TEMA 7

HERÈNCIA POLIMORFISME I

INTERFÍCIES

Índex de continguts

1 INTRODUCCIÓ

Un dels mecanismes més potents que incorpora el paradigma de programació orientada a objectes són l'herència i el polimorfisme.

El primer ens permet crear una jerarquia de classes relacionades entre sí de tal forma que la quantitat de codi, especialment el codi redundant, es redueix significativament.

El segon permet redefinir mètodes de tal forma que realitzen funcions diferents depenent del context en el qual es troben. En altres paraules, el polimorfisme és el mecanisme de la programació orientada a objectes que permet que una funció amb el mateix nom realitze tasques diferents depenent de l'objecte des del qual es fa referència.

Finalment també veurem com a les interfícies podrem especificar les operacions que haurien de definir-se a les classes que la implementen. Una interfície és un mitjà per a que els objectes no relacionats es comuniquen entres sí. Aquestes són les definicions de mètodes i valors sobre els quals els objectes estan d'acord per cooperar.

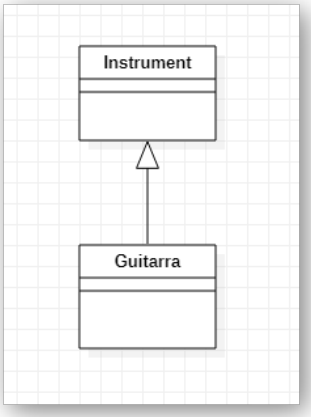
2 HERÈNCIA

En programació orientada a objectes l'herència és un mecanisme que permet potenciar la reutilització i l'extensibilitat en el desenvolupament de programari a banda de reduir la quantitat de codi redundant. Aquest mecanisme permet crear noves classes a partir d'una classe o jerarquia de classes preexistent (ja comprovades i verificades) evitant d'aquesta forma el redisseny, la modificació i verificació de la part ja implementada. L'herència facilita la creació d'objectes a partir d'altres ja existents i implica que una subclasse obté tot el comportament (mètodes) i finalment els atributs (variables) de la seua superclasse.

En aquest tema veurem com Java implementa el mecanisme de l'herència i quines són les opcions que ens dona.

Suposem el cas en el que disposem de dues classes (amb menys classes resultaria impossible l'herència), una s'anomenarà: classe base, superclasse o classe pare; mentre que l'altra serà la classe derivada, subclasse o classe filla.

En el nostre cas la classe pare, superclasse o classe base serà «Instrument» i la classe derivada, filla o subclasse serà «Guitarra» que en llenguatge natural es podria traduir com que una guitarra és un tipus d'instrument.



La classe base en una jerarquia d'herència és la classe que conté els atributs i mètodes comuns a totes les classes derivades i és per això que s'evita la redundància del codi que es produiria si haguérem de repetir codi comú a cadascuna de les subclasses, per tant, es podria deduir que una classe derivada és de facto una ampliació de les funcionalitats de la classe base ja que contindrà tant els membres de la classe base com els que s'afegeixen de més a la classe derivada.

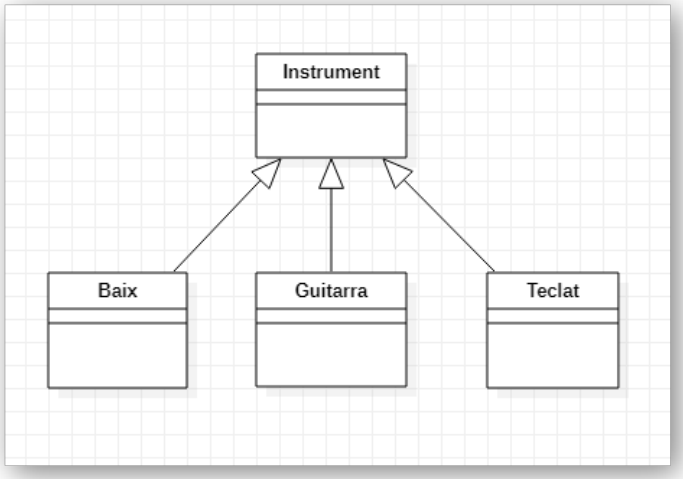
2.1 Declaració d'una classe derivada

Per tal de declarar que una classe hereta d'una altra, Java utilitza la paraula reservada 'extends'. Mira l'exemple següent:

```
public class Guitarra extends Instrument {

}
```

