UD3

LENGUAJE DE PROGRAMACIÓN JAVA

MP_0485
Programación

3.2 Herencia

Introducción

La Programación Orientada a Objetos (POO) hace que los problemas sean más sencillos, al permitir dividir el problema. Esta división se hace en objetos, de forma que cada objeto funcione de forma totalmente independiente.

Un **objeto** es un elemento del programa que posee sus propios datos y su propio funcionamiento.

Una **clase** describe un grupo de objetos que contienen una información similar (atributos) y un comportamiento común (métodos).

Antes de poder utilizar un objeto, se debe definir su clase. La clase es la definición de un tipo de objeto.

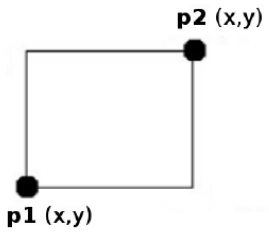
Composición

La composición es el **agrupamiento de uno o varios objetos y valores dentro de una clase**. La composición crea una relación **'tiene'** o **'está compuesto por'**.

Por ejemplo, un rectángulo está compuesto por dos puntos (cada uno con sus coordenadas x,y).

```
public class Rectangulo {
      Punto p1;
      Punto p2;
      ...
}

public class Punto {
      int x, y;
      ...
}
```



Una cuenta bancaria tiene un titular y un autorizado (ambas son personas con dni, nombre, dirección, teléfono, etc.). Además del saldo, la cuenta tendrá registrado un listado de movimientos (cada movimiento tiene asociada un tipo, fecha, cantidad, concepto, origen o destino, etc.).

```
public class CuentaBancaria {
	Persona titular;
	Persona autorizado;
	double saldo;
	Movimiento movimientos[];
	...
}

public class Persona {
	String dni, nombre, dirección, teléfono;
	...
}

public class Movimiento {
	int tipo;
	Date fecha;
	double cantidad;
	String concepto, origen, destino;
	...
}
```

La composición de clases es una capacidad muy potente de la POO ya que permite diseñar software como un conjunto de clases que colaboran entre sí: Cada clase se especializa en una tarea concreta y esto permite dividir un problema complejo en varios subproblemas pequeños. También facilita la modularidad y reutilización del código.

Herencia

Introducción

La herencia es una de las capacidades más importantes y distintivas de la POO. Consiste en derivar **o extender una clase nueva a partir de otra ya existente de forma que la clase nueva hereda todos los atributos y métodos de la clase ya existente**.

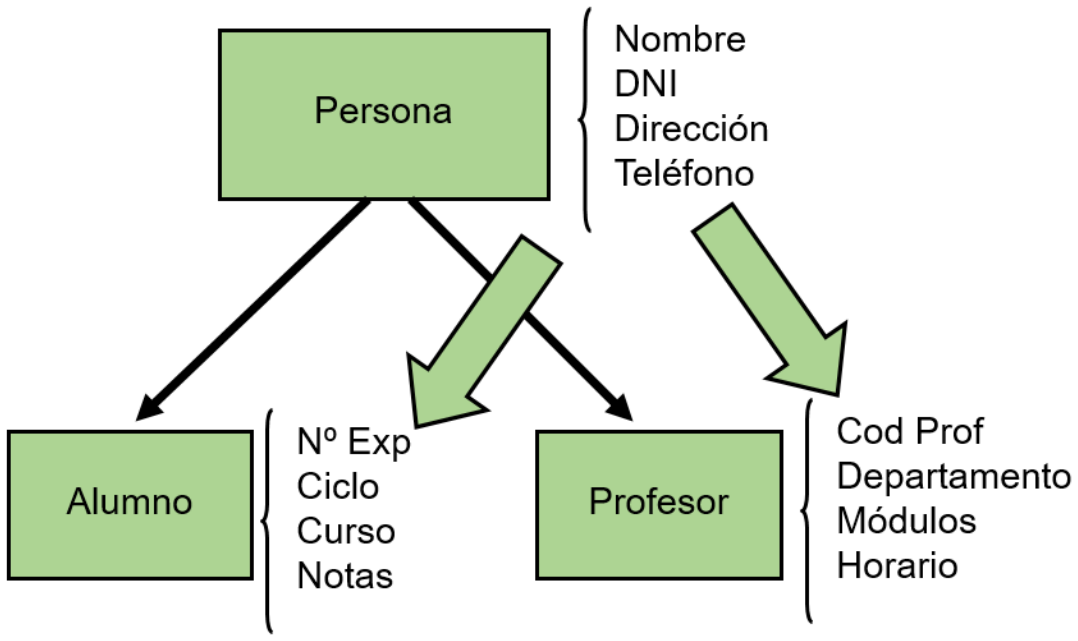
A la clase ya existente se la denomina **superclase**, clase **base o clase padre**. A la nueva clase se la denomina **subclase**, clase **derivada** o clase **hija**.

Cuando derivamos (o extendemos) una nueva clase, ésta hereda todos los datos y métodos miembro de la clase existente.

