DAM - M03Block2 - Exercici_de_repas - teoria_002

Contingut:

1. Encapsulació
2. Herència
    - 2.1. Classes abstractes
    - 2.2. Mètodes abstractes
    - 2.3. Interfaces
    - 2.4. Inicialització de subclasses (super)
    - 2.5. Sobreescriptura de mètodes (superclasse --> subclasse, + super)
    - 2.6. Sobreescriptura del mètode finalize()
    - 2.7. Sobreescriptura del mètode equals()
    - 2.8. Sobreescriptura del mètode toString()

1. Encapsulació

Un dels objectius de la programació orientada a objectes és l'encapsulació de dades i de mètodes de manera que els programadors usuaris d'una classe només poden accedir a les dades mitjançant els mètodes que la mateixa classe proporciona.

Amb l'encapsulació de dades i de mètodes s'aconsegueix:

- Protegir les dades de modificacions impròpies.
- Facilitar el manteniment de la classe, ja que si per algun motiu es creu que cal efectuar alguna reestructuració de dades o de funcionament intern, es podran efectuar els canvis pertinents sense afectar les aplicacions desenvolupades (sempre que no es modifiquin els prototipus dels mètodes existents).

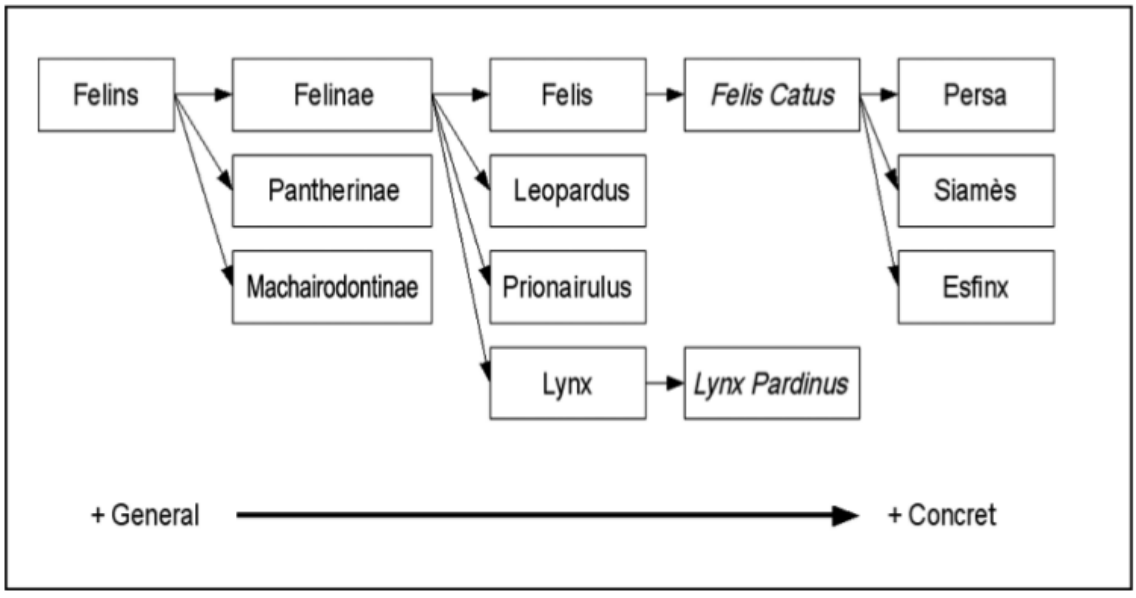
Així, doncs, ens interessa ocultar les dades i, potser, alguns mètodes.

2. Herència

Una eina molt útil per fer aplicacions escalables és l'herència. Aquest concepte està força vinculat a la taxonomia, la ciència de la classificació mitjançant l'organització dels elements en grups segons les seves relacions de similitud. Això és molt útil en el camp de la biologia en la classificació dels éssers vius.

Crear taxonomies és útil perquè permet definir les característiques d'un conjunt d'elements partint d'una descripció general i, a poc a poc, concretar fins a arribar als elements més concrets. En la figura, es parteix des de l'esquerra, amb el major grau de generalització, i a mesura que ens desplacem cap a la dreta s'arriba a un major grau de concreció. Dins un mateix nivell de la taxonomia, cada conjunt es distingeix dels altres per certes diferències, però tots comparteixen les propietats del nivell anterior.

