CENTRE ESPECÍFIC
D´EDUCACIÓ A DISTÀNCIA DE
LA COMUNITAT VALENCIANA

UNITAT 9
PROGRAMACIÓ ORIENTADA A OBJECTES II

PROGRAMACIÓ
CFGS DAW

Autors:
Carlos Cacho y Raquel Torres
Revisat per:
Lionel Tarazon - lionel.tarazon@ceedcv.es
Fco. Javier Valero – franciscojavier.valero@ceedcv.es
José Manuel Martí - josemanuel.marti@ceedcv.es
Salvador Rue Orquín – s.rueorquin@edu.gva.es

2023/2024

Llicència

CC BY-NC-SA 3.0 ES **Reconeixement – No Comercial – Compartir Igual (by-nc-sa)** No es permet un ús comercial de l'obra original ni de les possibles obres derivades, la distribució de les quals s'ha de fer amb una llicència igual a la que regula l'obra original. Aquesta és una obra derivada de l'obra original de Carlos Cacho i Raquel Torres.

Nomenclatura

Al llarg d'aquest tema s'utilitzaran diferents símbols per a distingir elements importants dins del contingut. Aquests símbols són:

Important

Atenció

Interessant

ÍNDEX DE CONTINGUT

1. RELACIONS ENTRE CLASSES

Com hem vist fins ara, es poden ocultar i encapsular diferents atributs i funcions del nostre programa dins les classes, estos seran cridats i tractats des de el programa principal, però estes classes queden aïllades entre sí, imaginem que hi tenim una classe Assignatura i un altra Professors, és lògic pensar que cada Assignatura estarà impartida per un o més professors, però si no les relacionem entre sí, mai hi tindrem forma de saber qui imparteix que.
Ara veurem com es poden relacionar les classes entre sí, i a més a més, veurem que segons el tipus de dependència entre les classes, les relacions poden ser de distints tipus.

1.1 Associacions

La associació és l'agrupament **d'un o diversos objectes i valors dins d'una classe**. Alhora, una associació pot ser de tipus **agregació**, és a dir, direm que una classe **'té'** diferents objectes d'altra independent ó pot ser una **composició**, com el nom indica, la classe està composta per altres objectes els quals donen sentit a la primera, de manera que si desapareix la primera, els objectes que la componen deixen de tindre sentit.

La composició crea una relació **'té'** o **'està compost per'**.

Un compte bancari té un titular i un o més autoritzats (totes són persones amb dni, nom, adreça, telèfon, etc.). A més del saldo, està compost per una série de moviments (cada moviment té associada un tipus, data, quantitat, concepte, origen o destinació, etc.). La diferencia està en que si el compte desapareix, les persones poden seguir sent clients del banc, tenir altres comptes o pasar a formar part del personal del banc, en canvi els moviments només tenen sentit per a eixe compte en concret, de manera que si el compte desapareix, estos moviments perden el seu valor.

```
public class CompteBancari {
	Persona titular;
	double  saldo;
	Moviment moviments[];
	Persona autoritzats[];
	...
}

public class Persona {
	String dni, nom, adreça, telèfon;
	...
}

public class Moviment {
	int tipus;
	Date  data;
	double quantitat;
	String concepte, origen, destinació;
	...
}
```

2. HERÈNCIA

2.1 Introducció

L'herència és una de les capacitats més importants i distintives de la POO. Consisteix en derivar o **estendre una classe nova a partir d'una altra ja existent de manera que la classe nova hereta tots els atributs i mètodes de la classe ja existent**.