Por ejemplo, si tenemos un programa que va a trabajar con alumnos y profesores, éstos van a tener atributos comunes como el nombre, dni, dirección o teléfono. Pero cada uno de ellos tendrán atributos específicos que no tengan los otros. Por ejemplo, los alumnos tendrán el número de expediente, el ciclo y curso que cursan y sus notas; por su parte los profesores tendrán el código de profesor, el departamento al que pertenecen, los módulos que imparten y su horario.

Por lo tanto, en este caso lo mejor es declarar una clase *Persona* con los atributos comunes (Nombre, DNI, Dirección, Teléfono) y dos sub-clases *Alumno* y *Profesor* que hereden de Persona (además de tener sus propios atributos).

Es importante recalcar que *Alumno* y *Profesor* también heredarán todos los métodos de Persona.



En Java se utiliza la palabra reservada **extends** para indicar herencia:

```
public class Alumno extends Persona {
… }
public class Profesor extends Persona{
… }
```

Constructores de clases derivadas

El constructor de una clase derivada debe encargarse de construir los atributos que estén definidos en la clase base además de sus propios atributos.

Dentro del constructor de la clase derivada, para llamar al constructor de la clase base se debe utilizar el método reservado ***super()*** pasándole como argumento los parámetros que necesite. Si no se llama explícitamente al constructor de la clase base mediante super() el compilador llamará automáticamente al constructor por defecto de la clase base. Si no tiene constructor por defecto el compilador generará un error.

Métodos heredados y sobrescritos

Hemos visto que una subclase hereda los atributos y métodos de la superclase; además, se pueden incluir nuevos atributos y métodos.

Por otro lado, puede ocurrir que alguno de los métodos que existen en la superclase no nos sirvan en la subclase (tal y como están programados) y necesitemos adecuarlos a las características de la subclase. Esto puede hacerse mediante la sobreescritura de métodos:

Un método está sobreescrito o reimplementado cuando se programa de nuevo en la clase derivada. Por ejemplo el método *mostrarPersona()* de la clase *Persona* lo necesitaríamos sobreescribir en las clases *Alumno* y *Profesor* para mostrar también los nuevos atributos.

El método sobreescrito en la clase derivada podría reutilizar el método de la clase base, si es necesario, y a continuación imprimir los nuevos atributos. En Java podemos acceder a método definidos en la clase base mediante super.metodo().

El método mostrarPersona sobreescrito en las clases derivadas podría ser:

```
super.mostrarPersona(); // Llamada al método de la clase base
System.out.println(…); // Imprimimos los atributos exclusivos de la clase derivada
```

Una clase **final** no puede ser heredada.

Un método **final** no puede ser sobrescrito por las subclases.

Ejemplo

En este ejemplo vamos a crear las clases *Persona* y sus clases heredadas: *Alumno* y *Profesor*.

En la **clase *Persona*** crearemos el constructor, un método para mostrar los atributos y los getters y setters. Las **clases *Alumno* y *Profesor*** heredarán de la clase *Persona* (utilizando la palabra reservada *extends*) y cada una tendrá sus propios atributos, un constructor que llamará también al constructor de la clase *Persona* (utilizando el método *super()*), un método para mostrar sus atributos, que también llamará al método de *Persona* y los getters y setters.

Es interesante ver cómo se ha sobreescrito el método *mostrarPersona()* en las clases heredadas.

El método se llama igual y hace uso de la palabra reservada *super* para llamar al método de *mostrarPersona()* de *Persona*. En la llamada del *main* tanto el objeto a (*Alumno*) como el objeto *profe* (*Profesor*) pueden hacer uso del método *mostrarPersona()*.

```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;
    }

}
```

```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;
    }
}
```

```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;
    }
}
```

```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();
    }
}
```

Salida:

```
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)
```

Polimorfismo

La sobreescritura de métodos constituye la base de uno de los conceptos más potentes de Java: la **selección dinámica de métodos**, que es un mecanismo mediante el cual la llamada a un método sobrescrito se resuelve en tiempo de ejecución y no durante la compilación. La selección dinámica de métodos es importante porque permite implementar el polimorfismo durante el tiempo de ejecución. Una variable de referencia a una superclase se puede referir a un objeto de una subclase. Java se basa en esto para resolver llamadas a métodos sobrescritos en el tiempo de ejecución.

Lo que determina la versión del método que será ejecutado es el tipo de objeto al que se hace referencia y no el tipo de variable de referencia.

El polimorfismo es fundamental en la programación orientada a objetos porque permite que una clase general especifique métodos que serán comunes a todas las clases que se deriven de esa misma clase. De esta manera las subclases podrán definir la implementación de alguno o de todos esos métodos.

La superclase proporciona todos los elementos que una subclase puede usar directamente.

También define aquellos métodos que las subclases que se deriven de ella deben implementar por sí mismas. De esta manera, combinando la herencia y la sobreescritura de métodos, una superclase puede definir la forma general de los métodos que se usarán en todas sus subclases.

