```
@Embeddable
```

```
public class Direccion {
```

```
    @Column(nullable=false)
    private String via;
```

```
    @Column(nullable=false, length = 5)
    private String codigoPostal;
```

```
    @Column(nullable=false)
    private String poblacion;
```

```
    @Column(nullable=false)
    private String provincia;
```

```
//...
```

```
}
```

```

@Entity
@Table(name="USERCONEMBEDD")
public class User {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    private String name;

    @Temporal(TemporalType.DATE)
    private Date birthDate;

    private Direccion address;

    // ...
}

```

De esta forma, Hibernate detecta que el campo **Direccion** es una clase **Embeddable**, y mapea las columnas a la tabla **USER**.

6.5.1 Sobrescritura con **@Embedded**

¿Qué pasaría si quisiéramos añadir dos direcciones a un usuario? Hibernate nos lanzará un error, indicando que no se soportan columnas con nombre repetido.

La solución la podemos aportar sobrescribiendo los atributos de la clase embebida, para que tengan otro nombre (o incluso otras propiedades)

```

@Entity
@Table(name="USERCONEMBEDD")
public class User {

```

```

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private long id;

//Otros atributos

@Embedded
@AttributeOverrides({
    @AttributeOverride(name = "via", column = @Column(name="VIA_FACTURACION")),
    @AttributeOverride(name = "codigoPostal", column =
@Column(name="CODIGOPOSTAL_FACTURACION", length=5)),
    @AttributeOverride(name = "poblacion", column =
@Column(name="POBLACION_FACTURACION")),
    @AttributeOverride(name = "provincia", column =
@Column(name="PROVINCIA_FACTURACION"))
})
private Direccion billingAddress;

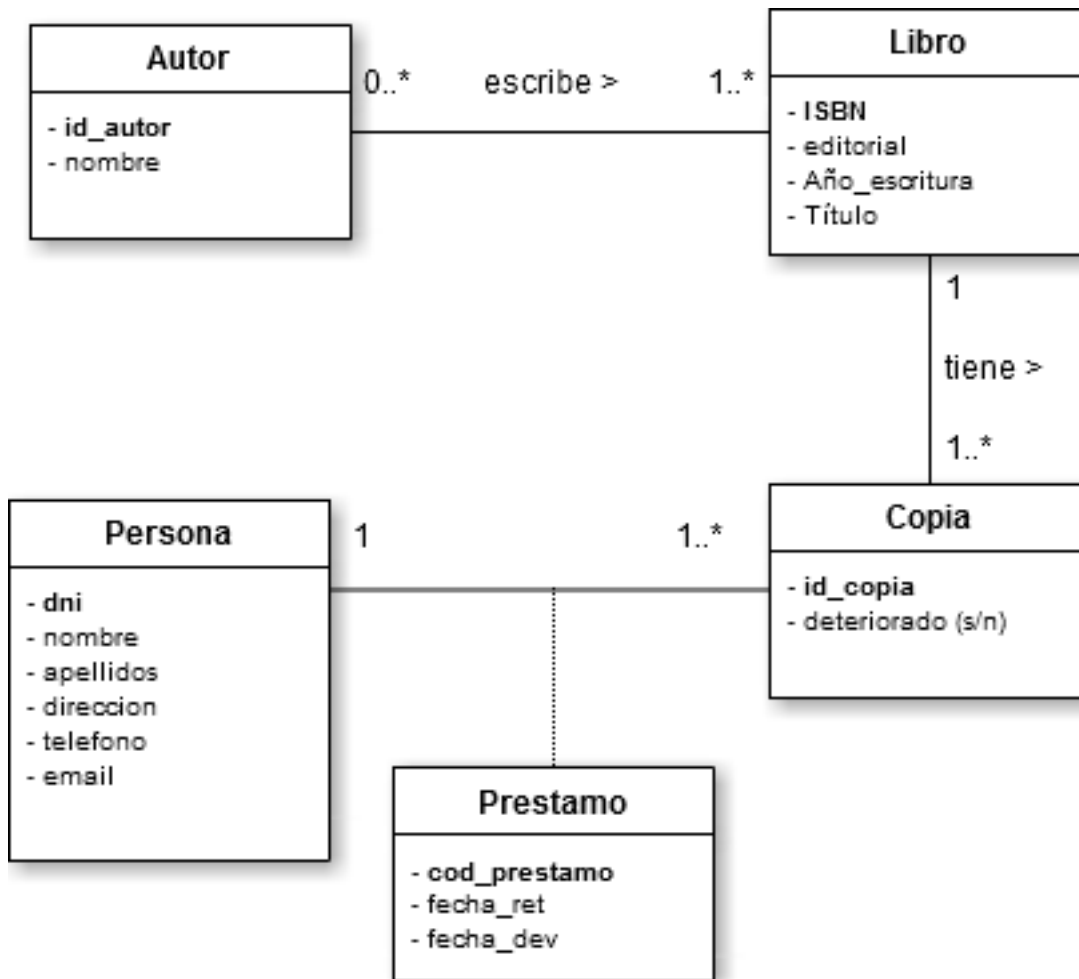
//...

```

La anotación **@Embedded** es útil cuando queremos mapear otras clases. El atributo **@AttributeOverrides** selecciona las propiedades que serán sobrescritas. El atributo **@AttributeOverride** indica el cambio que va a haber en un determinado atributo.

8.1 Asociaciones entre entidades

Normalmente, en todos los sistemas de información, las entidades del modelo de dominio suelen estar asociadas entre ellas, de forma que suele ser tan importante (o más) registrar la información asociada a la asociación como a la entidad.

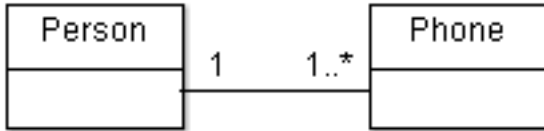


A nivel de base de datos, estas asociaciones suelen implementarse mediante claves externas. Una *foreign key* no es más que la presencia de la clave primaria de una tabla en otra tabla (puede ser reflexiva, y estar en la misma tabla), a la cual *apunta*.

JPA nos permite modelar estas asociaciones, tanto de forma *unidireccional* como de forma *bidireccional*. Veamos poco a poco todas ellas.

8.2. Asociaciones Many-To-One (Muchos a uno)

Se trata de la asociación más sencilla y común de todas. Este tipo de asociaciones se conoce en algunos contextos como una relación padre/hijo, donde el lado muchos es el *hijo* y el lado uno es el *padre*. La versión más sencilla de esta asociación es la **unidireccional**, en la que solo representamos la asociación en el lado muchos.



El código fuente sería el siguiente:

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private long id;

    private String name;

    public Person() { }

    public Person(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }
}
```



```
//resto de métodos...
```

```
}
```

```
@Entity
```

```
public class Phone {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private long id;
```

```
    private String number;
```

```
    @ManyToOne
```

```
    @JoinColumn(name = "person_id",
```

```
                foreignKey = @ForeignKey(name="PERSON_ID_FK"))
```

```
    private Person person;
```

```
    public Phone() { }
```

```
    public Phone(String number) {
```

```
        this.number = number;
```

```
    }
```

```
    public String getNumber() {
```

```
        return number;
```

```
    }
```

```
    public void setNumber(String number) {
```

```
        this.number = number;
```

```
    }
```

```
public Person getPerson() {  
    return person;  
}
```

```
public void setPerson(Person person) {  
    this.person = person;  
}
```

```
public long getId() {  
    return id;  
}
```

```
}
```

La anotación **@ManyToOne** nos permite indicar que una columna representa una asociación muchos-a-uno con otra entidad. El tipo de dato será el de la clase del lado *uno*. Complementariamente, podemos añadir las anotaciones **@JoinColumn**, que nos permite indicar el nombre de la columna que hará las funciones de clave externa, así como **@ForeignKey**, con la que podemos indicar el nombre de la restricción que se creará a nivel de base de datos (muy útil para depurar errores).

```
@ManyToOne  
@JoinColumn(name = "person_id",  
            foreignKey = @ForeignKey(name="PERSON_ID_FK"))  
private Person person;
```

Cada entidad tiene su ciclo de vida propio. Una vez que añadimos la asociación **@ManyToOne**, Hibernate establece el valor de la clave externa.