Afegim al nostre exemple dos instruments més; un baix i un teclat. El diagrama de classes quedaria així:



El codi en Java resultant seria el següent:

```
public class Instrument {

}
public class Guitarra extends Instrument {

}

public class Baix extends Instrument {

}
```

Respecte de la visibilitat dels membre de la superclasse, les subclasses tindran accés als membres, siguen atributs o mètodes, public o protected mentre que no heretaran els membres privats de la classe pare.

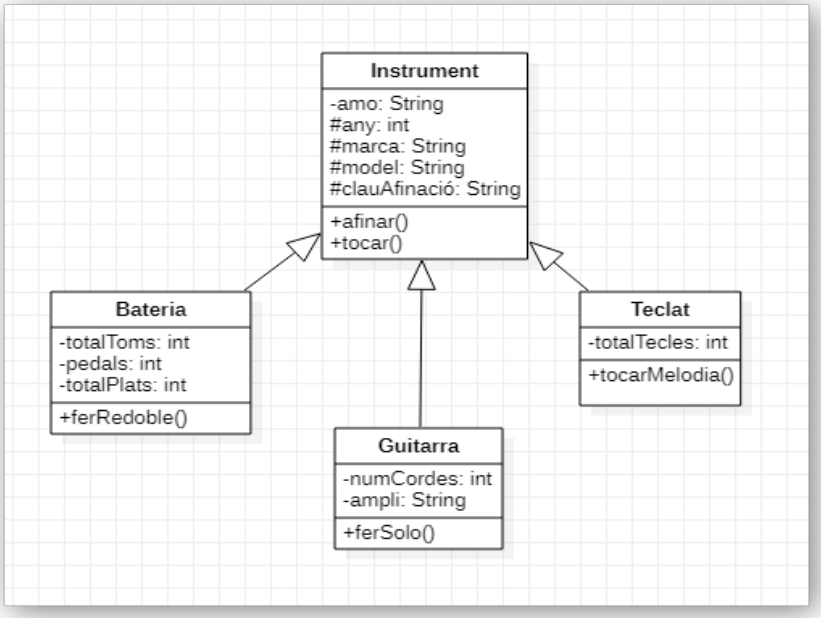
2.2 Disseny de subclasses

Dissenyar és adaptar la realitat que es vol representar a un model concret, en el nostre cas els diagrames de classes d'UML. És en aquest pas en el que s'han de prendre les decisions que poden condicionar el resultat final de la nostra aplicació i no sempre és fàcil determinar quines són les relacions que s'han d'establir. És per això que cal tenir en ment sempre la següent màxima: «La jerarquia de classes més eficient i efectiva és aquella que minimitza el codi al redundant al màxim i simplifica el codi»