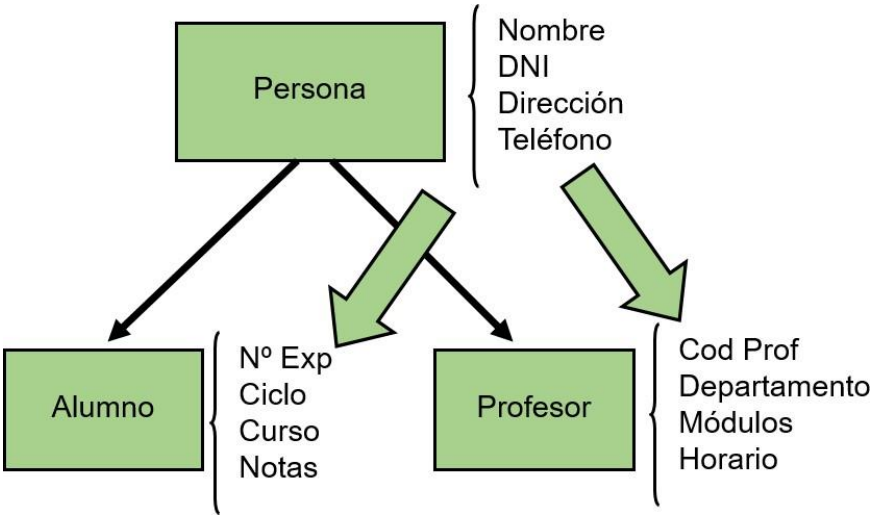
A la classe ja existent se la denomina **superclase**, classe **base** o classe **pare**. A la nova classe se la denomina **subclasse**, classe **derivada** o classe **filla**.

> Quan derivem (o estenem) una nova classe, aquesta hereta totes les dades i mètodes membre de la classe existent.

Per exemple, si tenim un programa que treballarà amb alumnes i professors, aquests tindran atributs comuns com el nom, DNI, adreça o telèfon. Però cadascun d'ells tindran atributs específics que no tinguen els altres. Per exemple els i les alumnes tindran el número d'expedient, el cicle i curs que cursen i les seues notes; per la seua part els i les docents tindran el codi de professor, el departament al qual pertanyen, els mòduls que imparteixen i el seu horari.

Per tant, en aquest cas el millor és declarar una classe *Persona* amb els atributs comuns (Nom, DNI, Adreça, Telèfon) i dos sub-classes *Alumne* i *Professor* que hereten de Persona (a més de tindre els seus propis atributs).

Es important remarcar que *Alumne* i *Professor* també heretaran tots els mètodes de Persona.



A Java s'utilitza la paraula reservada **extends** per a indicar herència:

```
public class Alumne extends Persona {
      ...
}

public class Professor extends Persona {
      …
}
```

2.2 Constructors de classes derivades

El constructor d'una classe derivada ha d'encarregar-se de construir els atributs que estiguen definits en la classe base a més dels seus propis atributs.

Dins del constructor de la classe derivada, per a cridar al constructor de la classe base es deu utilitzar el mètode reservat ***super()*** passant-li com a argument els paràmetres que necessite.

Si no es crida explícitament al constructor de la classe base mitjançant super() el compilador cridarà automàticament al constructor per defecte de la classe base. Si no té constructor per defecte el compilador generarà un error.

2.3 Mètodes heretats i sobreescrits

Hem vist que una subclasse hereta els atributs i mètodes de la superclasse; a més, es poden incloure nous atributs i nous mètodes.

D'altra banda, pot ocórrer que algun dels mètodes que existeixen en la superclasse no ens valguen en la subclasse (tal com estan programats) i necessitem adequar-los a les característiques de la subclasse. Això pot fer-se mitjançant la sobreescriurà de mètodes.

Un mètode està sobreescrit o reimplementat quan es programa de nou en la classe derivada. Per exemple el mètode *mostrarPersona()* de la classe *Persona* el necessitaríem sobreescriure en les classes *Alumne* i *Professor* per a mostrar també els nous atributs.

El mètode sobreescrit en la classe derivada podria reutilitzar el mètode de la classe base, si és necessari, i a continuació imprimir els nous atributs. En Java podem accedir a mètodes definits en la classe base mitjançant super.metodo().

El mètode mostrarPersona sobreescrit en les classes derivades podria ser:

```
super.mostrarPersona();  // Anomenada al mètode de la classe base
System.out.println(…); // Imprimim els atributs exclusius de la classe derivada
```

2.4 Classes i mètodes final

- Una classe ***final*** no pot ser heretada.
- Un mètode ***final*** no pot ser sobreescrit per les subclasses.

2.5 Accés a membres derivats

Encara que una subclasse inclou tots els membres de la seua superclasse, no podrà accedir a aquells que hagen sigut declarats com *private*.

Si en l'exemple 3 intentàrem accedir des de les classes derivades als atributs de la classe *Persona* (que són privats) obtindríem un error de compilació.

