Classes abstractes i interfaces en Java

Classes abstractes

Un dels principis del disseny en programació orientada a objectes és que s'ha de programar sobre abstraccions en comptes de sobre concrecions. Això permet fer les nostres aplicacions menys sensibles als canvis de requeriments o ampliacions de funcionalitat que pugui haver-hi.

Una classe abstracta es defineix amb el modificador **abstract**. Es caracteritza perquè no se'n poden instanciar objectes. És una classe que s'utilitza per derivar-ne una família de classes concretes, de les quals derivarem els objectes de la nostra aplicació. La classe abstracta pot contenir atributs i mètodes, que seran comuns a totes les classes derivades, les quals poden redefinir els mètodes segons la seva conveniència. Alguns mètodes seran abstractes. Això implica que no en definim la funcionalitat (són mètodes buits, sense cos), la qual es defineix realment a les classes derivades, de manera diferenciada per a cada classe. L'objectiu és proporcionar una interfase comuna per a la família de classes derivades.

Com a exemple, anem a crear una classe abstracta denominada Animal de la qual derivem les classes Gat i Gos. Les dues classes redefineixen la funció parla() declarada abstracta en la classe base Animal.

```
public abstract class Animal
{
    public abstract void parla();
}
class Gos extends Animal
{
    public void parla()
    {
        System.out.println("Guau!");
    }
}
class Gat extends Animal
{
    public void parla()
    {
        System.out.println("Meu!");
    }
}
class Ocell extends Animal
{
    public void parla()
    {
        System.out.println("Piu!");
    }
}
```

Dissenyem ara un programa ParlarAp que faci parlar els animals a través de la funció queParli().El polimorfisme ens permet passar la referència a un objecte de la classe Gat a la funció queParli() que coneix l'objecte per la seva classe base Animal.

```
public class ParlarAp
{
    public static void main(String[] args)
    {
        Gat gat=new Gat();
        queParli(gat);
        Gos gos = new Gos();
```

```
        queParli(gos);
        Ocell ocell = new Ocell();
        queParli(ocell);
    }
    static void queParli(Animal subjecte)
    {
        subjecte.parla();
    }
}
```

El compilador no sap exactament el tipus de l'objecte que se li passarà a la funció queParli() en el moment de l'execució del programa. Si es passa un objecte de la classe Gat s'imprimirà 'Meu', si es passa un objecte de la classe Gos s'imprimirà 'Guau'. El compilador només sap que se li passarà un objecte d'alguna classe derivada d'Animal. Per tant, el compilador no sap quina funció parla() serà invocada en el moment de l'execució del programa.

El polimorfisme ens ajuda a fer el programa més flexible, perquè en el futur podem afegir noves classes derivades d'Animal, sense que canviï per a res el mètode queParli(). Per exemple, podem afegir una classe Ocell, també derivada d'Animal, que parli fent ¡pio!.

Interfícies

L'ús de les classes abstractes permet donar flexibilitat als nostres dissenys i fer-los menys sensibles als canvis, però acaba comportant l'ús d'herència múltiple. Per evitar-ho, Java proporciona les **interface**.

Una interface és equivalent a una classe abstracta que **només conté atributs final i mètodes abstractes**. Aquestes limitacions permeten que una classe pugui implementar més d'una interface.

La derivació de classes a partir d'interfaces es fa amb la paraula **implements**.

Anem a crear una interface denominat Parlador que contingui la declaració d'una funció denominada parla(). Després fem que les classes derivades de la classe Animal implementin la interface Parlador.

```
public interface Parlador
{
    public abstract void parla();
}
public abstract class Animal implements Parlador
{
    public abstract void parla();
}
class Gos extends Animal
{
    public void parla()
    {
        System.out.println("¡Guau!");
    }
}
class Gat extends Animal
{
    public void parla()
    {
        System.out.println("¡Miau!");
    }
}
```

Suposem que ara volem representar un objecte d'un tipus totalment diferent que també pot emetre un soroll. Amb aquesta línia de disseny, podem fer que implementi també la interface Parlador i aprofitar el codi.

Definim una classe base Rellotge. Derivem a continuació la classe Cuco implementant la interface Parlador. Això implica que també ha de definir la funcionalitat de la funció parla().

La novetat és que ara podem passar-li a la funció queParli() una referència a qualsevol classe que implementi la interface Parlador, com si es tractés d'una herència ordinària.

```
public abstract class Rellotge
{
}
class Cuco extends Rellotge implements Parlador
{
    public void parla()
    {
        System.out.println("¡Cucu!");
    }
}
public class Parlar2Ap
{
    public static void main(String[] args)
    {
        Gat gat=new Gat();
        queParli(gat);
        Cuco cuco=new Cuco();
        queParli(cuco);
    }
    static void queParli(Parlador subjecte)
    {
        subjecte.parla();
    }
}
```

En executar el programa, veurem que s'imprimeix en la consola ¡Miau!, perquè a la funció queParli() se li passa un objecte de la classe Gat, i després ¡Cucu! perquè a la funció queParli se li passa un objecte de la classe Cuco.