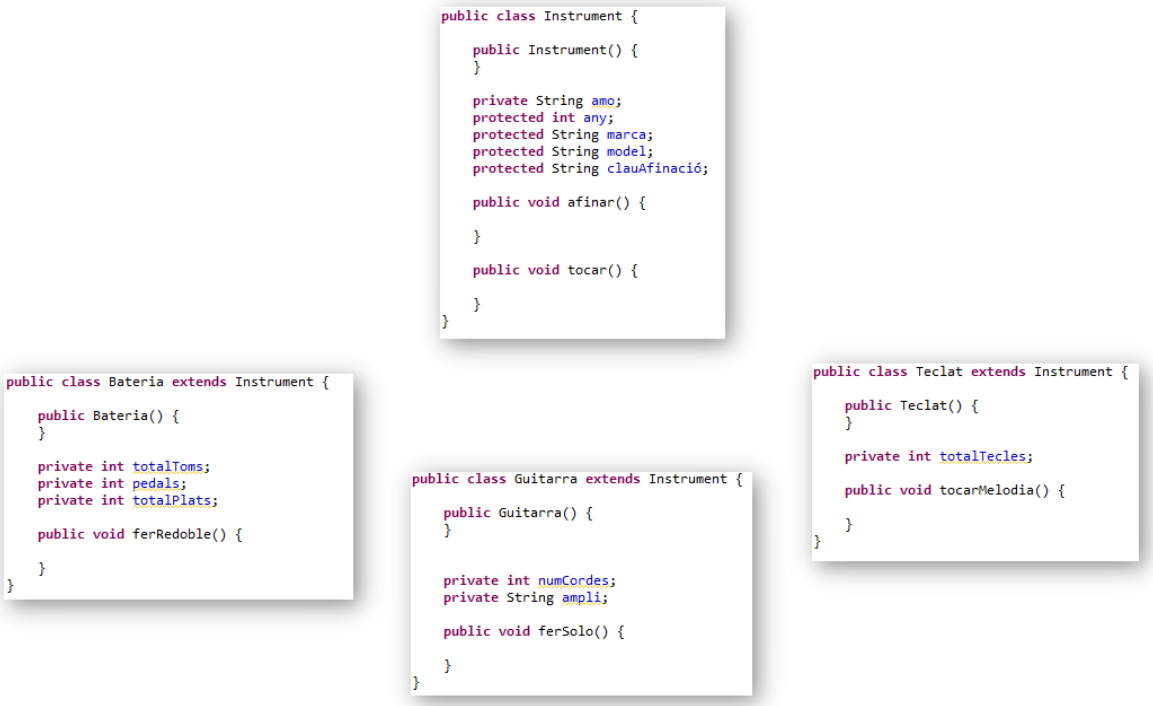
Al utilitzar el mecanisme de l'herència en programació orientada a objectes hem d'intentar sempre tindre un màxim de membres a la superclasse i després a les diferents subclasses només aquells membres o atributs que fan de discriminador, és a dir, que diferencien la classe especialitzada (o subclasse) de la classe superior.

Ampliem el nostre exemple de banda de rock anterior i afegim atributs i membres a totes les classes. Per exemple cada instrument tindrà amo, any, tipus, marca, model i clau d'afinació. A més a més, podrà afinar i tocar.

Ja més específicament d'una bateria hem de saber el total de Toms que té, el total de pedals (hi ha bateries amb doble pedal) i el total de plats a banda de poder tocar un redoble. D'una guitarra hem d'especificar quin ampli utilitza i quantes cordes té (pot tenir-ne fins 12). Finalment pel que fa al teclat, sabrem el nombre de tecles que té i podrà tocar melodies.



El codi resultant d'aquest disseny seria el següent:



```java
public class Instrument {

    public Instrument() {
    }

    private String amo;
    protected int any;
    protected String marca;
    protected String model;
    protected String clauAfinació;

    public void afinar() {

    }

    public void tocar() {

    }
}
```

```java
public class Bateria extends Instrument {

    public Bateria() {
    }

    private int totalToms;
    private int pedals;
    private int totalPlats;

    public void ferRedoble() {

    }
}
```

```java
public class Guitarra extends Instrument {

    public Guitarra() {
    }


    private int numCordes;
    private String ampli;

    public void ferSolo() {

    }
}
```

```java
public class Teclat extends Instrument {

    public Teclat() {
    }

    private int totalTecles;

    public void tocarMelodia() {

    }
}
```

Del codi anterior es pot extreure la següent informació:

- Amo és un atribut que no hereta cap subclasse al ser privat.
- Totes les subclasses tindran accés, encara que al codi no estiga, a any, marca, model i clauAfinació.
- Totes les subclasses quan criden als mètodes afinar i tocar, executaran el codi que hi ha a la superclasse sempre que no es s'indique el contrari (sobrecàrrega)
- Guitarra i Bateria no tenen accés a l'atribut totalTecles.
- Teclat i Bateria no tenen poden fer solos.