```
Person person = new Person("Pepe");  
em.persist( person );
```

```
Phone phone = new Phone("954000000");
```

```
phone.setPerson(person);  
em.persist(phone);
```

```
em.flush();  
phone.setPerson(null);
```

La asociación Uno-a-muchos nos permite enlazar, en una asociación padre/hijo, el lado *padre* con todos sus hijos. Para ello, en la clase debemos colocar una colección de elementos hijos.

Si la asociación **@OneToMany** no tiene la correspondiente asociación **@ManyToOne**, decimos que es **unidireccional**. En caso de que sí exista, decimos que es **bidireccional**.

9.1.1 One-To-Many unidireccional

Para representar esta asociación, añadimos en la clase padre un listado de elementos del tipo hijo.

```
@Entity  
public class Person {
```

```
    @Id  
    @GeneratedValue  
    private long id;
```

```
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)  
    private List<Phone> phones = new ArrayList<>();
```

```
    private String name;
```

```
    public Person() { }
```

```
    public Person(String name) {  
        this.name = name;
```

```

    }

    //Resto de métodos

    public List<Phone> getPhones() {
        return phones;
    }

}

```

```

@Entity
public class Phone {

    @Id
    @GeneratedValue
    private long id;

    private String number;

    public Phone() { }

    //Resto de métodos

}

```

Hibernate creará una tabla para cada entidad, y otra tabla para asociar ambas, añadiendo una restricción de unicidad al identificador del lado muchos. La anotación que define la asociación es **@OneToMany**. Vamos a repasar algunos elementos del código:

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
```

```
private List<Phone> phones = new ArrayList<>();
```

- La lista es inicializada al instanciar el objeto, para poder almacenar los elementos hijos.
- La opción `cascade = CascadeType.ALL` indica que las operaciones (formalmente llamadas *transiciones entre estados*) sobre el elemento padre se actualizarán hacia los hijos.
- La opción `orphanRemoval = true` indica que una entidad `Phone` será borrada cuando se elimine su asociación con la instancia de `Person`.

Para gestionar esta asociación *unidireccional*, debemos trabajar siempre con la lista de elementos que posea la entidad padre.

```
Person person = new Person("Pepe");  
Phone phone1 = new Phone("954000000");  
Phone phone2 = new Phone("600000000");
```

```
person.getPhones().add(phone1);  
person.getPhones().add(phone2);  
em.persist(person);  
em.flush();
```

```
person.getPhones().remove(phone1);
```

9.1.2 One-To-Many bidireccional

La asociación `@OneToMany` bidireccional necesita de una asociación `@ManyToOne` en el lado hijo. Aunque en los modelos estas asociaciones se representan bidireccionalmente, esta está representada solamente por una clave externa.

Toda asociación bidireccional debe tener un lado **propietario** (lado hijo). El otro lado vendrá referenciado mediante el atributo `mappedBy`.

```
@Entity  
public class Phone {
```

```
@Id
@GeneratedValue
private long id;
```

```
private String number;
```

```
@ManyToOne
private Person person;
```

```
public Phone() { }
```

```
//Resto de métodos
```

```
@Override
public int hashCode() {
    return Objects.hash(number);
}
```

```
@Override
public boolean equals(Object o) {
    if ( this == o ) {
        return true;
    }
    if ( o == null || getClass() != o.getClass() ) {
        return false;
    }
    Phone phone = (Phone) o;
    return Objects.equals( number, phone.number );
}
```

```
}
```

```

@Entity
public class Person {

    @Id
    @GeneratedValue
    private long id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();

    private String name;

    public Person() {
    }

    public Person(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Phone> getPhones() {

```

```
        return phones;
    }
```

```
    public void addPhone(Phone phone) {
        phones.add(phone);
        phone.setPerson(this);
    }
```

```
    public void removePhone(Phone phone) {
        phones.remove(phone);
        phone.setPerson(null);
    }
```

```
}
```

@NaturalId es una anotación propia de Hibernate. Representa un identificador único que, si bien no es el mejor candidato para clave primaria, es conveniente indicárselo a Hibernate, ya que lo puede usar para hacer más eficiente.

Como la clase **Phone** hace uso de una columna **@NaturalId** (haciendo el número de teléfono único), los métodos **equals()** y **hashCode()** pueden hacer uso de esta propiedad, y reducir la lógica de **removePhone()** a llamar al método **remove** de Java Collection.

Podemos destacar aquí la lógica de estos métodos:

```
    public void addPhone(Phone phone) {
        phones.add(phone);
        phone.setPerson(this);
    }
```

```
    public void removePhone(Phone phone) {
        phones.remove(phone);
        phone.setPerson(null);
    }
```


El primero, realiza la asociación de un teléfono una persona: se añade el teléfono a la lista de teléfonos de la persona, y se asigna la persona como propietaria del teléfono.

El segundo, realiza la operación inversa, sacando el teléfono de la lista de teléfonos de la persona, y asignando como nulo la persona que es *dueña* del teléfono.

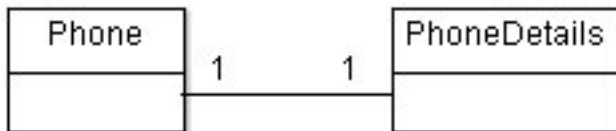
```
Person person = new Person("Pepe");  
Phone phone1 = new Phone("954000000");  
Phone phone2 = new Phone("600000000");
```

```
person.addPhone(phone1);  
person.addPhone(phone2);  
em.persist(person);  
em.flush();
```

```
person.removePhone(phone1);
```

La asociación bidireccional es más eficiente gestionando el estado de persistencia de la asociación. La eliminación de un elemento solo requiere de una sentencia UPDATE (poniendo a **NULL** la clave externa); además, si el ciclo de vida de la clase *hija* está enmarcado dentro del de su clase *padre* (es decir, que no puede existir sin ella) podemos anotar la asociación con el atributo **orphanRemoval**, de forma que al desasociar un hijo, automáticamente se eliminará la fila asociada.

Las asociaciones Uno-a-Uno también pueden ser unidireccionales o bidireccionales.



10.1.1 @OneToOne unidireccional

En una asociación uno-a-uno, solamente una instancia de una clase se asocia con una instancia de otra. Al usar el esquema *unidireccional*, tenemos que decidir un lado como **propietario**.

```
@Entity
public class Phone {

    @Id
    @GeneratedValue
    private long id;

    private String number;

    @OneToOne
    @JoinColumn(name = "details_id")
    private PhoneDetails details;

    public Phone() {
    }

    //Resto de métodos

    public PhoneDetails getDetails() {
        return details;
    }

    public void setDetails(PhoneDetails details) {
        this.details = details;
    }

    public long getId() {
```

```

        return id;
    }

}

@Entity
public class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

    private String provider;

    private String technology;

    public PhoneDetails() {
    }

    public PhoneDetails(String provider, String technology) {
        this.provider = provider;
        this.technology = technology;
    }

    public String getProvider() {
        return provider;
    }

    public String getTechnology() {
        return technology;
    }

    public void setTechnology(String technology) {

```

```

        this.technology = technology;
    }
}

```

En apariencia, el tratamiento de esta asociación es como una **@ManyToOne** (desde el punto de vista relacional, estamos usando una clave externa). Sin embargo, no tendría sentido que entender la entidad **PhoneDetails** como padre, y **Phone** como hija (no podemos tener unos detalles de teléfono sin un teléfono). Para solucionar esto, podemos usar una asociación bidireccional.