També podem declarar els atributs com *proteted*. D'aquesta manera podran ser accedits des de les classes heretades, (mai des d'altres classes).

- Els atributs declarats com ***protected*** son públics per a les classes heretades i privats per a les altres classes.

2.6 Exemple 3

En aquest exemple crearem les classe *Persona* i les seues classes heretades: *Alumne* i *Professor*.

En la **classe *Persona*** crearem el constructor, un mètode per a mostrar els atributs i els getters i setters. Les **classes *Alumne* i *Professor*** heretaran de la classe *Persona* (utilitzant la paraula reservada *extends*) i cadascuna tindrà els seus propis atributs, un constructor que anomenarà també al constructor de la classe *Persona* (utilitzant el mètode *super()*), un mètode per a mostrar els seus atributs, que també cridarà al mètode de Persona i els getters i setters.

És interessant veure com s'ha sobreescrit el mètode *mostrarPersona()* en les classes heretades. El mètode es diu igual i fa ús de la paraula reservada *super* per a cridar al mètode de mostrarPersona() de Persona . En la crida del *main,* tant l'objecte a *(Alumne*) com l'objecte *profe* (*Professor*) poden fer ús del mètode *mostrarPersona()*.

Persona

```java
public class Persona {

    private String nombre;
    private String dni;
    private String direccion;
    private int telefono;

    public Persona(String nom, String dni, String direc, int tel)
    {
        this.nombre = nom;
        this.dni = dni;
        this.direccion = direc;
        this.telefono = tel;
    }

    public void mostrarPersona()
    {
        System.out.println("Nombre: " + this.nombre);
        System.out.println("DNI: " + this.dni);
        System.out.println("Dirección: " + this.direccion);
        System.out.println("Teléfono: " + this.telefono);
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }

    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }

    public int getTelefono() {
        return telefono;
    }

    public void setTelefono(int telefono) {
        this.telefono = telefono;
    }

}
```

Alumne

(hereta de Persona)

```java
import java.util.ArrayList;
import java.util.Iterator;


public class Alumno extends Persona{

    private int exp;
    private String ciclo;
    private int curso;
    private ArrayList notas;

    // Al constructor hemos de pasarle los atributos de la clase Alumno y la de Persona
    public Alumno(String nom, String dni, String direc, int tel, int exp, String ciclo, int curso, ArrayList notas)
    {
        // Llamamos al constructor de la clase Persona
        super(nom, dni, direc, tel);

        this.exp = exp;
        this.ciclo = ciclo;
        this.curso = curso;
        this.notas = notas;
    }

    public void mostrarPersona()
    {
        // Llamamos al método de la clase madre para que muestre los datos de Persona
        super.mostrarPersona();

        System.out.println("Núm. expediente: " + this.exp);
        System.out.println("Ciclo: " + this.ciclo);
        System.out.println("Curso: " + this.curso);
        System.out.println("Notas:");
        for( Iterator it = this.notas.iterator(); it.hasNext(); )
        {
            System.out.println("\tNota: " + it.next());
        }
    }
    public int getExp() {
        return exp;
    }

    public void setExp(int exp) {
        this.exp = exp;
    }

    public String getCiclo() {
        return ciclo;
    }

    public void setCiclo(String ciclo) {
        this.ciclo = ciclo;
    }

    public int getCurso() {
        return curso;
    }

    public void setCurso(int curso) {
        this.curso = curso;
    }

    public ArrayList getNotas() {
        return notas;
    }

    public void setNotas(ArrayList notas) {
        this.notas = notas;
    }
}
```