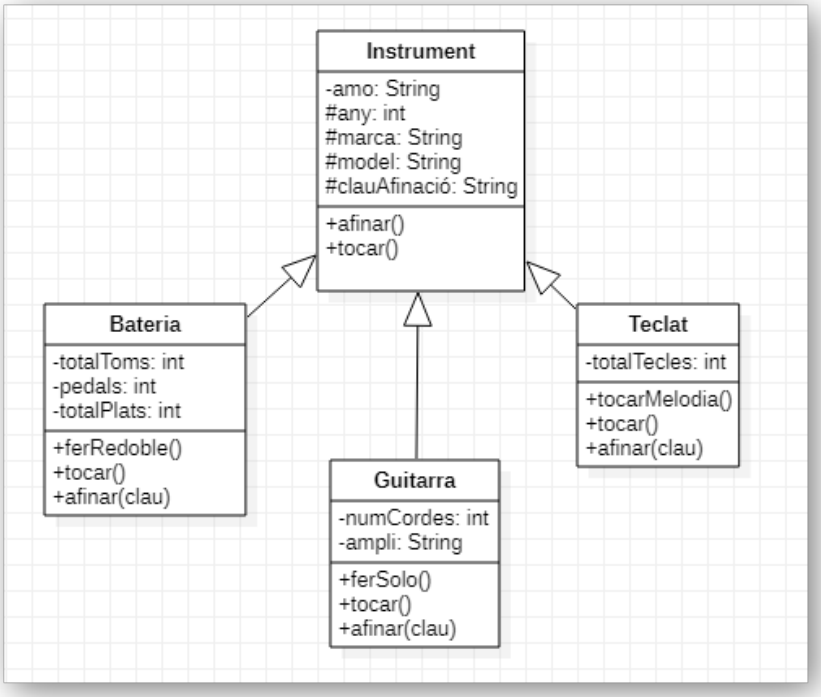
2.3 Sobrecàrrega de mètodes a la classe derivada

La sobrecàrrega de mètodes és la creació de diversos mètodes amb el mateix nom però amb diferent llista de tipus de paràmetres. Java diferencia entre els mètodes sobrecarregats en base al número i tipus de paràmetres o arguments que té el mètode i no pel tipus que retorna.

Una classe derivada pot redefinir un mètode de la classe base amb el mateix nom però una llista diferent d'arguments

Si pel contrari tenim un mateix mètode amb el mateix nom i la mateixa signatura (els mateixos paràmetres) el que estem fent es sobreescriure (overridint) i no sobrecarregar (overloading)

Seguint amb l'exemple de la nostra banda de rock mireu el següent codi:



Com es pot observar a la imatge anterior hi ha una sobrecarrega del mètode afinar a cada classe derivada perquè seria el mateix que el de la classe pare però amb diferents paràmetres mentre que el mètode tocar està sobreescrit ja que la interfície és idèntica. En codi podria quedar de la següent forma:

```java
public class Instrument {

    public Instrument() {
    }

    private String amo;
    protected int any;
    protected String marca;
    protected String model;
    protected String clauAfinació;

    public void afinar() {
        System.out.println("Afinar Super Classe");

    }

    public void tocar() {
        System.out.println("Tocar instrument");
    }
}
```

```java
public class Guitarra extends Instrument {

    public Guitarra() {
    }


    private int numCordes;
    private String ampli;

    public void ferSolo() {

    }

    public void tocar () {
        System.out.println("Tocar la guitarra");
    }

    public void afinar (char clau) {
        System.out.println("Afinar la guitarra en clau: " + clau );
    }
}
```

```java
public class Teclat extends Instrument {

    public Teclat() {
    }

    private int totalTecles;

    public void tocarMelodia() {

    }

    public void tocar () {
        System.out.println("Tocar el teclat");
    }

    public void afinar (char clau) {
        System.out.println("Afinar el teclat en clau: " + clau );
    }
}
```

```java
public class Bateria extends Instrument {

    public Bateria() {
    }

    private int totalToms;
    private int pedals;
    private int totalPlats;

    public void ferRedoble() {

    }

    public void tocar () {
        System.out.println("Tocar la bateria");
    }

    public void afinar (char clau) {
        System.out.println("Afinar la bateria en clau: " + clau );
    }
}
```

2.4 Herència pública

En Java tenim la possibilitat de fer protected (#) private (-) public (+) i package (~) ens determina la visibilitat dels membres d'una classe. Java considera que l'herència és sempre pública i que la classe derivada tindrà per tant accés als membres protected i public de la classe pare però no als membres privats. El fet de poder tindre elements privats a la nostra classe base seria per poder ocultar informació a les classes derivades.