10.1.2 @OneToOne bidireccional

Para añadir el tratamiento bidireccional, añadimos un reflejo de la asociación en la clase **PhoneDetails**.

```

@Entity
public class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @OneToOne(mappedBy = "phone", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
    private PhoneDetails details;

    public Phone() {
    }

    //Resto de métodos

    public PhoneDetails getDetails() {
    }
}

```

```

        return details;
    }

    public void addDetails(PhoneDetails details) {
        details.setPhone(this);
        this.details = details;
    }

    public void removeDetails() {
        if (details != null) {
            details.setPhone(null);
            this.details = null;
        }
    }
}

```

@Entity

```
public class PhoneDetails {
```

```

    @Id
    @GeneratedValue
    private Long id;

```

```
    private String provider;
```

```
    private String technology;
```

```

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "phone_id")
    private Phone phone;

```

```
    public PhoneDetails() {
```

```
}
```

```
//Resto de métodos
```

```
}
```

En este caso, los métodos:

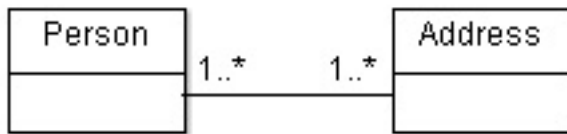
```
public void addDetails(PhoneDetails details) {  
    details.setPhone(this);  
    this.details = details;  
}
```

```
public void removeDetails() {  
    if (details != null) {  
        details.setPhone(null);  
        this.details = null;  
    }  
}
```

gestionan la lógica de la asociación. De esa forma, el ciclo de vida de uso sería como sigue:

```
Phone phone = new Phone("954000000");  
PhoneDetails details = new PhoneDetails("Movistar", "Fijo");  
phone.addDetails(details);  
em.persist(phone);
```

Todas las asociaciones muchos a muchos necesitan una tabla que realice de enlace entre ambas entidades asociadas. Veamos el tratamiento unidireccional y bidireccional.



11.2 @ManyToMany unidireccional

Tendremos que definir qué lado es el propietario de la asociación. En esa clase, incluimos una lista de elementos de la clase opuesta.

@Entity

```
public class Person {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
```

```
    private List<Address> addresses = new ArrayList<>();
```

```
    public Person() {
    }
```

```
    public List<Address> getAddresses() {
        return addresses;
    }
```

```
}
```

@Entity

```
public class Address {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String street;  
  
    private String number;  
  
    public Address() {  
    }  
  
    public Address(String street, String number) {  
        this.street = street;  
        this.number = number;  
    }  
  
    public Long getId() {  
        return id;  
    }  
  
    public String getStreet() {  
        return street;  
    }  
  
    public String getNumber() {  
        return number;  
    }  
  
}
```


Cuando una entidad se elimina de la colección **@ManyToMany**, Hibernate simplemente elimina la fila correspondiente de la tabla de enlace. Desafortunadamente, esta operación requiere eliminar todas las entradas asociadas para el padre seleccionado, y recrear aquellas que actualmente están en el contexto de persistencia.

Con un esquema unidireccional, un posible manejo de estas entidades sería el siguiente:

```
Person person1 = new Person();
Person person2 = new Person();

Address address1 = new Address( "Rue de l Percebe", "13" );
Address address2 = new Address( "Av. de la Constitución", "1" );

person1.getAddresses().add(address1);
person1.getAddresses().add(address2);

person2.getAddresses().add(address1);

em.persist(person1);
em.persist(person2);

em.flush();
```

11.3 @ManyToMany bidireccional

Una asociación bidireccional **@ManyToMany** tiene un lado *propietario* y un lado **mappedBy**. Para preservar la sincronización entre ambos, es buena práctica añadir métodos *helper* para manejar las asociaciones (como en ocasiones anteriores).

```
@Entity
public class Person {
```

```
    @Id
```

```
@GeneratedValue  
private Long id;
```

```
@NaturalId  
private String registrationNumber;  
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})  
private List<Address> addresses = new ArrayList<>();
```

```
public Person() {  
}
```

```
public Person(String registrationNumber) {  
    this.registrationNumber = registrationNumber;  
}
```

```
public List<Address> getAddresses() {  
    return addresses;  
}
```

```
public void addAddress(Address address) {  
    addresses.add( address );  
    address.getOwners().add( this );  
}
```

```
public void removeAddress(Address address) {  
    addresses.remove( address );  
    address.getOwners().remove( this );  
}
```

```
@Override  
public boolean equals(Object o) {  
    if ( this == o ) {  
        return true;  
    }
```

```

    }
    if ( o == null || getClass() != o.getClass() ) {
        return false;
    }
    Person person = (Person) o;
    return Objects.equals( registrationNumber, person.registrationNumber );
}

```

```

@Override
public int hashCode() {
    return Objects.hash( registrationNumber );
}

```

```

}

```

```

@Entity
public class Address {

```

```

    @Id
    @GeneratedValue
    private Long id;

```

```

    private String street;

```

```

    private String number;

```

```

    private String postalCode;

```

```

    @ManyToMany(mappedBy = "addresses")
    private List<Person> owners = new ArrayList<>();

```

```

    public Address() {
    }

```

```
public Address(String street, String number, String postalCode) {  
    this.street = street;  
    this.number = number;  
    this.postalCode = postalCode;  
}
```

```
public Long getId() {  
    return id;  
}
```

//Resto de métodos

```
@Override  
public boolean equals(Object o) {  
    if ( this == o ) {  
        return true;  
    }  
    if ( o == null || getClass() != o.getClass() ) {  
        return false;  
    }  
    Address address = (Address) o;  
    return Objects.equals( street, address.street ) &&  
        Objects.equals( number, address.number ) &&  
        Objects.equals( postalCode, address.postalCode );  
}
```

```
@Override  
public int hashCode() {  
    return Objects.hash( street, number, postalCode );  
}
```

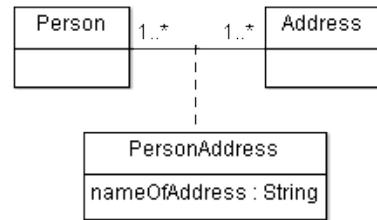
```
}
```

Cabe destacar la lógica de los métodos *helper*, que nos permiten establecer las referencias (o eliminarlas) entre ambas clases.

```
public void addAddress(Address address) {  
    addresses.add( address );  
    address.getOwners().add( this );  
}  
  
public void removeAddress(Address address) {  
    addresses.remove( address );  
    address.getOwners().remove( this );  
}
```

11.4 @ManyToMany con atributos extra

Un caso que suele presentarse en muchas ocasiones con las asociaciones muchos a muchos es que necesitamos añadir un atributo que no es de ninguna de las dos entidades, sino de la asociación en sí.



Algunos autores llaman a este tipo de asociación con atributos **clase de asociación**.

Como este nuevo atributo no es ni de la dirección, ni de la persona, no lo podemos colocar en ninguna de las dos entidades, por lo que tenemos que seguir los siguientes pasos:

- Generar una nueva entidad **PersonAddress**
- Romper la asociación **@ManyToMany** en ambos extremos en dos asociaciones que den el mismo resultado: **@ManyToOne** + **@OneToMany**.
- Manejar de forma conveniente la clave primaria de esta nueva entidad. Al ser una clave primaria compuesta, necesitaremos de una clase extra, **PersonAddressId**, y de la anotación **@IdClass**, para poder manejarla.

@Entity

```
public class Address {
```

```
    @Id
    @GeneratedValue
    private Long id;
```

```
    private String street;
```

```
    private String number;
```

```
    private String postalCode;
```

```
    @OneToMany(mappedBy = "address", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<PersonAddress> owners = new ArrayList<>();
```

```
    //Resto de métodos
```

```
    public List<PersonAddress> getOwners() {
        return owners;
    }
```

```
    //Resto de métodos
```

```
}
```

@Entity

```

public class Person {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String registrationNumber;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<PersonAddress> addresses = new ArrayList<>();

    //Resto de métodos

    public List<PersonAddress> getAddresses() {
        return addresses;
    }

    //Resto de métodos
}

```

Como podemos ver, las entidades **Person** y **Address** tienen ahora una asociación **@OneToMany** en lugar de una **@ManyToMany**. ¿Dónde está el resto de esa asociación? Veamos la entidad **PersonsAddress**.

```

@Entity
@IdClass(PersonAddressId.class)
public class PersonAddress {

    @Id
    @ManyToOne
    @JoinColumn(

```



```
        name="person_id",
        insertable = false, updatable = false
    )
    private Person person;
```

```
@Id
@ManyToOne
@JoinColumn(
    name="address_id",
    insertable = false, updatable = false
)
    private Address address;
```

```
    private String nameOfAddress;
```

```
    public PersonAddress() { }
```

```
    public PersonAddress(Person person, Address address) {
        this.person = person;
        this.address = address;
```

```
    }
```

```
    public PersonAddress(Person person, Address address, String name) {
        this(person, address);
        this.nameOfAddress = name;
    }
```

```
    //Resto de métodos
```

```
}
```

```

public class PersonAddressId implements Serializable {

    private Long person;
    private Long address;

    public PersonAddressId() {

    }

    //Getters, setters, equals y hashCode

}

```

La anotación `@Id` solo está permitida, en primera instancia, para claves primarias simples, por lo que una entidad, por norma, no puede tener dos atributos anotados con `@Id`. Para poder manejar una clave primaria **compuesta**, JPA nos obliga a utilizar alguna estrategia diferente, como `@IdClass` o `@EmbeddId`. Nosotros optamos por la primera. Para ello, creamos una clase que tendrá los campos que conforman la clave primaria y que cumplirá con las siguientes características:

- Debe ser una clase pública
- Debe tener un constructor sin argumentos
- Debe implementar `Serializable`
- No debe tener **clave primaria propia**
- Debe implementar los métodos `equals` y `hashCode`.