Professor

(hereta de Persona)

```java
import java.util.ArrayList;
import java.util.Iterator;

public class Profesor extends Persona{

    private int cod;
    private String depto;
    private ArrayList modulos;
    private String horario;

    // Al constructor hemos de pasarle los atributos de la clase Peofesor y la de Persona
    public Profesor(String nom, String dni, String direc, int tel, int cod, String depto, ArrayList mod, String horario)
    {
        // Llamamos al constructor de la clase Persona
        super(nom, dni, direc, tel);

        this.cod = cod;
        this.depto = depto;
        this.modulos = mod;
        this.horario = horario;
    }
    public void mostrarPersona()
    {
        // Llamamos al método de la clase madre para que muestre los datos de Persona
        super.mostrarPersona();

        System.out.println("Código: " + this.cod);
        System.out.println("Departamento: " + this.depto);
        System.out.println("Horario: " + this.horario);
        System.out.println("Modulos:");
        for( Iterator it = this.modulos.iterator(); it.hasNext(); )
        {
            System.out.println("\tMódulo: " + it.next());
        }
    }

    public int getCod() {
        return cod;
    }

    public void setCod(int cod) {
        this.cod = cod;
    }

    public String getDepto() {
        return depto;
    }

    public void setDepto(String depto) {
        this.depto = depto;
    }

    public ArrayList getModulos() {
        return modulos;
    }

    public void setModulos(ArrayList modulos) {
        this.modulos = modulos;
    }

    public String getHorario() {
        return horario;
    }

    public void setHorario(String horario) {
        this.horario = horario;
    }
}
```

Programa Principal

```java
import java.util.ArrayList;

public class Herencia {

    public static void main(String[] args) {

        // Probamos la clase persona
        // Llamamos al contructor con el nombre, dni, dirección y teléfono
        Persona p = new Persona("Pepe", "00000000T", "C/ Colón", 666666666);

        System.out.println("Mostramos una persona");
        p.mostrarPersona();

        //Probamos la clase Alumno

        // Creamos las notas
        ArrayList notas = new ArrayList();

        notas.add(7);
        notas.add(9);
        notas.add(6);

        // Llamamos al contructor con el nombre, dni, dirección, teléfono, expediente, ciclo, curso y las notas
        Alumno a = new Alumno("María", "12345678Z", "P/ Libertad", 111111111, 1, "DAW", 1, notas);

        System.out.println("--------------------");
        System.out.println("Mostramos un alumno");
        a.mostrarPersona();


        //Probamos la clase Profesor

        // Creamos los módulos
        ArrayList modulos = new ArrayList();

        modulos.add("Programación");
        modulos.add("Lenguajes de marcas");
        modulos.add("Entornos de desarrollo");

        // Llamamos al contructor con el nombre, dni, dirección, teléfono, expediente, ciclo, curso y las notas
        Profesor profe = new Profesor("Juan", "00000001R", "C/ Java", 222222222, 3, "Informática", modulos, "Mañanas");

        System.out.println("--------------------");
        System.out.println("Mostramos un profesor");

        profe.mostrarPersona();
    }
}
```

Eixida:

```
Output - Herencia (run)
run:
Mostramos una persona
Nombre: Pepe
DNI: 00000000T
Dirección: C/ Colón
Teléfono: 666666666
--------------------
Mostramos un alumno
Nombre: María
DNI: 12345678Z
Dirección: P/ Libertad
Teléfono: 111111111
Núm. expediente: 1
Ciclo: DAW
Curso: 1
Notas:
        Nota: 7
        Nota: 9
        Nota: 6
--------------------
--------------------
Mostramos un profesor
Nombre: Juan
DNI: 00000001R
Dirección: C/ Java
Teléfono: 222222222
Código: 3
Departamento: Informática
Horario: Mañanas
Modulos:
        Módulo: Programación
        Módulo: Lenguajes de marcas
        Módulo: Entornos de desarrollo
BUILD SUCCESSFUL (total time: 0 seconds)
```

3. POLIMORFISME

La sobreescritura de mètodes constitueix la base d'un dels conceptes més potents de Java: la **selecció dinàmica de mètodes**, que és un mecanisme mitjançant el qual l'anomenada a un mètode sobreescrit es resol en temps d'execució i no durant la compilació. La selecció dinàmica de mètodes és important perquè permet implementar el polimorfisme durant el temps d'execució. Una variable de referència a una superclase es pot referir a un objecte d'una subclasse. Java es basa en això per a resoldre anomenades a mètodes sobreescrits en el temps d'execució.

El que determina la versió del mètode que serà executat és el tipus d'objecte al qual es fa referència i no el tipus de variable de referència.

El polimorfisme és fonamental en la programació orientada a objectes perquè permet que una classe general especifique mètodes que seran comuns a totes les classes que es deriven d'aqueixa mateixa classe. D'aquesta manera les subclasses podran definir la implementació d'algun o de tots aqueixos mètodes.