FIGURA 1.1. Una taxonomia per classificar felins



S'anomena **herència** la capacitat de definir una classe tan sols especificant-ne les diferències

respecte a una altra classe prèviament definida. En la resta d'aspectes, es considera que es comporta exactament igual.

Donada una classe concreta dins la jerarquia, tota classe més general és la seva **superclasse**, mentre que en el cas d'una classe més concreta, es tracta de la **subclasse**. Exemple: en la foto anterior (figura 1.1.), **Felins** és la superclasse de **Felinae**, **Pnatherinae** i **Machairodontinae** i aquestes 3 són subclasses de **Felins**.

Un aspecte molt important i útil que aporta l'herència és la capacitat de disposar d'objectes que pertanyen a diverses classes alhora: objectes amb més d'un tipus. Donat un objecte d'una classe concreta, aquest objecte no solament serà del tipus que marca la seva classe, sinó que també ho serà de totes les seves superclasses. Exemple: en la foto anterior (figura 1.1.), una instància de la classe **Lynx** serà del tipus **Lynx**, **Felinae** i **Felins**.

Perquè la classe B hereti la classe A hem de declarar-la **class B extends A**. D'aquesta manera qualsevol instància de la classe B podrà accedir a les propietats/atributs i mètodes de la classe A sempre que no siguin **private**.

El constructor de la classe base (la superclasse) se invoca abans que el constructor de la classe derivada (la subclasse).
Els constructors no s'hereten.

2.1. Classes abstractes

És possible definir classes que especifiquin un conjunt d'atributs i d'operacions, però de les quals no existeixin objectes. És a dir, classes que no es poden instanciar. Pel que fa a la implementació, qualsevol intent d'instanciar-les provocarà un error. Aquestes classes s'anomenen **classes abstractes** i el seu objectiu és facilitar la creació de jerarquies d'herència.
Una **classe abstracta** no es pot instanciar, ja que representa un concepte abstracte. Està pensada per operar com a superclasse d'altres classes en qualsevol nivell dins una jerarquia.

Atès que una classe abstracta no es pot instanciar, mai no hi haurà objectes d'aquesta classe sobre els quals cridar operacions (cridar als seus mètodes). Tot i així, especificar-hi operacions (implementar els mètodes) no és un fet il·lògic, ja que cal recordar que totes són heretades per les respectives subclasses i, si aquestes no són abstractes, es poden cridar. Aquest és, en definitiva, l'objectiu final de definir una classe abstracta.
Una classe abstracta pot tenir mètodes normals i abstractes.

Les propietats/atributs poden ser static, final o static final amb qualsevol modificador d'accés (public, private, protected).

Poden tenir constructors. Una subclasse podria instanciar-se cridant al seu propi constructor i al constructor de la superclasse (amb **super(llistaDeParàmetres)**) per així poder inicialitzar els seus atributs i els de la superclasse.

2.2. Mètodes abstractes

Si volem que qualsevol subclasse que hereti d'una superclasse estigui obligada a implementar el seu propi codi en algun dels mètodes de la superclasse, s'haurà de declarar el mètode amb la paraula **abstract** en la superclasse.
És un mètode declarat en la superclasse que estarà buit (sense codi) i que serà implementat obligatòriament per les classes filles i per tant haurà de ser **public** o **protected** en la superclasse.
En una classe abstracte, per a definir que un mètode és abstracte li hem de posar la paraula reservada **abstract**. Recordar que el fet que una classe sigui declarada com abstracte només és per assegurar-nos de que no es creïn instàncies d'ella. Una classe abstracta pot tenir mètodes normals i abstractes.

2.3. Interfaces

Una classe només pot heretar d'1 classe. Per a poder heretar de més d'1 classe hem de fer servir les interfaces ja que no hi ha límit en la quantitat d'interfaces que pot heretar (implementar) una classe.

Una interface pot heretar de més d'una interface.

Una interface és un paquet amb propietats/atributs constants i mètodes abstractes (només estan els noms dels mètodes, no tenen codi).

