PAU CASESNOVES
Centre Integrat de Formació Professional

Desenvolupament d'aplicacions web

Programació

UT 6.1: Herència i polimorfisme

Joan Pons Tugores

Índex de continguts

Herència

Ens podem trobar que tenguem una classe dissenyada i ens adonem que en necessitam una altre de molt semblant, igual que la que tenim amb uns quants mètodes o atributs més, o amb alguns mètodes que necessiten tenir un comportament diferent del que ja tenim.

Podem crear aquesta classe nova des del principi, però, a part de la feina que comporta, tendrem dues classes totalment deslligades i que fàcilment poden perdre aquesta semblança que haurien de tenir.

Una altra opció és utilitzar el concepte d'herència: Crear la nova classe a partir de l'existent de forma que qualsevol canvi que es faci a l'original es reflecteixi a la nova. Només hi haurem d'afegir aquells membres que necessitem a la nova i que no té l'original.

Relació *és-un*

Que una classe comparteixi alguns atributs amb una altra no vol dir que justifiqui crear una relació d'herència entre elles. Podem tenir la classe *Alumne* i la classe *Mascota*, però el fet que tots dos tenguin un atribut *nom*, no vol dir que una de les classes pugui heretar d'una altra.

Normalment es diu que per crear una relació d'herència entre dues classes hi ha d'haver entre elles una relació del tipus ***és-un***, és a dir, un *Alumne* no és una *Mascota*, ni una *Mascota* és un *Alumne*, per tant no podem crear la relació d'herència entre ells.

En canvi, si tenim una classe *Vehicle* i una altra classe *Cotxe* que comparteixen atributs, com que podem dir que un *Cotxe* és un *Vehicle* podem crear la classe Cotxe a partir de la classe Vehicle.

És a dir, no basta que dues classes comparteixin atributs per establir una herència entre elles, sinó que a més, els conceptes que representen han de tenir una relació es-un entre ells.

Herència

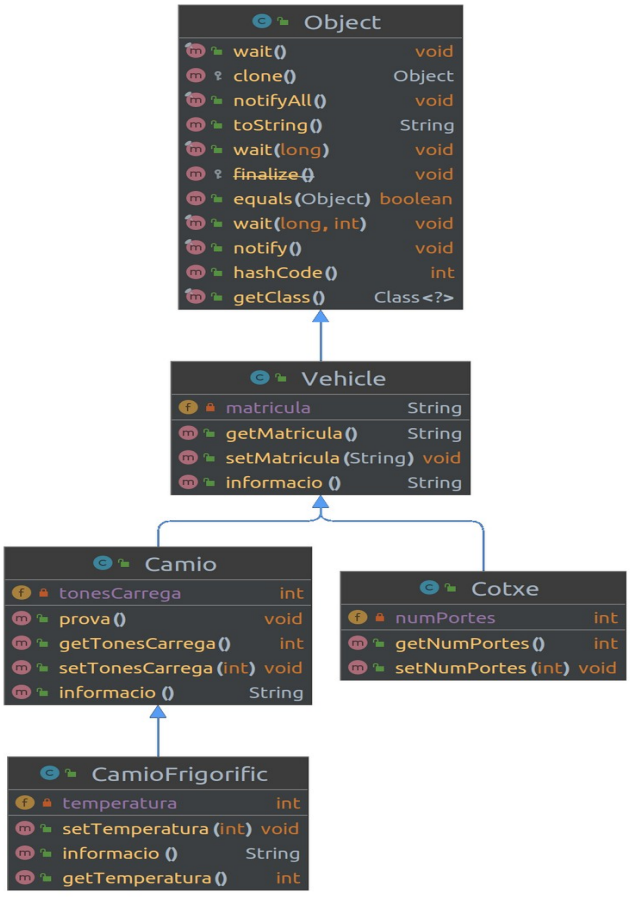
La idea de l'herència és molt simple però molt potent: Si tenim una classe A semblant a una altra B podem crear A a partir de B de forma que tengui tots els membres que té B.

- **Superclasse**: la classe de la que es deriven altres classes.
- **Subclasse**, classe derivada o classe filla: classe creada a partir d'una altra.

Una classe pot tenir n subclasses, però una subclasse només pot tenir una superclasse. En Java no existeix l'herència múltiple.

Una classe pot tenir una superclasse, que pot tenir una superclasse, que pot ...