2.5 Constructors en herència

De la mateixa manera que s'hereten mètodes i atributs també s'hereten els constructors de la classe. D'aquesta forma si declarem un objecte d'una classe derivada, primer s'executa el constructor de la classe base i a continuació s'executa la part de codi de la classe derivada.

Continuant amb la nostra banda de rock farem algunes modificacions. Afegim un constructor parametritzat a la classe instrument que s'encarregarà d'inicialitzar els atributs amo, any, marca i model.

```
public class Instrument {

    public Instrument() {

    }

    public Instrument (String strAmo, int iAny, String strMarca, String strModel ) {
        this.amo = strAmo;
        this.any = iAny;
        this.marca = strMarca;
        this.model = strModel;
    }

    private String amo;
    protected int any;
    protected String marca;
    protected String model;
    protected String clauAfinació;

    public void afinar() {
        System.out.println("Afinar Super Classe");

    }

    public void tocar() {
        System.out.println("Tocar instrument");
    }
}
```

Per l'altre costat a la classe Guitarra hi afegim també un nou constructor (sobrecarregat) amb els paràmetres: amo, any, marca, model, número de cordes i l'amplificador que utilitza.

```
public class Guitarra extends Instrument {

    public Guitarra() {
    }

    public Guitarra (String strAmo, int iAny, String strMarca, String strModel, int iCordes, String strAmpli ) {
        super(strAmo, iAny, strMarca, strModel);
        this.numCordes = iCordes;
        this.ampli = strAmpli;
    }

    private int numCordes;
    private String ampli;

    public voidferSolo() {

    }

    public void tocar () {
        System.out.println("Tocar la guitarra");
    }

    public void afinar (char clau) {
        System.out.println("Afinar la guitarra en clau: " + clau );
    }
}
```

Amb aquest disseny de classes podríem per exemple executar el següent codi

```
public class Test {

    public static void main (String [] args) {
        Guitarra gibsonLesPaul = new Guitarra ("Vicent", 2007, "Gibson", "LesPaul", 6, "Marshall" );
        gibsonLesPaul.afinar('R');
        gibsonLesPaul.tocar();
    }
}
```

Qué és el que estaria passant si ferem un traça?

Al crear un objecte 'gibsonLesPaul' de la classe guitarra passant-li paràmetres, es cridaria al constructor de la classe guitarra. Aquest el primer que faria seria cridar al constructor de la classe pare utlitzant 'super' per indicar-li que ha d'executar el constructor de la superclasse assignant-li d'aquesta forma: l'amo, any, marca i model per immediatament després assignar el nombre de cordes i l'amplificador de la classe derivada.

El mètode super(), es crida només quan s'utilitzen constructors sobrecarregats per tal que execute també el constructor de la superclasse amb paràmetres i no el constructor per defecte que és el que faria si no li diguem el contrari.

Exemple

Que executaria el següent codi?

```
class B1
{
  public  B1() { System.out.println("Constructor-B1"); }
}
class B2
{
  public  B2() { System.out.println("Constructor-B2");}
}
class D extends B1
{
  public  D() { System.out.println("Constructor-D"); }
}
class H extends B2
{
  public  H() { System.out.println("Constructor-H"); }
}
class Dh extends H
{
  public  Dh() { System.out.println("Constructor-Dh"); }
}
class Constructor
{
  public static void main(String [] ar)
  {
    D d1 = new D();
    System.out.println("_________________\n");
    Dh d2 = new Dh();
  }
}
```

A més a més, super() també ens permet cridar a mètodes de la classe pare de la següent forma:

```
super.metode(argumnets);
```