No hi pot haver cap instància d'una interface. Les classes que implementin una interface són compatibles amb el tipus de la interface, es a dir, es poden crear objectes de la classe però que siguin del tipus de la interface.
Exemple: una **interface A** i una **classe B** que té com a interface a **A**. Es pot declarar un objecte de la classe **B** de 2 maneres:

- B objecteB = new B(); . D'aquesta manera podem accedir a tots els atributs i mètodes de la **classe B** i als atributs de la **interface A**.
- A objecteB = new B(); . D'aquesta manera només podem accedir als atributs i mètodes de la **classe B** que estiguin definits en la **interface A**.

Tots els mètodes són automàticament abstractes i publics. Qui ha d'implementar el codi dels mètodes són les classes que implementin la interface.

Totes les propietats/atributs són constants estàtiques (public static final).

No poden tenir constructors.

Si volem que diverses classes facin servir el mateix codi en un mètode a llavors millor que heretin d'una classe abstracta (no es pot instanciar) on el mètode no serà abstracte i tindrà el codi que volem que executin totes les subclasses. Sinó, millor fer servir les interfaces on el mètode és automàticament abstracte i per tant cada subclasse haurà d'implementar el seu propi codi per aquest mètode.

Perquè la classe A faci servir la interface B hem de declarar-la **class A implements B {...}**.
Exemple:

```
public interface Tripulants {

    public String agafarTarees();
    public String agafarInforme();
}
```

```
public class Capita implements Tripulants {

    public String agafarTarees(){
        return "Aquestes són les tarees del capità";
    }

    public String agafarInforme() {
        return "Informe del capità.";
    }

}
```

Les interfaces també es poden assignar a un paquet. La inexistència del modificador d'accés public fa que la interface sigui accessible a nivell del paquet (té el "no modifier").

El cos de la interface és la llista de mètodes i/o constants que conté la interface. Per a les constants no cal indicar que són static i final i per als mètodes no cal indicar que són public. Aquestes característiques s'assignen implícitament.

Per acabar, cal comentar que, com que per definició totes les dades membre que es defineixen en una interface són static i final , i atès que les interface no es poden instanciar, també resulten una bona eina per implantar grups de constants.
Exemple:

```
public interface DiesSetmana
{
int DILLUNS = 1, DIMARTS=2, DIMECRES=3, DIJOUS=4;
int DIVENDRES=5, DISSABTE=6, DIUMENGE=7;
String [] NOMS_DIES = {"", "Dilluns", "Dimarts", "Dimecres",
   "Dijous", "Divendres", "Dissabte", "Diumenge"};
}
```

Aquesta definició ens permet utilitzar les constants declarades en qualsevol classe que implementi la interface, de manera tan simple com:

```
System.out.println (DiesSetmana.NOMS_DIES[DILLUNS]);
```

2.4. Inicialització de subclasses (super)

Una **classe A** que derivi d'una **superclasse B**, per a poder accedir als constructors de la superclasse ha de fer servir **super(llista de paràmetres)**. D'aquesta manera es cridarà al constructor de la superclasse que tingui els paràmetres llistats. D'aquesta manera el constructor de **A** pot inicialitzar els atributs heretats de la superclasse i després inicialitzar els seus propis.

Exemple:

```
public abstract class Tripulant {
    protected String ID;
    protected String nom;


    // Aquestconstructor només podrà ser cridat per un objecte
    // de la classe Tripulant o d'alguna que la hereti.
    protected Tripulant(String ID, String nom){
        this.ID = ID;
        this.nom = nom;
    }
}
```

```
public class Oficial extends Tripulant {
    private boolean serveiEnElPont;
    private String descripcioFeina;


    public Oficial(String ID, String nom, boolean serveiEnElPont, String descripcioFeina){
        super(ID, nom);
        this.serveiEnElPont = serveiEnElPont;
        this.descripcioFeina = descripcioFeina;
    }
}
```

La utilització de la paraula reservada **super** com a mètode per cridar un constructor de la superclasse en el disseny d'un constructor de la subclasse només es pot efectuar en la primera sentència del constructor de la subclasse.