Para poder manejar ahora el ciclo de vida de esta nueva entidad, podemos modificar los métodos *helper* que estábamos usando:

```

public void addAddress(Address address, String name) {
    PersonAddress personAddress = new PersonAddress( this, address, name );
    addresses.add( personAddress );
    address.getOwners().add(personAddress);
}

```

```

public void removeAddress(Address address) {
    PersonAddress personAddress = new PersonAddress( this, address);
    address.getOwners().remove( personAddress );
    addresses.remove( personAddress );
}

```

14.1 Generación automática del esquema

Como hemos podido comprobar en las lecciones anteriores, Hibernate permite autogenerar el esquema de nuestra base de datos a partir del mapeo que hayamos realizado (ya sea a través de XML o a través de anotaciones).

Aunque la generación automática del esquema es muy útil para el testeo o el prototipado, en un entorno de producción es menos usual, siendo más flexible manejar el esquema a través de scripts sql incrementales.

A nivel de configuración, tenemos una propiedad JPA, llamada `hibernate.hbm2ddl.auto`, que nos permite generar el DDL para un buen número de opciones:

Opción	Propósito
<code>none</code>	Valor por defecto. No realiza ninguna acción
<code>create-only</code>	Solamente realiza el proceso de creación de la base de datos.
<code>drop</code>	Realiza solamente el borrado de la base de datos.
<code>create</code>	Realiza un borrado de la base de datos, y posteriormente su creación.

<code>create-drop</code>	Elimina el esquema y lo vuelve a crear al crear el contexto de persistencia o el <code>SessionFactory</code> . Adicionalmente, también lo elimina cuando uno u otro se cierran.
<code>validate</code>	Valida el esquema de la base de datos
<code>update</code>	Actualiza la base de datos, con los cambios necesarios.

15.1 Introducción

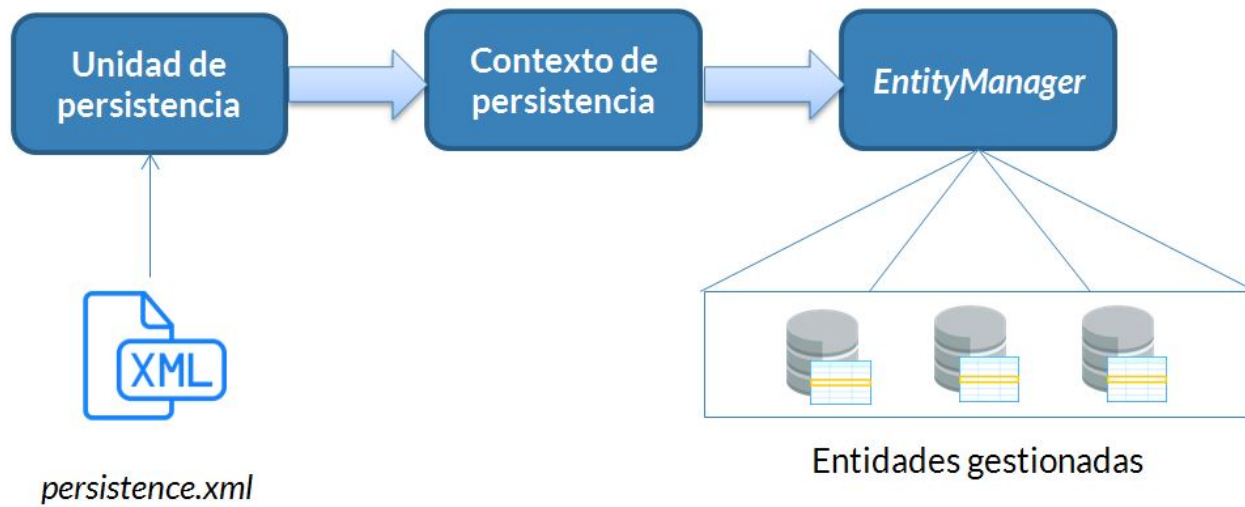
Hasta ahora hemos utilizado algunos conceptos sobre persistencia sin llegar a entrar en profundidad en conocerlos. Vamos a profundizar ahora un poco en ellos.

15.2 Unidad de persistencia y contexto de persistencia

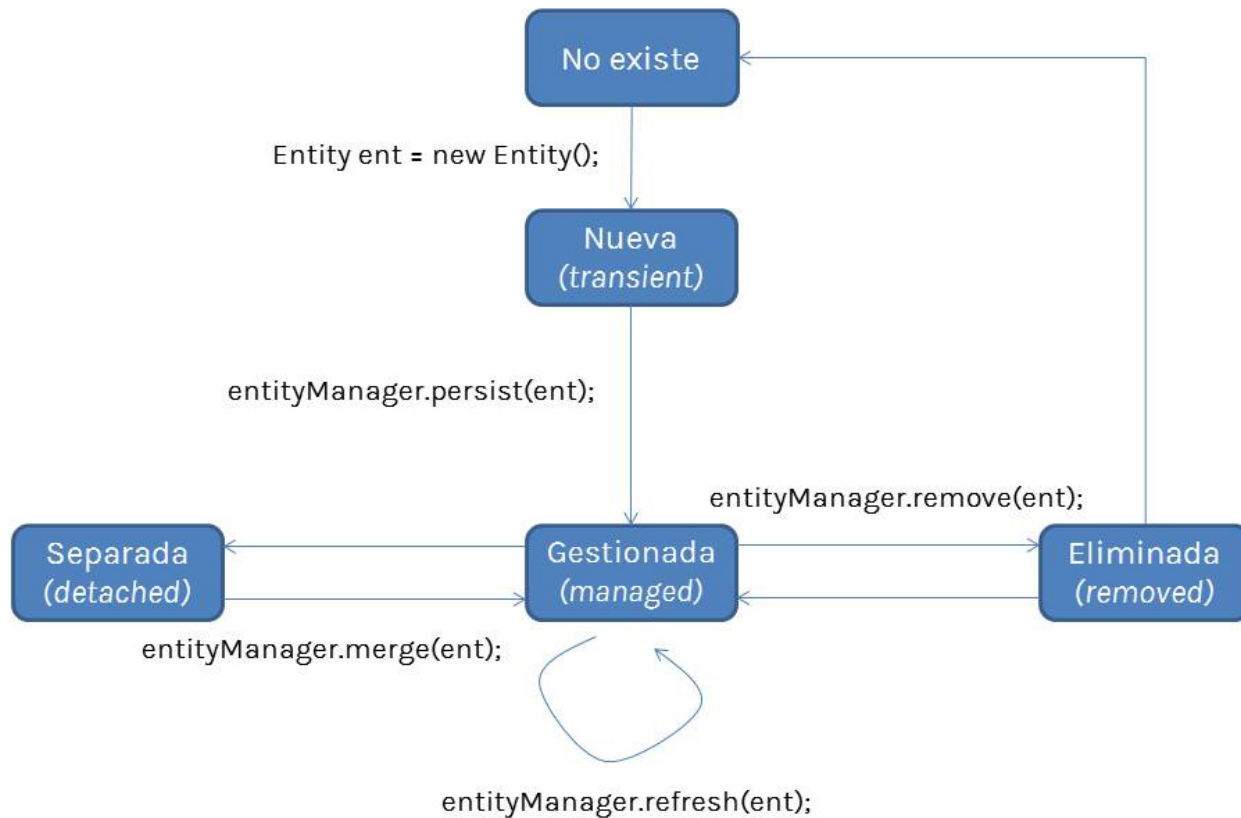
Una **unidad de persistencia** representa un conjunto de entidades que pueden ser mapeadas en una base de datos, así como la información necesaria para que la aplicación se pueda conectar a la misma. Toda esta información viene definida en un fichero llamado *persistence.xml*.