2.6 Conversió entre subclasse i superclasse

De vegades en programació hem de poder convertir entre tipus de dades, aquest mecanisme s'anomena també «cast». Per exemple si tenim un double amb valor 6,8 i volem convertir-lo en un moment determinat de la nostra aplicació a un int, ho fariem de la següent forma:

```
double d = 6,8;
int i = (int)d;
```

Entre objectes també existeix la possibilitat de aplicar aquest mecanisme. Al nostre model de banda de rock, ens podríem trobar amb la següent situació:

```
Instrument ins = new Instrument();
Guitarra git = new Guitarra();
Teclat tec = new Teclat();
ins = git; // Conversió automàtica.
ins = tec; // Conversió automàtica.
```

2.7 Classes no derivables: atribut final

Si volem que alguna classe en concret no siga ja més derivable, és a dir, que no volem que es creen més subclasses a partir d'una classe concreta, ho indiquem utilitzant la paraula reservada «final»

```
public final class Bateria extends Instrument {

    public Bateria() {
    }

    private int totalToms;
    private int pedals;
    private int totalPlats;

    public void ferRedoble() {

    }

    public void tocar () {
        System.out.println("Tocar la bateria");
    }

    public void afinar (char clau) {
        System.out.println("Afinar la bateria en clau: " + clau );
    }
}
```

Amb aquest codi aconseguim que no es puguen crear subclasses de la classe Bateria.

3 POLIMORFISME

El polimorfisme (del Grec πολύς, polys, "molt, molts" i μορφή, morphē, "forma, figura") és una característica d'alguns llenguatges de programació que tenen la propietat d'enviar missatges sintàcticament iguals als objectes de diferents tipus. L'únic requisit que han de complir els objectes que s'utilitzen de manera polimòrfica és saber respondre al missatge que se'ls hi envia.

El polimorfisme permet fer referència a altres objectes de classes mitjançant el mateix element de programa i realitzar la mateixa operació de diferents maneres d'acord amb l'objectes al qual es fa referència en cada moment.

Si mirem la nostra banda de rock, l'exemple més clar és el del mètode tocar, tots el nostres instruments poden tocar però cadascun ho fa d'una forma diferent.

3.1 Lligadura

El terme lligadura es correspon a l'enllaç que es produeix entre un objecte o instància d'una classe i les seues propietats. Si aquestes propietats són atributs, la lligadura és refereix a la connexió entre l'atribut i la referència a memòria on es trobaria el valor de l'atribut en qüestió mentre que si estem parlant de mètodes, la lligadura és refereix a la connexió entre el nom del mètode que es pretén executar i el codi que executaria.

Per un altre costat, el temps de lligadura és el moment en el que l'atribut s'associa amb el seu valor o que un mètode s'agrupa amb el seu codi corresponent. Aquest temps de lligadura pot ser 'estàtic' o 'dinàmic'. El primer es produeix durant la compilació del programa mentre que la segona es produeix en temps d'execució.

La majoria de llenguatges de programació com C utilitzen la lligadura estàtica en temps de compilació mitjançant l'enllaçador, pel contrari els llenguatges de programació que utilitzen lligadura dinàmica, no determinen quin és el codi que s'executarà en la crida al mètode fins que arriba el moment en temps d'execució i és només en eixe cas en el que es determinarà de totes les possibilitats de codi a executar (polimorfisme) quin és el que s'enllaçarà o lligarà (lligadura efectiva).

Java és un llenguatge orientat a objectes que utilitza la lligadura dinàmica en temps d'execució amb l'excepció dels mètodes static o final que utilitzen lligadura estàtica.

3.2 Classes i mètodes abstractes

Les classes i mètodes abstractes són un «fenomen» que es dona en algunes de les generalitzacions que ens apareixen quan analitzem una realitat des del punt de vista de la programació orientada a objectes. És a dir, l'abstracció com a tècnica en POO només es pot donar en l'herència entre classes, concretament a les classes més altes de la jerarquia.

Una classe abstracta és aquella de la qual mai s'instanciarà cap objecte. Per exemple Instrument a la nostra banda de rock on o s'instancia una guitarra, un baix o una bateria però mai un instrument com a tal.