La superclase proporciona tots els elements que una subclasse pot usar directament. També defineix aquells mètodes que les subclasses que es deriven d'ella han d'implementar per si mateixes. D'aquesta manera, combinant l'herència i la sobreescritura de mètodes, una superclase pot definir la forma general dels mètodes que s'usaran en totes les seues subclasses

3.1 Exemple 4

Anem a probar un exemple senzill però que resumeix tot el que és important del polimorfisme.

Anem a crear la **classe *Mare*** amb un mètode *llamame()*. A continuació crearem **dues classes derivades** d'aquesta: ***Hija1*** i ***Hija2***, sobreescrivint el mètode *llamame()*. En el *main* crearem un objecte de cada classe i els assignarem a una variable de tipus *Mare* (anomenada *madre2*) amb la qual anomenarem al mètode *llamame()* dels tres objectes.

És important observar que la variable Mare madre2 es pot assignar a objectes de classe Hija1 i Hija2. Això és possible perquè Hija1 i Hija2 també són de tipus Mare (a causa de l'herència).

També és important veure que la variable Mare madre2 cridarà al mètode llamame() de la classe de l'objecte al qual fa referència (a causa del polimorfisme).

Observe's les anomenades madre2.llamame() de les línies 36 d'ara en avant:

- En el primer s'invoca al mètode llamame() de la classe Mare perquè 'madre2' fa referència a un objecte de la classe Mare.
- En el segon s'invoca al mètode llamame() de la classe Hija1 perquè ara 'madre2' fa referència a un objecte de la classe Hija1.
- En el tercer s'invoca al mètode llamame() de la classe Hija2 perquè ara 'madre2' fa referència a un objecte de la classe Hija2.

```java
class Madre {
    void llamame(){
        System.out.println("Estoy en la clase Madre");
    }
}

class Hija1 extends Madre {
    void llamame(){
        System.out.println("Estoy en la subclase Hija1");
    }
}

class Hija2 extends Madre {
    void llamame(){
        System.out.println("Estoy en la subclase Hija2");
    }
}

class Ejemplo {
    public static void main(String args[]){
        // Creamos un objeto de cada clase
        Madre madre = new Madre();
        Hija1 h1 = new Hija1();
        Hija2 h2 = new Hija2();

        // Declaramos otra variable de tipo Madre
        Madre madre2;

        // Asignamos a madre2 el objeto madre
        madre2 = madre;
        madre2.llamame();

        // Asignamos a madre2 el objeto h1 (Hija1)
        madre2 = h1;
        madre2.llamame();

        // Asignamos a madre2 el objeto h2 (Hija2)
        madre2 = h2;
        madre2.llamame();
    }
}
```

Eixida:

```
run:
Estoy en la clase Madre
Estoy en la subclase Hija1
Estoy en la subclase Hija2
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. CLASSES ABSTRACTES

Una classe abstracta és **una classe que declara l'existència d'alguns mètodes però no la seua implementació** (és a dir, conté la capçalera del mètode però no el seu codi). Els mètodes sense implementar són mètodes abstractes.

Una classe abstracta pot contindre tant mètodes abstractes (sense implementar) com no abstractes (implementats). Però almenys un ha de ser abstracte.

Per a declarar una classe o mètode com a abstracte s'utilitza el modificador **abstract**.

> Una classe abstracta **no es pot instanciar**, però **sí es pot heretar**. Les subclasses hauran d'implementar obligatòriament el codi dels mètodes abstractes (llevat que també es declaren com a abstractes).

Les classes abstractes són útils quan necessitem definir una forma generalitzada de classe que serà compartida per les subclasses, deixant part del codi en la classe abstracta (mètodes "normals") i delegant una altra part en les subclasses (mètodes abstractes).

> No poden declarar-se constructors o mètodes estàtics abstractes.

La finalitat principal d'una classe abstracta és crear una classe heretada a partir d'ella. Per això, en la pràctica és obligatori aplicar herència (si no, la classe abstracta no serveix per a res). El cas contrari és una classe *final*, que no pot heretar-se com ja hem vist. Per tant una classe no pot ser *abstract* i *final* al mateix temps.

Per exemple, aquesta classe abstracta Principal ten dos mètodes: un concret i un altre abstracte.