Dentro de este fichero podemos definir una o más unidades de persistencia, incluyendo siempre un nombre único y una fuente de datos (*datasource*).

Un **contexto de persistencia** representa un conjunto de entidades que se encuentran gestionadas *en un momento dado*. Podríamos decir que es algo así como una **instancia** de una unidad de persistencia.



Tal y como vimos en apartados anteriores, una entidad gestionada por un contexto de persistencia puede estar en alguno de los siguientes estados:

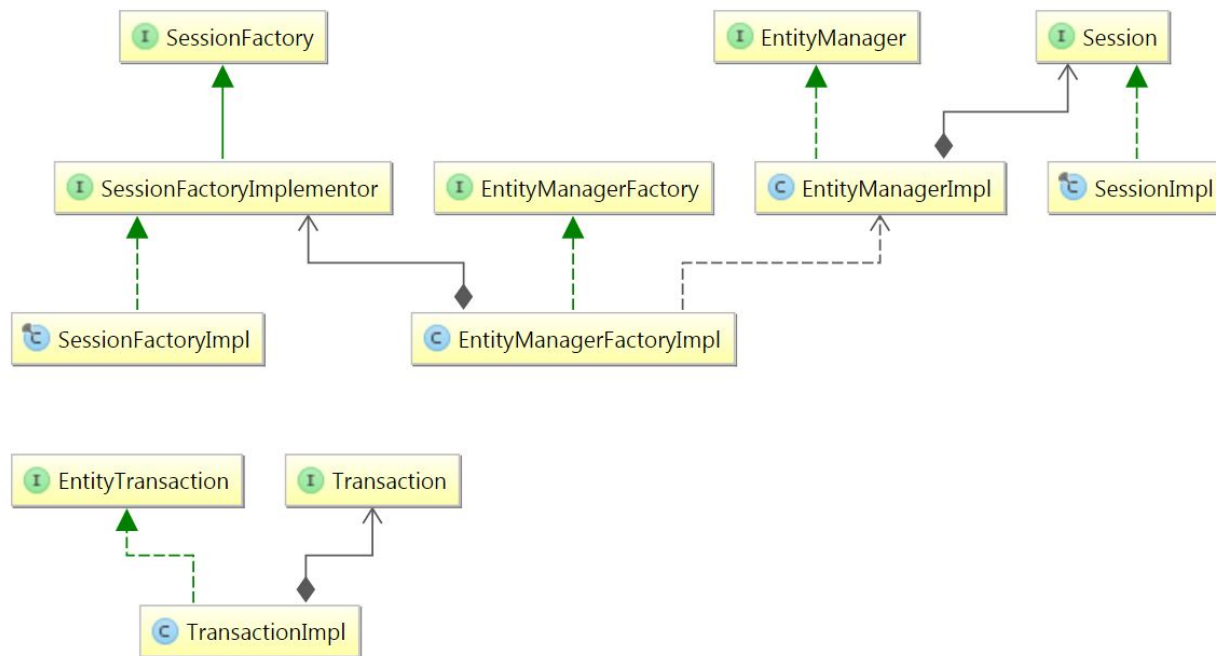


La mayoría de los métodos del gestor de entidades van dirigidos a mover las entidades entre esos diferentes estados.

15.3 API de JPA y API de Hibernate

Hasta ahora hemos venido maridando el uso de JPA con Hibernate. Cada uno de ellos tiene su propio *API*, ofreciendo de una forma diferente las mismas funcionalidades.

En el siguiente diagrama de clases podemos apreciarlo:



Los diferentes esquemas de trabajo de JPA (a través de un *EntityManagerFactory* y un *EntityManager*) y de Hibernate nativo (a través de un *SessionFactory* y un *Session*) tienen en el fondo una misma implementación.

JPA define un mecanismo sencillo y útil para poder acceder al API subyacente de Hibernate:

```

Session session = entityManager.unwrap( Session.class );
SessionImplementor sessionImplementor = entityManager.unwrap(SessionImplementor.class );
SessionFactory sessionFactory =
entityManager.getEntityManagerFactory().unwrap(SessionFactory.class );
  
```

15.4 Gestión de entidades

15.4.1 Persistencia de una entidad

Al crear una nueva entidad a través del operador **new**, esta estaría en estado *nuevo*. Para hacerla persistente, tenemos que asociarla a un *EntityManager*.

```
Person person = new Person();  
person.setId( 1L );  
person.setName("John Doe");
```

```
entityManager.persist( person );
```

Si la entidad tuviera asociado un mecanismo para la generación del Id, no tendríamos que asignárselo manualmente, y se generaría automáticamente al persistir el objeto.

15.4.2 Eliminación de una entidad

Las entidades también pueden ser *eliminadas*. Para ello, tan solo tenemos que

```
entityManager.remove( person );
```

15.4.3 Obtención de una referencia a una entidad sin inicializar sus datos

En ocasiones puede interesarnos hacer referencia a una entidad manejada por el contexto de persistencia, pero sin tener que cargar todos sus datos (por ejemplo, porque sean muchos o muy pesados). El caso más común es para establecer una asociación con otra entidad.

JPA nos permite hacerlo a través del método `getReference`:

```
Book book = new Book();  
book.setAuthor( entityManager.getReference( Person.class, personId ) );
```

Este código asume que se está usando la carga *lazy*.

15.4.4 Obtención de una referencia a una entidad inicializando sus datos

Las llamadas a este método son muy comunes, por ejemplo para mostrar todos los datos de una entidad en la interfaz de usuario.

```
Person person = entityManager.find( Person.class, personId );
```


En caso de que la entidad exista en el contexto de persistencia, nos devolvería una instancia de ella; si no existe, nos devuelve **null**.

15.4.5 Modificación de una entidad que ya está manejada/persistida

Los cambios sobre las entidades manejadas por JPA serán detectados automáticamente, y persistidos cuando el contexto de persistencia sea *flushed*.

```
Person person = entityManager.find( Person.class, personId );
person.setName("John Doe");
entityManager.flush();
```

15.4.6 Refresco del estado de una entidad

Podemos recargar una entidad desde el contexto de persistencia en cualquier momento.

```
Person person = entityManager.find( Person.class, personId );

entityManager.createQuery( "update Person set name = UPPER(name)" ).executeUpdate();

entityManager.refresh( person );
assertEquals("JOHN DOE", person.getName() );
```

Esta operación es útil en algunos casos, como por ejemplo si sabemos que el estado en la base de datos ha cambiado desde que leímos la entidad, o si alguno de los valores ha sido generado a nivel de base de datos con un disparador o *trigger*.

15.4.7 Trabajo con entidades *detached* (separadas)

Una entidad puede pasar a estado *detached* por varias razones:

- Porque el contexto de persistencia se haya cerrado.
- Porque el contexto de persistencia se haya refrescado (**clear**).
- Llamando a los métodos adecuados, como **detach** (o con Hibernate nativo **evict**).

Las entidades en este estado no van a funcionar de la misma manera: sus cambios no se almacenarán en la base de datos, no podemos refrescarlas, etc...

Para volver a incluir una entidad dentro del contexto de persistencia, podemos realizar una operación **merge**. Esta operación toma los datos de la entidad (*detached*) que recibe como argumento, obteniendo como resultado una nueva entidad, copia de la anterior, y que si estará manejada por el contexto de persistencia:

```
Person person = entityManager.find( Person.class, personId );
//Al limpiar el entityManager la entidad se convierte en detached
entityManager.clear();
person.setName( "Mr. John Doe" );

person = entityManager.merge( person );
```

15.5 Transiciones en cascada

Las transiciones en cascada son una herramienta potente, que nos permiten propagar un cambio de estado desde una entidad *padre* a otras entidades *hijas* relacionadas con la primera. Dentro del interfaz **javax.persistence.CascadeType** tenemos definidos varios tipos:

- **ALL**: se propagan todos los cambios de estados.
- **PERSIST**: se propagan las operaciones de persistencia.
- **MERGE**: se propagan las operaciones **merge**.
- **REMOVE**: se propagan las eliminaciones.
- **REFRESH**: se propagan las actualizaciones.
- **DETACH**: se propagan las separaciones.