A nivell de codi l'únic canvi que es produeix és que a la classe general «Instrument» se li afegeix la paraula reservada abstract com a modificador de comportament de la classe.

```
public abstract class Instrument {
```

Normalment els mètodes d'una classe abstracta també seran abstractes.

Al convertir la classe Instrument a abstracta ja no la podríem instanciar és a dir, no podrem crear cap objecte de la classe Instrument, és a dir, no podrem en cap moment fer Instrument ins = new Instrument(); ja que ens donaria error de compilació però en canvi, si podríem fer Instrument ins = new Guitarra(); ja que la classe Guitarra no és abstracta i una Guitarra és, segons el nostre model, un instrument.

Mètodes abstractes

Un mètode abstracte és un mètode que defineix un comportament o funcionalitat concreta però no especifica la seua implementació i ha de ser la classe filla la que implemente el mètode.

En altres paraules, a la classe para o superclasse podem tindre mètode tocar, public void abstract tocar(), que no tindra cap codi a la seua implementació si no que serà la classe filla o subclasse, guitarra per exemple, la que incorpore el codi dins del seu mètode public void tocar(). Fixeu-se que aquest últim ja no porta la paraula reservada «abstract»

La implementació de mètodes abstractes per part de les classes filles en una jerarquia utilitzen la lligadura dinàmica ja que no es possible determinar el codi que s'ha d'executar en temps de compilació.

3.3 Ús del polimorfisme

Arribats a aquest punt podem treure les següents conclusions: que el polimorfisme ens permet que diferents objectes responguen de manera diferent al mateix missatge (mètode amb el mateix nom) i que aquesta diferenciació de quin codi executar ho fa el compilador amb lligadura dinàmica (en temps d'execució i no de compilació)

Per usar el polimorfisme a Java hem de seguir les següents regles:

1. Crear una jerarquia de classes amb les operacions importants definies per els mètodes membre declarades com abstractes a la base.

2. Les implementacions específiques dels mètodes abstractes s'han de fer a les classes derivades.

3. Les instàncies d'aquestes classes s'usen mitjançant una referència a la base amb lligadura dinàmica la qual és l'essència del polimorfisme a Java

Finalment,en realitat no és necessari declarar abstractes els mètodes a la classe base si després es redefineixen amb la mateixa signatura (nom més paràmetres) a la classe derivada.

3.4 Avantatges del polimorfisme

El polimorfisme fa el seu sistema més flexible, sense perdre cap dels avantatges de la compilació estàtica de tipus que tenen lloc en temps de compilació; tal és el cas de Java.

Les aplicacions més freqüents del polimorfisme són:

- Especialització de classes derivades. És a dir, especialitzar classes que han estat definits des; per exemple: Quadrat és una especialització de la classe Rectangle perquè qualsevol quadrat és un tipus de rectangle; aquesta classe de polimorfisme augmenta la eficiència de la subclasse, mentre conserva un alt grau de flexibilitat i permet un mitjà uniforme de manejar rectangles i quadrats.

- Estructures de dades heterogenis. De vegades és útil poder manipular conjunts similars dors d'objectes; amb el polimorfisme es poden crear i gestionar fàcilment estructures de dades heterogenis que són fàcils de dissenyar i dibuixar, sense perdre la comprovació de tipus dels elements utilitzats.

- Gestió d'una jerarquia de classes. Són col·leccions de classes altament estructurades

amb relacions d'herència que es poden estendre fàcilment.

3.5 Mètodes no derivables: atribut final

Al context de l'herència, la paraula reservada final, s'utilitza per protegir la redefinició dels mètodes de la classe base; un mètode que té l'atribut final, no pot tornar a definir-se a la o les classes derivadesredefinció

4 INTERFÍCIES

Una interfície en Java, és sintàcticament similar a una classe abstracta, en la qual pot especificar un o més mètodes que no tenen cos. Aquests mètodes han de ser implementats per una classe perquè es defineixen les seues accions.