Totes les classes en Java descendeixen d'Object. Per tant, les classes formen una jerarquia en forma d'arbre, on l'arrel és Object.



Superclasse

Qualsevol classe pot tenir subclasses, no hem de fer res especial per marcar-la com a tal.

Superclasse només és un concepte, no implica cap comanda ni res especial. Si tenim una classe A, en crear una subclasse seva B direm que A és la superclasse de B, però el codi d'A seguirà sent el mateix, no l'hem de modificar per res.

Podem dir que hi ha dues maneres de trobar una relació d'herència entre classes quan dissenyam una aplicació (No són dos tipus d'herència):

- Per especialització: Tenim una classe i veim que en necessitam definir casos especialitzats d'aquesta classe. Per exemple tenim la classe Vehicle i ens adonam que necessitam tractar de forma separada camions i autocars, però volem aprofitar el codi de Vehicle i que es mantingui una relació entre ells.
- Per generalització: A la nostra aplicació tenim una classe Camió i una altre Autocar i ens adonam que comparteixen gran part del codi i que a més tenim processos que han de tractar-los indistintament. Ens podem plantejar crear la classe Vehicle.

Subclasse

Per crear una subclasse ho feim de la següent manera:

```
public class Cotxe extends Vehicle{
```

La paraula clau ***extends*** indica que és una subclasse i *Vehicle* ens diu quina és la seva superclasse.

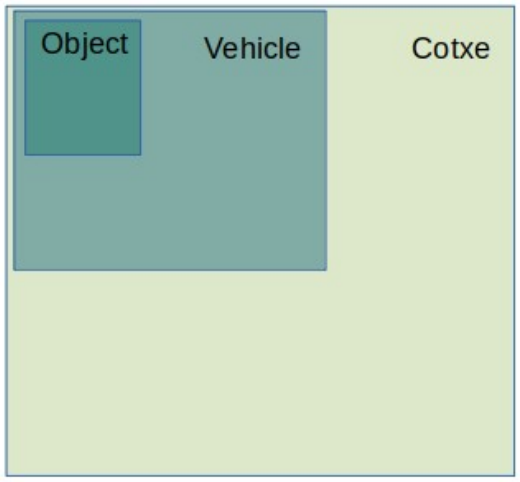
Una subclasse hereta tots els membres *public* i *protected* de la superclasse i si es troben al mateix paquet, els que no tenen qualificador d'accés.

Els atributs heretats es poden utilitzar directament.

Podem declarar atributs nous o amb el mateix nom que un de la superclasse (ocultar-lo).

En crear un objecte de la classe *Cotxe* conté un objecte de la classe Vehicle i de totes les seves superclasses fins arribar a *Object*.

Això vol dir que encara que des de cotxe no tenguem accés als membres privats de *Vehicle* o de *Object*, hi són.



A una subclasse:

- Els mètodes heretats es poden utilitzar directament.
- Podem escriure nous mètodes:
  - Totalment nous.
  - Amb la mateixa signatura que un d'instància de la superclasse: **sobreescriptura**.
  - Amb la mateixa signatura que un estàtic de la superclasse: **ocultació**.
  - Nous constructors. Criden al de la superclasse implícitament(si la classe té el constructor sense paràmetres) o explícita (super).

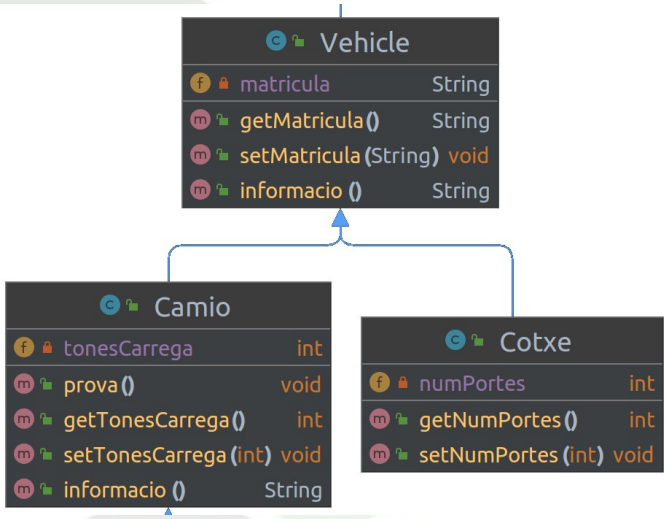
Els **membres privats** de la superclasse no són heretats ni accessibles directament, només hi podem accedir a través d'altres membres accessibles.