Podemos ilustrar las operaciones en cascada con el siguiente código:

@Entity

```

public class Person {

    @Id
    private Long id;

    private String name;

    @OneToMany(mappedBy = "owner", cascade = CascadeType.ALL)
    private List<Phone> phones = new ArrayList<>();

    //getters y setters

    public void addPhone(Phone phone) {
        this.phones.add( phone );
        phone.setOwner( this );
    }
}

```

```

@Entity
public class Phone {

    @Id
    private Long id;

    @Column(name = "`number`")
    private String number;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person owner;

    //getters y setters
}

```

Veamos algunas de las operaciones más comunes.

15.5.1. PERSIST

Nos permite almacenar una entidad *hija* a través de su *padre*.

```
Person person = new Person();  
person.setId(1L);  
person.setName("Pepe Pérez");
```

```
Phone phone = new Phone();  
phone.setId(1L);  
phone.setNumber("954000000");
```

```
person.addPhone(phone);
```

```
entityManager.persist( person );
```

15.5.2. MERGE

Nos permite volver a incluir dentro del contexto de persistencia una entidad *hija* a través de su *padre*.

```
Phone phone = entityManager.find( Phone.class, 1L );  
Person person = phone.getOwner();
```

```
person.setName("Pepe Pérez");  
phone.setNumber("954000000");
```

```
entityManager.clear();
```

```
entityManager.merge(person);
```

15.5.3. REMOVE

Nos permite eliminar una entidad *hija* a través de su *padre*.

```
Person person = entityManager.find( Person.class, 1L );
```

```
entityManager.remove( person );
```

15.6 *Fetch plans* (planes de carga de datos)

Como hemos visto, JPA+Hibernate, a través del contexto de persistencia, ponen a nuestra disposición los datos almacenados en una base de datos. Pero, ¿es necesario que tengamos siempre todos los datos de las entidades que manejamos? Podemos intervenir en este proceso, indicando el tipo de *fetching* que queremos realizar.

Tenemos a nuestra disposición dos esquemas de trabajo:

- EAGER: Hibernate cargará todos los datos de las entidades que sean necesarias (incluidas las de las entidades hijas si hay asociaciones).
- LAZY: Con el modo *perezoso* solo se cargarán los datos cuando estos sean realmente necesarios (es decir, cuando se vayan a utilizar).

16.1 Introducción


Cualquier RDBMS define una serie de operaciones para manejar los datos mediante DML (*Data Manipulation Language*).

Normalmente, estas operaciones son *INSERT*, *UPDATE* y *DELETE*, y trabajan a nivel de fila. Sin embargo, a la hora de programar una aplicación, nos encontramos con que determinadas operaciones de la lógica de negocio deberían ser tratadas como atómicas (indivisibles), de forma que **o se hacen completas, o no deben hacerse**. Los RDBMS suelen permitir realizar estos conjuntos de operaciones atómicas a través de transacciones.

Supongamos que un cliente de un banco quiere realizar un traspaso de dinero a otro cliente de ese mismo banco. Esta operación (muy simplificada) consiste en actualizar el saldo de emisor, decrementándolo; y actualizar el saldo de receptor, aumentándolo.

TITULAR	SALDO
Pepe	500
Juan	300

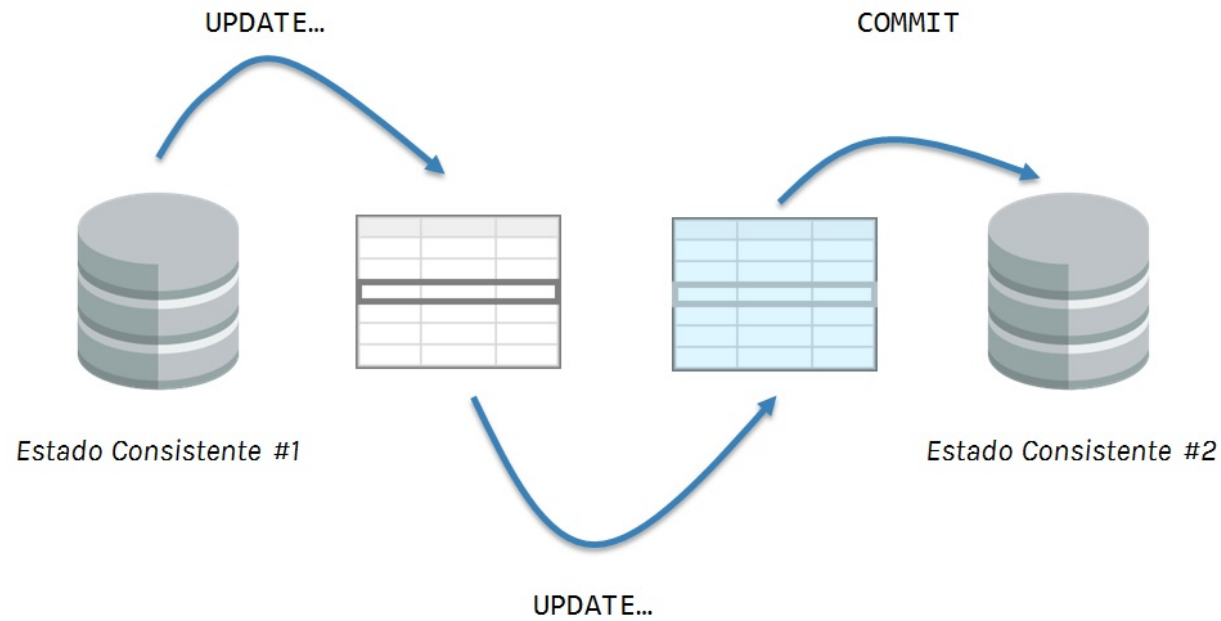
Transferencia
De Pepe a Juan
de 100€



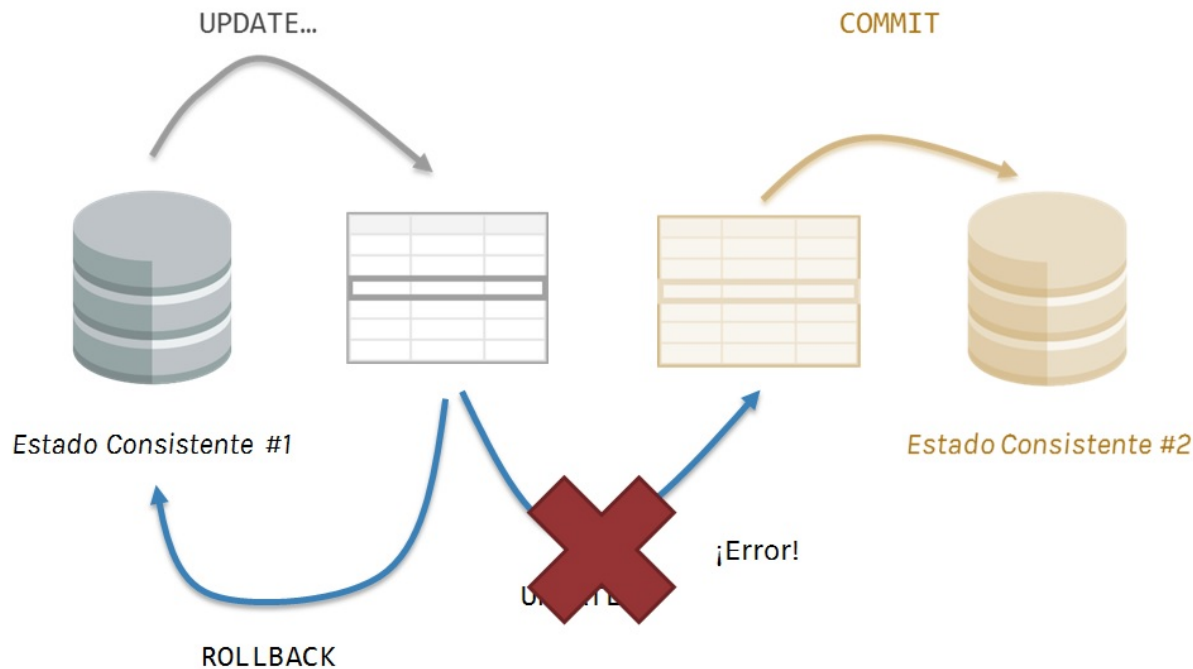
TITULAR	SALDO
Pepe	400
Juan	400

A nivel de base de datos, esta operación debería realizarse con dos sentencias UPDATE; sin embargo, debería tratarse de forma **atómica**. O se ejecutan las dos sentencias, o no se ejecuta ninguna (imaginad la cara de Pepe si le restaran saldo en su cuenta, y la cara de Juan al no recibir su transferencia).