Per tant, una interfície especifica què s'ha de fer, però no com fer-ho. Una vegada que es defineix una interfície, qualsevol quantitat de classes pot implementar-la. A més, una classe pot implementar qualsevol quantitat d'interfícies.

Per implementar una interfície, una classe ha de proporcionar cossos (implementacions) per als mètodes descrits per la interfície. Cada classe és lliure de determinar els detalls de la seva pròpia implementació. Dues classes poden implementar la mateixa interfície de diferents maneres, però cada classe encara admet el mateix conjunt de mètodes. Per tant, el codi que téTema 7. Herència, polimorfisme i Interfícies coneixement de la interfície pot usar objectes de qualsevol classe, ja que la interfície amb aquests objectes és la mateixa.

4.1 Implementació d'una interfície

La interfície especifica el comportament comú que tenen un conjunt de dades el qual es realitza en cadascuna d'elles i es coneix també com a implementació d'una interfície. La sintaxis és similar a la derivació o extensió d'una classe amb la paraula reservada implements en lloc d'extends.

```
Class nomClasse implements nomInterficie {
// definició d'atributs
// implementació dels mètodes de la classe
// implementació dels mètodes de la interfícies
}
```

La classe que implementa la interfície ha d'especificar el codi (la implementació) de cadascun dels mètodes, en cas de no fer-ho, la classe esTema 7. Herència, polimorfisme i Interfícies converteix en abstracta i com a tal s'ha de declarar. És a dir, si una classe implementa una interfície, està obligada a implementar tots els seus mètodes.

Per tal d'il·lustrar amb un exemple sobre una interfície, utilitzarem un comandament a distància de qualsevol dispositiu electrònic. Es suposa que des

d'un comandament a distància podrem: encendre i apagar el dispositiu, pujar i baixar el volum, silenciar etc..

La nostra interfície creada amb Java podria ser com la següent:

```
public interface Comandament {
	void engegar();
	void apagar();
	void pujaVolum(int increment);
	void baixaVolum(int decrement);
	void silenciar();
}
```

Els mètodes declarats ací a la interfície Comandament deurien ser autoexplicatoris. Hem inclòs només unes poques de les múltiples accions que es poden realitzar amb un comandament a distància d'un dispositiu electrònic i encara se li podrien afegir algunes més ben segur.

No hi ha definició de cap dels mètodes ací, Els mètodes declarats en una interfície són sempre abstractes i sempre públics per defecte. Ara qualsevol classe que necessite de l'ús de la funcionalitat proveïda per Comandament només ha de delcarar que implementa la interficie a més a més de definir cadascun dels mètodes que la formen.

Posem per exemple una televisió, el codi Java seria el següent:

```java
public class TV implements Comandament {

    public void engegar() {
        // Codi necessari per engegar la Tele
    }

    public void apagar() {
        // Codi necessari per apagar la Tele
    }

    public void pujaVolum(int increment) {
        // Codi necessari per pujar volumn a la Tele
    }

    public void baixaVolum(int decrement) {
        // Codi necessari per baixar volumn a la Tele
    }

    public void silenciar() {
        // Codi necessari per silenciar la Tele

    }

}
```

4.2 Jerarquia d'interfícies

Les interfícies també es poden organitzar de forma jerarquica de manera que els mètodes siguen heredats

```
interficie extends interficie2 extends interficie3...
```

4.3 Herència de classes i implementació d'interfícies

Les interfícies no són classes ja que especifiquen un comportament mitjançant mètodes per la classe que els implemente; per això, una classe pot heretar de la seua classe base i al mateix temps implementar una interfície. Per exemple si tenim una classe base anomenada Electrodomèstic de la qual hereten una sèrie de classes com podrien ser: Televisió, DVD, Microones, Forn, Nevera etc..i a més a més també disposem d'una interfície Comandament, podríem crear una aplicació en la que la televisió herete de la classe general electrodomèstic i a banda implemente la interfície comandament com hem vist abans.

```
public class Televisio extends Electrodomestic implements
comandament { ... }
```

4.4 Classes internes

Una classe interna és una classe que es declara dintre d'una altra classe.