Si només hagués herència simple, Cuco hauria de derivar de la classe Animal (el que no és lògic) o be no es podria passar a la funció queParli(). Amb interfícies, qualsevol classe en qualsevol família pot implementar la interface Parlador, i es podrà passar un objecte d'aquesta classe a la funció queParli(). Les interfícies, doncs, proporcionen més polimorfisme que el que es pot obtenir del mecanisme de l'herència simple de classes.

Com a regla de disseny general, és sempre preferible programar amb abstraccions (classes abstractes i interfaces) en comptes de amb concrecions (implementacions). Tot i que no es poden instanciar objectes de classes abstractes ni d'interfaces, es poden declarar identificadors d'aquests tipus, i inicialitzar-los en temps d'execució a objectes de tipus concret.

A l'exemple següent, programem amb la classe abstracta Animal.

```
public class Parladors1
{
    public static void main(String[] args)
    {
        Animal [] animal = new Animal[3];
        animal[0] = new Gos();
        animal[1] = new Gat();
        animal[2] = new Ocell();
        for (int i=0; i<animal.length; i++)
            queParli(animal[i]);
```

```
    }
    static void queParli(Parlador subjecte)
    {
        subjecte.parla();
    }
}
```

També podem programar directament amb les interfaces per obtenir més flexibilitat i explotar el polimorfisme amb enllaç tardà, tal i com fem a l'exemple següent.

```
public class Parladors2
{
    public static void main(String[] args)
    {
        Parlador [] parlador= new Parlador[4];
        parlador[0] = new Gos();
        parlador[1] = new Gat();
        parlador[2] = new Ocell();
        parlador[3] = new Cuco();
        for (int i=0; i<parlador.length; i++)
            queParli(parlador[i]);
    }
    static void queParli(Parlador subjecte)
    {
        subjecte.parla();
    }
}
```

El mètode queParli() accepta qualsevol classe que implementi la interface Parlador.

Classes i interfaces de valors constants

Sovint és necessari definir constants d'àmbit global de la nostra aplicació. Aquests valors constants poden fer referència a noms (com el de l'empresa), factors de conversió, etc. Per posar a l'abast de les nostres classes aquestes constants, definim una o vàries classes que continguin aquestes constants. Podem utilitzar dos mecanismes bàsicament per a fer-ho: definir **classes final** (sense possibilitat d'herència) o bé definir **interfaces**. En ambdós casos, hem de definir les constants amb el modificador **final**. A més, sobretot en el primer cas, utilitzarem també el modificador **static**, per no haver d'instanciar cap objecte de la classe contenidora de les constants.

Vegem en primer lloc un exemple de definició amb una classe final.

```
/**
 * FinalConstants.java
 * Exemple de classe final amb constants static
 * @author Jose Moreno
 * @version
 */
public final class FinalConstants
{
    static final double CONV_EURO_PTA = 166.386; // pta/euro
    static final double VELOC_LLUM = 300000; // km/s
    static final String RAO_SOCIAL = "Contractes, S.A.";
}
```

El següent quadre mostra una aplicació senzilla per utilitzar la classe anterior.

```
/**
 * ConstantsAp1.java
 * Us de classe final amb constants static
 * @author Jose Moreno
 * @version
 */
public class ConstantsAp1
{
    public static void main (String args[])
    {
        System.out.println("Nom de l'empresa: "+FinalConstants.RAO_SOCIAL);
        System.out.println("1 euro = "+FinalConstants.CONV_EURO_PTA+" pessetes");
        System.out.println("La velocitat de la llum en el buit es " +
    FinalConstants.VELOC_LLUM+" km/s");
    }
}
```

Alternativament, podem utilitzar una interface per encapsular les constants.

```
/**
 * InterfConstants.java
 * Exemple d'interface amb constants static
 * @author Jose Moreno
 * @version
 */
public interface InterfConstants {
    static final double CONV_EURO_PTA = 166.386; // pta/euro
    static final double VELOC_LLUM = 300000; // km/s
    static final String RAO_SOCIAL = "Contractes, S.A.";
}
```

Ara, la classe on s'utilitzen les constants només ha d'implementar la interface per tenir accessibles les constants sense haver de fer referència a la interface contenidora.

```
/**
 * ConstantsAp2.java
 * Us d'interface amb constants static
 * @author Jose Moreno
 * @version
 */
public class ConstantsAp2 implements InterfConstants
{
    public static void main (String args[])
    {
        System.out.println("Nom de l'empresa: "+RAO_SOCIAL);
        System.out.println("1 euro = "+CONV_EURO_PTA+" pessetes");
        System.out.println("La velocitat de la llum en el buit es "+VELOC_LLUM+" km/s");
    }
}
```