Para confirmar una transacción, lo hacemos mediante la instrucción SQL **COMMIT**.



Si durante el transcurso de alguna de estas dos operaciones hubiera algún problema, deberíamos deshacer los cambios realizados. A esta operación se le conoce como **ROLLBACK**.



Hibernate y JPA nos permiten, como no podía ser de otra manera, el manejo de transacciones, para poder gestionar todo esto desde nuestra aplicación.

16.2 Transacciones *Resource Local* o *JTA*

Resource Local es un tipo de transaccionalidad mediante la cual delegamos la responsabilidad de realizar las transacciones en el programador. Las transacciones se obtienen a partir del *EntityManager*, y son de

tipo `javax.persistence.EntityTransaction`. Podemos manejar las transacciones a través de los métodos `begin()`, `commit()` y `rollback()`. Su uso está orientado sobre todo a aplicaciones de escritorio o que no se ejecutan en un servidor de aplicaciones.

JTA, en cambio, es parte de la especificación *Java EE*. Se trata de un estándar (que debe ser implementado por el servidor de aplicaciones que utilicemos), que centraliza la gestión de las transacciones. Utiliza el tipo `javax.transaction.UserTransaction` que también tiene los métodos `begin()`, `commit()` y `rollback()`. Su uso está orientado a las aplicaciones que se ejecutan en un servidor de aplicaciones, como las aplicaciones Web MVC. La configuración de JPA, Hibernate y JTA es muy cómoda cuando trabajamos con otros frameworks, como puede ser Spring Web MVC y Spring Boot, quien realiza la configuración por nosotros.

A continuación podemos ver un ejemplo de control de una transacción con *Resource Local*:

```
@Entity
public class UserAccount {

    @Id
    private int id;

    @Column
    private String name;

    @Column
    private double balance;

    public UserAccount() {

    }

    //Getters y Setters

}

public class App {
```

```

static EntityManagerFactory emf;

static EntityManager em;

public static void main(String[] args) {

    // Configuramos el EMF a través de la unidad de persistencia
    emf = Persistence.createEntityManagerFactory("Transacciones");

    // Generamos un EntityManager
    em = emf.createEntityManager();

    // Construimos un objeto de tipo User
    UserAccount user1 = new UserAccount();
    user1.setId(1);
    user1.setName("Pepe");
    user1.setBalance(500);

    insertUserAccount(user1);

    // Construimos otro objeto de tipo User
    UserAccount user2 = new UserAccount();
    user2.setId(2);
    user2.setName("Juan");
    user2.setBalance(300);

    insertUserAccount(user2);

    makeTransfer(user1, user2, 100.0);

    // Cerramos el EntityManager

```

```
em.close();  
emf.close();
```

```
}
```

```
public static void insertUserAccount(UserAccount userAccount) {  
    try {  
        //Iniciamos una transacción  
        em.getTransaction().begin();  
        //Persistimos los datos  
        em.persist(userAccount);  
        //Commiteamos la transacción  
        em.getTransaction().commit();  
        System.out.println("El objeto ha sido dado de alta correctamente. Muchas  
gracias.");  
    } catch (Exception e) {  
        System.out.println("El objeto no ha sido dado de alta correctamente. Disculpe las  
molestias");  
        System.err.println(e.getMessage());  
        if (em.getTransaction().isActive()) {  
            em.getTransaction().rollback();  
        }  
    }  
}
```

```
public static void makeTransfer(UserAccount origen, UserAccount destino, double cantidad)  
{  
    try {  
        //Iniciamos una transacción  
        em.getTransaction().begin();
```

```

        origen.setBalance(origen.getBalance() - cantidad);
        destino.setBalance(destino.getBalance() + cantidad);

        em.persist(origen);
        em.persist(destino);

        //Commiteamos la transacción
        em.getTransaction().commit();

        System.out.println("El traspaso ha sido realizado correctamente. Muchas
gracias.");
    } catch (Exception e) {
        System.out.println("El traspaso no ha sido realizado correctamente. Disculpe las
molestias");
        System.err.println(e.getMessage());
        if (em.getTransaction().isActive()) {
            em.getTransaction().rollback();
        }
    }
}
}
}

```

17.2 JPQL básico

Por lo que hemos trabajado anteriormente, solo hemos podido obtener, a través del *EntityManager*, entidades a partir de su identificador, y solamente una entidad por consulta. Para aquellos que ya conozcan SQL esto resultará un enfoque muy pobre. JPQL nos va a permitir realizar consultas en base a muchos criterios, así como obtener más de un valor por consulta.

La consulta JPQL más básica tiene la siguiente estructura:

```
SELECT c FROM Customer c
```

Esta sentencia obtiene todas las instancias de **Customer** que existan en la base de datos. La expresión puede parecer un poco extraña la primera vez que se ve, pero es muy sencilla de entender. Las palabras SELECT y FROM tienen un significado similar a las sentencias homónimas del lenguaje SQL, indicando que se quiere seleccionar (SELECT) cierta información desde (FROM) cierto lugar. La segunda c es un alias para la clase Customer, y ese alias es usado por la primera c (llamada expresión) para acceder a la clase (tabla) a la que hace referencia el alias, o a sus propiedades (columnas). El siguiente ejemplo nos ayudará a comprender esto mejor:

```
SELECT c.customerName FROM Customer c
```

El alias c nos permite utilizar la expresión c.CustomerName para obtener los nombres de todos los clientes almacenados en la base de datos. Las expresiones JPQL utilizan la notación de puntos, pudiendo acceder a objetos anidados:

```
SELECT e.propiedad.anidada.masanidada FROM Entidad e
```

Al igual que en SQL, JPQL nos permite obtener más de una propiedad:

```
SELECT c.customerName, c.city, c.country FROM Customer c
```

17.2.1 Consultas con parámetros

JPQL nos permite la inclusión de parámetros en base a un índice y en base a un nombre:

```
SELECT e FROM Employee e WHERE e.jobTitle = :jobTitle
```

```
SELECT o FROM Order o WHERE o.orderDate BETWEEN ?1 AND ?2 AND status = ?3
```

17.2.2 Consultas con salida ordenada

También nos permite la ordenación de los resultados, al estilo SQL. Este orden puede ser ascendente (ASC) o descendente (DESC).

```
SELECT o FROM Order o ORDER BY o.orderDate DESC
```

17.3 JPA Query

JPA nos provee de dos interfaces, `javax.persistence.Query` y `javax.persistence.TypedQuery`, que se obtienen directamente desde el `EntityManager`. Para ello, podemos usar el método `EntityManager#createQuery`. Para consultas con nombre (las veremos más adelante), podemos usar el método `EntityManager#createNamedQuery`.

```
Query query = em.createQuery(  
    "select c from Customer c"  
);
```

```
List<Customer> listCustomer = (List<Customer>) query.getResultList();
```

Para aquellas consultas que devuelven más de un resultado, tenemos a nuestra disposición el método `getResultList()`; si la consulta devuelve un solo resultado, tendremos entonces que llamar a `getSingleResult()`.