2.5. Sobreescriptura de mètodes (superclasse --> subclasse, + super)

Sobreescriure un mètode vol dir tornar-lo a definir en una subclasse, de manera que el mètode associat i, per tant, el codi que s'executa, tingui un comportament diferent al de la superclasse.

En el disseny cal indicar que un mètode s'ha sobreescrit. Això es fa tornant-lo a especificar en les subclasses, exactament igual, tot i ja existir en una superclasse.

Un cop un mètode ha estat sobreescrit, les subclasses successives, a mesura que es descendeix per la jerarquia, hereten la darrera versió, a menys que elles també decideixin sobreescriure-ho.

Quan sobre un objecte donat es crida un mètode que la seva classe ha sobreescrit (de la qual ell s'ha instanciat), el mètode que s'executa sempre és el definit en la classe a què pertany l'objecte.

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 un mètode polimòrfic (està definit en la superclasse i en la subclasse), el codi que s'acaba executant és el relatiu al tipus real de l'objecte, no el de la superclasse.

Exemple:

```
public abstract class Tripulant {
    protected String ID;
    protected String nom;


    // CONSTRUCTORS:
    // Aquest constructor només podrà ser cridat per un objecte
    // de la classe Tripulant o d'alguna que la hereti.
    protected Tripulant(String ID, String nom){
        this.ID = ID;
        this.nom = nom;
        this.actiu = actiu;
        this.dataAlta = dataAlta;
        this.departament = departament;
        this.llocDeServei = llocDeServei;
    }


    // METODES:
    protected void patata() {
        System.out.println("Executat Tripulant.patata()");
    }
}

public class Oficial extends Tripulant {
    private boolean serveiEnElPont;
    private String descripcioFeina;


    // CONSTRUCTORS:
    public Oficial(String ID, String nom, boolean serveiEnElPont, String descripcioFeina){
        super(ID, nom);
        this.serveiEnElPont = serveiEnElPont;
        this.descripcioFeina = descripcioFeina;
    }


    // METODES:
    protected void patata() {
        System.out.println("Executat Oficial.patata()");
    }

}


public class IKSRotarran {

    public static void main(String[] args) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");

        Oficial capitaTmp1 = new Oficial("001-A", "Martok", true, LocalDateTime.parse("15-08-1954 00:01", formatter), 1, 1, true, "");
        System.out.println("capitaTmp1 ha estat declarat del tipus Oficial:");
        capitaTmp1.patata();

        Tripulant capitaTmp2 = new Oficial("001-A", "Martok", true, LocalDateTime.parse("15-08-1954 00:01", formatter), 1, 1, true, "");
        System.out.println("capitaTmp2 ha estat declarat del tipus Tripulant:");
        capitaTmp2.patata();
    }
}
```

El resultat d'executar el main() és:

```
capitaTmp1 ha estat declarat del tipus Oficial:
Executat Oficial.patata()

capitaTmp2 ha estat declarat del tipus Tripulant:
Executat Oficial.patata()
```

Un altre exemple:

```
public class X {
  met1 () {...codi X...}
}

public class Z extends X {
  met1 () {...codi Z...} // Sobreescriptura de met1() d'X
  met2 () {...}  // Mètode inexistent a la classe X
}

X ox = new X ();
X oz = new Z ();

ox.met1();  // (1)
oz.met1();  // (2)
ox.met2();  // (3)
oz.met2();  // (4)
```

És clar que en la instrucció (1) s'executarà la versió de met1() de la classe X , ja que tant la variable de referència "ox" com l'objecte són de la classe X . En la instrucció (2) s'executarà la versió de met(1) de la classe Z , ja que preval la classe a la qual pertany l'objecte per damunt de la classe a la qual pertany la variable "oz" emprada per fer referència a l'objecte. Les instruccions (3) i (4) són errònies i el compilador no les accepta perquè, en la classe a què pertanyen les variables de referència "ox" i "oz", no existeix cap mètode anomenat met2() .
Si es vol aplicar el mètode met2() de la classe Z a l'objecte apuntat per "oz", cal aplicar una conversió cast de la variable "oz" cap a la classe Z tot escrivint:

((Z)oz).met2();

En el llenguatge Java, per tal d'aplicar sobreescriptura d'algun mètode a una subclasse cal recordar les següents regles, que cal tenir en compte:
• El nom i la llista i ordre dels arguments han de ser iguals al del mètode de la superclasse que es vol sobreescriure.
• El tipus de retorn de tots dos mètodes ha de ser igual.
• El mètode de la classe derivada no pot ser menys accessible que el de la classe pare.
• El mètode de la classe derivada no pot provocar més excepcions que el mètode del pare.