Els **constructors** tampoc s'hereten. El constructor de la subclasse ha de cridar un constructor de la superclasse. Si no ho feim explícitament amb super el compilador intentarà cridar el constructor sense arguments. Si la superclasse no el té donarà error.

Sobreescriptura i ocultació

Sobreescriptura

Un mètode d'una subclasse en sobreescriu un de la seva superclasse si té la mateixa signatura i tipus de retorn. Encara que tractem l'objecte de la subclasse com un de la superclasse, s'executarà el de la subclasse.



Podem pensar en la sobreescriptura com una substitució, el mètode de la subclasse substitueix el de la superclasse.

@Override és una anotació que indica al compilador que estam sobreescrivint un mètode. Si aquest mètode no té la mateixa signatura i tipus de retorn que el de la superclasse tendrem un error de compilació. Convé utilitzar-la.

La classe *Camio* sobreescriu el mètode *informació* de la classe *Vehicle*. Te ún mètode amb la mateixa signatura que la superclasse.

Ocultació

- Si a una subclasse declaram un **atribut amb el mateix nom que un de la superclasse** l'oculta, és a dir, a la subclasse només podreu utilitzar el de la subclasse, només tendrem accés al de la superclasse utilitzant super. Si feim un càsting d'un objecte de la classe derivada cap a la superclasse, llavors podrem accedir a l'atribut de la superclasse i no al de la subclasse.

- Si a una subclasse declaram un **mètode estàtic** amb la mateixa signatura que un mètode estàtic de la superclasse, l'oculta.

Polimorfisme

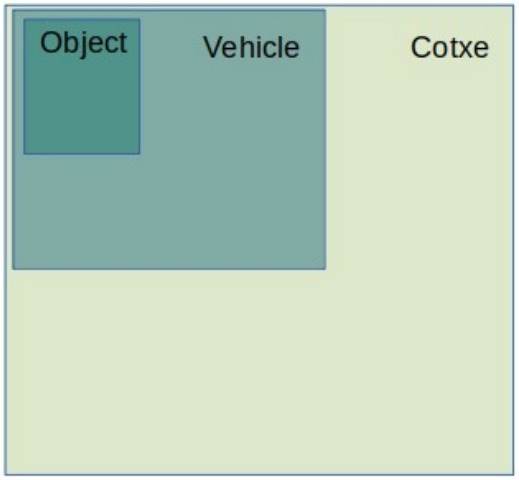
Un objecte és de la classe que s'ha instanciat, però també és un objecte de totes les seves superclasses i es pot utilitzar per tot on s'esperi un objecte d'aquestes classes.

Per exemple, si tenim la jerarquia *Objecte* → *Vehicle* → *Cotxe* a qualsevol lloc on s'esperi un *Object* o un *Vehicle* hi podem passar un objecte *Cotxe*.

Polimorfisme: La capacitat d'un objecte per comportar-se de formes diferents.

Un objecte d'una subclasse pot ser considerat com un objecte de qualsevol de les seves superclasses fent un càsting implícit o explícit. Llavors no podrem executar cap mètode ni utilitzar cap atribut que hagi estat afegit per la subclasse.

```
Cotxe seat=new Cotxe();
Vehicle v=seat;
```



El codi anterior és correcte. v serà un vehicle.

Ho podem fer perquè un *Cotxe* és un *Vehicle*. L'objecte realment és un cotxe, però *v* només ens dona accés als membres de Vehicle i les seves superclasses.

No hem perdut informació, si feim

```
Cotxe c=(Cotxe) v;
```

Recuperam tot el que té l'objecte *seat* original.

Ara hem de fer el càsting explícit perquè un Vehicle no és un cotxe.

Si l'aplicació necessita per exemple llistar les dades de Vehicle d'objectes Cotxe, Camió, ...qualsevol de les seves subclasses, tenim dues opcions:

- sobrecarregar el mètode amb tantes versions com subclasses hi hagi, on cada un tengui un paràmetre de la subclasse. Mala idea.

```
public void mostrarVehicle(Cotxe vehicle){ ... }
public void mostrarVehicle(Camio vehicle){ ... }
public void mostrarVehicle(Autocar vehicle){ ... }
```

- crear un sol mètode amb un paràmetre del tipus de la superclasse, Vehicle. Aquest mètode acceptarà objectes de qualsevol de les subclasses.

```
public void mostrarVehicle(Vehicle vehicle){
}
```

Per altre banda, si tenim dos objectes d'una superclasse tampoc tenen perquè comportar-se de la mateixa manera. Si un s'ha creat com a objecte de la superclasse i l'altre com un objecte de la subclasse, en executar els mètodes sobreescrits tendran comportaments diferents.