Para asignar los parámetros dinámicos a las consultas, tenemos a nuestra disposición diferentes versiones del método `setParameter`, que acepta tanto un índice como un nombre de parámetro, y un objeto para utilizar en la consulta.

```
Query query = em.createQuery(  
    "SELECT e FROM Employee e WHERE e.jobTitle = :jobTitle"  
);
```

```
query.setParameter("jobTitle", "Sales Rep");
```

```
Query query = em.createQuery(  
    "SELECT o FROM Order o WHERE o.orderDate BETWEEN ?1 AND ?2 AND status = ?3 ORDER BY  
    o.orderDate DESC"  
);
```

```
Calendar calendar = Calendar.getInstance();  
calendar.set(2003, 0, 1);
```

```
query.setParameter(1, calendar.getTime());
```

```
calendar.set(2003, 5, 30);  
query.setParameter(2, calendar.getTime());
```

```
query.setParameter(3, "Shipped");
```

17.4 Joins con JPQL

La operación JOIN es una operación básica en SQL, y sirve para realizar una reunión conveniente entre dos tablas a partir de una clave externa.

17.4.1 Joins explícitos

Como no podía ser de otra manera, JPQL permite realizar los mismos JOIN que en SQL (JOIN, INNER JOIN, LEFT I RIGHT JOIN), si bien uno de los más interesantes el *JOIN FETCH*. Este join sobrescribe la forma en que se recuperan determinadas asociaciones, evitando las N+1 consultas y sustituyéndolas por un JOIN.

```
Query query = em.createQuery(  
    "select c "  
    + "from Customer c "  
    + "left join fetch c.employee "  
);
```

Para que un JOIN FETCH sea válido, la asociación debe estar en alguna de las clases que estén descritas en la cláusula SELECT

17.4.2 Joins implícitos

Un JOIN implícito se realiza siempre a través de alguna de las asociaciones de una entidad, navegando a través de sus propiedades.

```
TypedQuery<Customer> query = em.createQuery(  
    "select c "  
    + "from Customer c "
```

```
+ "where c.employee = :employee", Customer.class);
```

```
query.setParameter("employee", em.find(Employee.class, 1370));
```

El uso de TypedQuery en el ejemplo anterior es meramente ilustrativo. Se puede realizar un JOIN implícito con Query

17.5 Consultas de actualización

JPQL también nos permite lanzar consultas de actualización de datos. La estructura sintáctica de las consultas es muy similar a la de SQL nativo:

```
UPDATE Entity e
SET e.prop1 = newValue1, e.prop2 = newValue2, ...
WHERE ...
```

```
DELETE FROM Entity e
WHERE ...
```

Para invocarlas, tenemos que utilizar el método `executeUpdate` de la interfaz `Query`.

```
em.getTransaction().begin();
```

```
//UPDATE
```

```
//Incremento del 10% en el límite de crédito a todos los clientes
```

```
int numUpdateResults = em.createQuery(
    "update Customer c "
    + "set c.creditLimit = c.creditLimit * 1.1")
    .executeUpdate();
System.out.println("Número de registros afectados: " + numUpdateResults);
```

```
//DELETE
```

```
Date date = null;
```



```
try {
    date = new SimpleDateFormat("dd/MM/yyyy").parse("06/06/2003");

    int numDeleteResults = em.createQuery(
        "delete from Payment p "
        + "where p.paymentDate = :fecha")
        .setParameter("fecha", date)
        .executeUpdate();

    System.out.println("Número de registros afectados: " + numDeleteResults);

} catch (ParseException e) {
    System.err.println("Error en el parseo de la fecha");
}
```

```
em.getTransaction().commit();
```

JPQL no permite la ejecución de sentencias *INSERT*. Sin embargo, HQL si que lo permite:

```
int insertedEntities = session.createQuery(
    "insert into Partner (id, name) " +
    "select p.id, p.name " +
    "from Person p ")
    .executeUpdate();
```

17.6 NamedQueries

Las consultas con nombre son un tipo de consultas especiales ya que, una vez que son definidas, no pueden ser modificadas. Son leídas y transformadas en SQL durante la inicialización del contexto de persistencia. Por ello, son más eficientes y ofrecen un rendimiento mayor. Se suelen definir mediante anotaciones en las clases entidad, si bien también pueden declararse en XML.

El asistente de Eclipse que nos ha generado las entidades desde las tablas nos ha incluido una sentencia por defecto en cada una de ellas:

```
@Entity
@Table(name="customers")
@NamedQuery(name="Customer.findAll", query="SELECT c FROM Customer c")
public class Customer implements Serializable {
    //Resto del código
}
```

Como podemos comprobar, la anotación `@NamedQuery` requiere de dos argumentos: el nombre que le vamos a dar a la consulta, y la definición de la consulta en sí.

Para ejecutar esta consulta, tenemos que utilizar el método `createNamedQuery` de *EntityManager*.

```
Query query = em.createNamedQuery("Customer.findAll");
```

Se pueden añadir más de una consulta con nombre, a través de la anotación `@NamedQueries`. Las consultas también pueden recibir parámetros.

```
@Entity
@Table(name="customers")
@NamedQueries({
    @NamedQuery(name="Customer.findAll", query="SELECT c FROM Customer c"),
    @NamedQuery(name="Customer.findByName", query="SELECT c FROM Customer c WHERE c.customerName LIKE :name"),
    @NamedQuery(name="Customer.findByEmployee", query="SELECT c FROM Customer c WHERE c.employee = :employee"),
})
public class Customer implements Serializable {

}
```

17.7 Consultas con SQL nativo

JPA e Hibernate también permiten la ejecución de SQL nativo (en particular, el dialecto del RDBMS que estemos usando). Esto es muy útil cuando nuestro sistema tiene funcionalidades específicas, o si nuestra experiencia en SQL nos permite implementar sentencias que sean realmente eficientes.

17.7.1 Consultas escalares

La consulta más básica es aquella en la que obtenemos una lista de valores en un `Object[]`.

```
List<Object[]> customers = entityManager.createNativeQuery(
    "SELECT * FROM customers" )
    .getResultList();

List<Object[]> employees = entityManager.createNativeQuery(
    "SELECT employeeNumber, firstName, lastName FROM employees" )
    .getResultList();

for(Object[] employee : employees) {
    Number employeeNumber = (Number) employee[0];
    String firstName = (String) employee[1];
    String lastName = (String) employee[2];
}
```

17.7.2 Consultas de entidades

También podemos obtener entidades desde consultas nativas:

```
List<Customer> customers = entityManager.createNativeQuery(
    "SELECT * FROM customers", Customer.class )
    .getResultList();
```

Dado que la entidad *Customer* tiene asociaciones y algunas de ellas están mapeadas con *fetch EAGER*, al ejecutar este ejemplo, JPA+Hibernate rescatan el resto de datos necesarios a través de consultas JPQL.

17.7.3 Manejo de asociaciones

JPA e Hiberante también nos permite manejar asociaciones con SQL nativo.

Por ejemplo, si nuestra entidad tiene una asociación *Many-To-One* con otra, al usar SQL nativo tenemos que devolver la clave externa para que no se produzca un error:

```
List<Employee> employeesList = em.createNativeQuery(
    "SELECT employeeNumber, lastName, firstName, extension, email, officeCode,
    reportsTo, jobTitle FROM employees",
    Employee.class).getResultList();
```

17.7.4 Consultas SQL con nombre

Por último, decir que también podemos definir consultas con nombre, como ocurría con JPQL. Estas se definen a través de la anotación `@NamedNativeQuery`.

```
@Entity
@Table(name="employees")
@NamedQuery(name="Employee.findAll", query="SELECT e FROM Employee e")
@NamedNativeQuery(name="Employee.nativeFindAll", query="SELECT * FROM employees",
    resultClass=Employee.class)
public class Employee implements Serializable {
    //Resto de código
}
```

Su uso también es muy sencillo

```
List<Employee> employeesList = em.createNamedQuery("Employee.nativeFindAll").getResultList();
```

}

TEMA 5: SERVERS STATELESS: OAUTH I SERVEIS WEB (SOAP I API REST)

BIBLIOGRAFIA

Xavier Torrens. Apunts Es Liceu de Desenvolupament Web en Entorn Servidor dels cursos acadèmics 2017/18 i 2018/19.

Open Webinars (2019). <http://openwebinars.com>. Data consulta: 13/07/2019

IOC. Desenvolupament Web en Entorn Servidor (2019)