Si dins una classe cal accedir a la versió de la superclasse per un mètode sobreescrit, disposem de la paraula reservada **super** amb la sintaxi:

super.nomMètode(<paràmetres>)

2.6. Sobreescriptura del mètode finalize()

El mètode **finalize()**, definit en la classe **Object** i, per tant, existent per herència en totes les classes, és cridat de manera automàtica pel recuperador de memòria just abans de destruir un objecte i cal sobreescriure'l en les classes en què pertoqui efectuar alguna actuació abans de destruir-ne els objectes.

Si, a banda d'indicar-hi les instruccions corresponents a l'actuació que pertoqui, cal mantenir les instruccions de finalització que hi pogués haver dissenyades en la classe base, cal dissenyar el mètode de manera similar a:

```
void finalize() {
        <codi_corresponent_a_l'actuació>
        super.finalize();
}
```

2.7. Sobreescriptura del mètode equals()

El llenguatge Java proporciona l'operador de comparació "==", que, aplicat sobre **dades de tipus primitius**, compara si les dues dades contenen el mateix valor, i aplicat sobre **referències a objectes** compara si les dues referències fan referència a un mateix objecte. Aquest operador s'ha d'usar amb una mica més de cura quan s'opera amb objectes.

És clar que en moltes classes (per no dir totes) pot ser necessari disposar d'algun mecanisme per comprovar si dos objectes són iguals o no, a partir d'un criteri determinat respecte al seu contingut, i això no ho proporciona l'operador "==".

Amb aquest propòsit, Java proporciona un mètode a la classe **Object** , que s'hereta en totes les classes i ens proposa la seva utilització en les diverses classes després de la sobreescriptura. És el mètode següent:

```
public boolean equals (Object obj)
```

La implementació d'aquest mètode en la classe Object (que és la que s'hereta en cas de no sobreescriure'l) retorna el resultat de la comparació "**==**" entre la referència que apunta l'objecte sobre el qual s'aplica el mètode i la referència passada com a paràmetre. És a dir, si no se sobreescriu, resulta que **x.equals(y)** dóna el mateix resultat que "**x == y**".

Des de Eclipse fem click dret sobre el codi de la classe --> Source --> Generate hasCode() and equals().

En el cas de sobreescriure aquest mètode (el **equals()**), la documentació oficial de Java recomana sobreescriure també el mètode **hashCode()** per assegurar que dos objectes que han resultat iguals amb **equals()**, donaran el mateix resultat amb **hashCode()**.

2.8. Sobreescriptura del mètode toString()

¿Alguna vegada heu provat d'executar **System.out.println(obj)** en què **obj** fa referència a un objecte d'una classe qualsevol dissenyada per vosaltres?
El mètode **System.out.println()** està pensat per mostrar una cadena i, si com a paràmetre se li indica una referència a un objecte, la màquina virtual Java crea una representació String de l'objecte que tenir com a resultat alguna cosa semblant a:

```
Oficial capita = new Oficial("001-A", "Martok", true, LocalDateTime.parse("15-
            08-1954 00:01", formatter), 1, 1, true, "Capitanejar la nau.");

System.out.println("L'objecte capita: " + capita);
```

i el **println()** donaria alguna cosa semblant a:

```
L'objecte capita: Exercici_reforç_de_1r.Oficial@2baf7c4
```

La màquina virtual Java utilitza el mètode **toString()** en qualsevol lloc on necessiti tenir una representació en cadena d'un objecte i, és clar, la conversió proporcionada pel mètode heretat de la classe Object no acostuma a ser útil. Ens convé, doncs, sobreescriure'l.

Des de Eclipse fem click dret sobre el codi de la classe --> Source --> Generate toString().