Per exemple si la classe Vehicle té el mètode:

```
public void identificar(){
        System.out.println("Som un Vehicle");
}
```

I una subclasse el sobreescriu, per exemple cotxe

```
@Override
public void identificar(){
        System.out.println("Som un Cotxe");
}
```

Llavors si tenim el mètode:

```
public void mostrarVehicle(Vehicle vehicle){
        vehicle.identificar();
}
```

Què mostrarà? Depèn:

- Si li passam un *Cotxe* mostrarà *Som un Cotxe.*
- Si li passam un *Camio* i camió no ha sobreescrit el mètode mostrarà *Som un Vehicle.*

super, getClass() i instanceof

super

Permet invocar un mètode sobreescrit o un camp ocult de la classe pare. S'utilitza quan el mètode de la subclasse ha de realitzar les mateixes accions que les del mètode que sobreescriu i unes quantes més.

```
super.mètode();
x=super.atribut;
```

Es obligatori en els constructors si la superclasse no té un constructor sense paràmetres. A un constructor la crida al super ha de ser la primera instrucció.

```
public Cotxe(...){
    super(...);
```

getClass()

És un mètode de la classe *Object*, per tant el tenen tots els objectes que poguem crear. Ens diu de quina classe és realment l'objecte, amb quina classe feren el *new*, encara que s'hagin fet càstings.

```
Vehicle vehicle=new Cotxe(...);
System.out.println(vehicle.getClass()); //Mostra Cotxe
```

instanceof

És un operador del llenguatge. Permet comprovar si un objecte és una instància d'una determinada classe. Un objecte serà instància de la classe a partir de la qual va ser creat i de totes les seves superclasses. Només es pot utilitzar amb classes de la mateixa jerarquia d'herència.

if (fruita instanceof Poma) System.out.println("És una poma!");

Si el codi està ben fet no hauria de fer falta utilitzar getClass ni instanceof.

final

Un mètode es pot declarar final de forma que les subclasses no el puguin sobreescriure. D'aquesta manera controlam el codi que s'executa per a totes les subclasses.

```
public final void metodeImmutable() {
    System.out.println("No es pot sobreescriure aquest mètode");
}
```

Una classe es pot declara final de forma que no es poden crear subclasses a partir d'ella. Pot servir per evitar que s'incloguin classes nostres a altres projectes i obtinguin les modificacions que necessiten fent subclasses de les nostres i modificant-les.

```
public final class Net extends Fill {
}
public class Besnet /*extends Net*/{
    //No és possible que sigui subclasse de Net perquè és final
}
```

abstract

Classes abstractes

Pot ser que no ens interessi que es puguin crear objectes d'una determinada classe:

- Perquè és la **classe arrel d'una jerarquia** i només volem objectes de les subclasses. Per exemple, si tenim *Vehicle*, *Cotxe*, *Moto*, ... podem considerar que a l'aplicació no hi pot haver *Vehicles* perquè tots han de ser o una *Moto* o un *Cotxe*, ...
- Perquè és una **classe d'utilitat**, que només té membres de classe i per tant no cal crear-ne objectes, com per exemple *UtilitatsConsola* o *OrdenacioICerca*.

Per aconseguir-ho, basta posar *abstract* a la definició de la classe.

```
public abstract class Meva{
  private int num;
```

Mètodes abstractes

Pot ser que cada subclasse tengui una determinada implementació d'un cert mètode i que aquest mètode no tengui sentit a la superclasse. Tot i això ens pot interessar que el mètode sigui a la superclasse per poder-lo utilitzar amb el polimorfisme. En aquest cas es pot fer el mètode abstracte a la superclasse

```
public abstract void metodeAImplementarPerLesSubclasses();
```

Definir un mètode abstracte té dues implicacions:

- La classe ha de ser abstracte: no podem tenir objectes amb mètodes sense implementar
- Les subclasses han d'implementar el mètode o també han de ser abstractes.