```
public abstract class Principal {
     // Mètode concret amb implementació
     public void metodeConcret() {
           ...
     }
    // Mètode abstracte sense implementació
    public abstract void metodeAbstracte();
}
```

Aquesta subclasse hereta de Principal tots dos mètodes, però està obligada a implementar el codi del mètode abstracte.

```
class Secundària extends Principal {
     // Implementació concreta
     public void metodeAbstracte() {
           ...
     }
}
```

5. INTERFICIES

Una interfície és una **declaració d'atributs i mètodes sense implementació** (sense definir el codi dels mètodes). S'utilitzen per a definir el conjunt mínim d'atributs i mètodes de les classes que implementen aquesta interfície. En certa manera, és paregut a una classe abstracta amb tots els seus membres abstractes.

Si una classe és una plantilla per a crear objectes, **una interfície és una plantilla per a crear classes**.


Una interfície és una declaració d'atributs i mètodes sense implementació.

Mitjançant la construcció d'una interfície, el o la programadora pretén especificar què caracteritza a una col·lecció d'objectes i, igualment, especificar quin comportament han de reunir els objectes que vulguen entrar dins d'eixa categoria o col·lecció.

En una interfície també es poden declarar constants que defineixen el comportament que han de suportar els objectes que vulguen implementar aqueixa interfície.

La sintaxi típica d'una interfície és la següent:

```
public interface Nom {
// Declaració d'atributs i mètodes (sense definir codi)
}
```

Si una interfície defineix un tipus però aqueix tipus no proveeix de cap mètode, podem preguntar-nos: *per a què serveixen llavors les interfícies a Java?*

La implementació (herència) d'una interfície no podem dir que evite la duplicitat de codi o que afavorisca la reutilització de codi perquè realment no proveeixen codi.

En canvi sí que podem dir que reuneix els altres dos avantatges de l'herència: afavorir el manteniment i l'extensió de les aplicacions. Per què? Perquè **en definir interfícies permetem l'existència de variables polimòrfiques i la invocació polimòrfica de mètodes.**

Un aspecte fonamental de les interfícies a Java és **separar l'especificació d'una classe (què fa) de la implementació (com ho fa).** Això s'ha comprovat que dona lloc a programes més robustos i amb menys errors.

És important tindre en compte que:

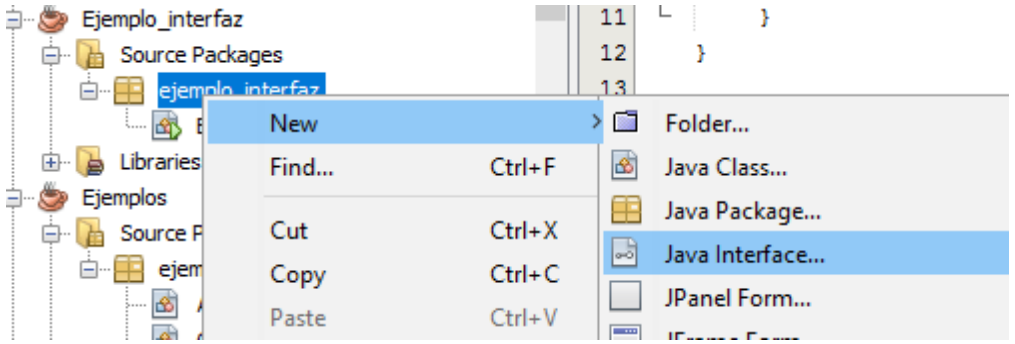
- Una interfície no es pot instanciar en objectes, només serveix per a implementar classes.
- Una classe pot implementar diverses interfícies (separades per comes).
- Una classe que implementa una interfície ha de proporcionar implementació per a tots i cadascun dels mètodes definits en la interfície.
- Les classes que implementen una interfície que té definides constants poden usar-les en qualsevol part del codi de la classe, simplement indicant el seu nom.

Si per exemple la classe *Cercle* implementa la interfície *Figura* la sintaxi seria:

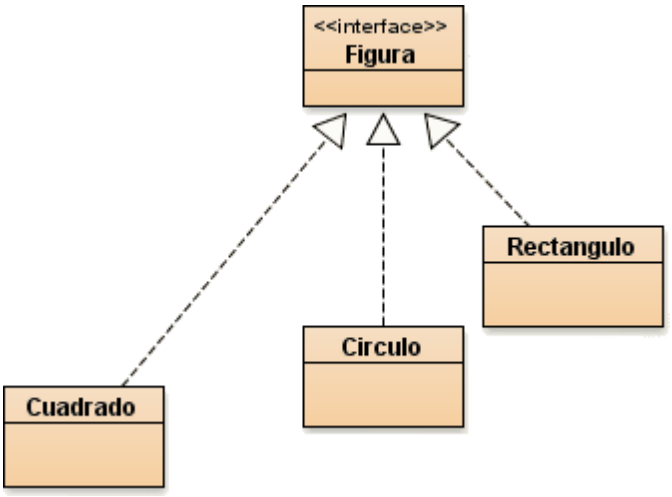
```
public class Cercle implements Figura {
      ...
}
```

5.1 Exemple 6

En aquest exemple crearem una interfície Figura i posteriorment la implementarem en vàries classes. Per a crear una interfície hem de punxar amb el botó dret sobre el paquet on la vulguem crear i després NEW > Java Interface.



Veurem un exemple simple de definició i ús de interfície a Java. Les classes que usarem i les seues relacions es mostren en l'esquema:




```
public interface Figura {
    float PI = 3.1416f;   // Por defecto public static  final. La f final indica que el número es float
    float area(); // Por defecto abstract public
}
```

```java
public class Cuadrado implements Figura {
    private float lado;

    public Cuadrado (float lado) {
        this.lado = lado;
    }

    public float area() {
        return lado*lado;
    }
}
```

```java
public class Rectangulo implements Figura {
    private float lado;
    private float altura;

    public Rectangulo (float lado, float altura) {
        this.lado = lado;
        this.altura = altura;
    }

    public float area() {
        return lado*altura;
    }
}
```

```java
public class Circulo implements Figura {
    private float diametro;

    public Circulo (float diametro) {
        this.diametro = diametro;
    }

    public float area() {
        return (PI*diametro*diametro/4f);
    }
}
```

```java
public static void main(String[] args) {
    // TODO code application logic here
    Figura cuad1 = new Cuadrado (3.5f);
    Figura cuad2 = new Cuadrado (2.2f);
    Figura cuad3 = new Cuadrado (8.9f);

    Figura circ1 = new Circulo (3.5f);
    Figura circ2 = new Circulo (4f);

    Figura rect1 = new Rectangulo (2.25f, 2.55f);
    Figura rect2 = new Rectangulo (12f, 3f);

    ArrayList serieDeFiguras = new ArrayList();

    serieDeFiguras.add (cuad1);
    serieDeFiguras.add (cuad2);
    serieDeFiguras.add (cuad3);

    serieDeFiguras.add (circ1);
    serieDeFiguras.add (circ2);
    serieDeFiguras.add (rect1);
    serieDeFiguras.add (rect2);

    float areaTotal = 0;
    Iterator it = serieDeFiguras.iterator(); //creamos un iterador

    while (it.hasNext()){
       Figura tmp = (Figura)it.next();
       areaTotal = areaTotal + tmp.area();
    }

    System.out.println ("Tenemos un total de " + serieDeFiguras.size() + " figuras y su área total es de "  +
    areaTotal + " uds cuadradas");
}
```

El resultat d'execució podria ser una cosa així:

```
Output - UD9 (run)
run:
Tenemos un total de 7 figuras y su área total es de 160.22504 uds cuadradas
BUILD SUCCESSFUL (total time: 0 seconds)
```

En aquest exemple **la interfície Figura defineix un tipus de dada.** Per això podem crear un ArrayList de figures on inserim quadrats, cercles, rectangles, etc. (polimorfisme). Això ens permet donar-li un tractament comú a totes les figures: Mitjançant un bucle while recorrem la llista de figures i cridem al mètode area() que serà diferent per a cada classe de figura.

6. Agraïments

Anotacions actualitzades i adaptats al CEEDCV a partir de la següent documentació:

[1] Anotacions Programació de José Antonio Díaz-Alejo. IES Camp de Morvedre.