Ejemplo

Vamos a probar un ejemplo sencillo pero que resume todo lo importante del polimorfismo. Vamos a crear la **clase *Madre*** con un método *llamame()*. A continuación crearemos **dos clases derivadas** de ésta: ***Hija1*** e ***Hija2***, sobreescribiendo el método *llamame()*. En el *main* crearemos un objeto de cada clase y los asignaremos a una variable de tipo *Madre* (llamada *madre2*) con la que llamaremos al método *llamame()* de los tres objetos.

Es importante observar que la variable Madre madre2 se puede asignar a objetos de clase Hija1 e Hija2. Esto es posible porque Hija1 e Hija2 también son de tipo Madre (debido a la herencia).

También es importante ver que la variable Madre madre2 llamará al método llamame() de la clase del objeto al que hace referencia (debido al polimorfismo).

Obsérvese las llamadas madre2.llamame() de las líneas 36 en adelante:

• En el primero se invoca al método llamame() de la clase Madre porque 'madre2' hace referencia a un objeto de la clase Madre.

• En el segundo se invoca al método llamame() de la clase Hija1 porque ahora 'madre2' hace referencia a un objeto de la clase Hija1.

• En el tercero se invoca al método llamame() de la clase Hija2 porque ahora 'madre2' hace referencia a un objeto de la clase Hija2.

```
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();
    }
}
```

Salida:

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

Clases abstractas

Una clase abstracta es **una clase que declara la existencia de algunos métodos pero no su implementación** (es decir, contiene la cabecera del método pero no su código). Los métodos sin implementar son métodos abstractos.

Una clase abstracta puede contener tanto métodos abstractos (sin implementar) como no abstractos (implementados). Pero al menos uno debe ser abstracto.

Para declarar una clase o método como abstracto se utiliza el modificador **abstract**.

Una clase abstracta **no se puede instanciar**, pero **si heredar**. Las subclases tendrán que implementar obligatoriamente el código de los métodos abstractos (a no ser que también se declaren como abstractas):

Las clases abstractas **son útiles cuando necesitemos definir una forma generalizada de clase que será compartida por las subclases, dejando parte del código en la clase abstracta** (métodos "normales") y delegando otra parte en las subclases (métodos abstractos).

No pueden declararse constructores o métodos estáticos abstractos.

La finalidad principal de una clase abstracta es crear una clase heredada a partir de ella. Por ello, en la práctica es obligatorio aplicar herencia (si no, la clase abstracta no sirve para nada). El caso contrario es una clase *final*, que no puede heredarse como ya hemos visto. Por lo tanto, una clase no puede ser *abstract* y *final* al mismo tiempo.

Por ejemplo, esta clase abstracta Principal tienes dos métodos: uno concreto y otro abstracto.

```
public abstract class Principal {
     // Método concreto con implementación
     public void metodoConcreto() {
           ...
     }
    // Método abstracto sin implementación
    public abstract void metodoAbstracto();
}
```

Esta subclase hereda de Principal ambos métodos, pero está obligada a implementar el código del método abstracto.

```
class Secundaria extends Principal {
     // Implementación concreta
     public void metodoAbstracto() {
           ...
     }
}
```

Interfaces

Una interfaz es una **declaración de atributos y métodos sin implementación** (sin definir el código de los métodos). Se utilizan para definir el conjunto mínimo de atributos y métodos de las clases que implementen dicha interfaz. En cierto modo, es parecido a una clase abstracta con todos sus miembros abstractos.

Si una clase es una plantilla para crear objetos, **una interfaz es una plantilla para crear clases**.

> Una interfaz es una declaración de atributos y métodos sin implementación.

Mediante la construcción de un interfaz, el programador pretende especificar qué caracteriza a una colección de objetos e, igualmente, especificar qué comportamiento deben reunir los objetos que quieran entrar dentro de esa categoría o colección.

En una interfaz también se pueden declarar constantes que definen el comportamiento que deben soportar los objetos que quieran implementar esa interfaz.

La sintaxis típica de una interfaz es la siguiente:

```
public interface Nombre {
        // Declaración de atributos y métodos (sin definir código)
}
```

Si una interfaz define un tipo, pero ese tipo no provee de ningún método, podemos preguntarnos: ¿para qué sirven entonces las interfaces en Java?

La implementación (herencia) de una interfaz no podemos decir que evite la duplicidad de código o que favorezca la reutilización de código puesto que realmente no proveen código.

En cambio, sí podemos decir que reúne las otras dos ventajas de la herencia: favorecer el mantenimiento y la extensión de las aplicaciones. ¿Por qué? Porque **al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos.**

Un aspecto fundamental de las interfaces en Java es **separar la especificación de una clase (qué hace) de la implementación (cómo lo hace).** Esto se ha comprobado que da lugar a programas más robustos y con menos errores.

Es importante tener en cuenta que:

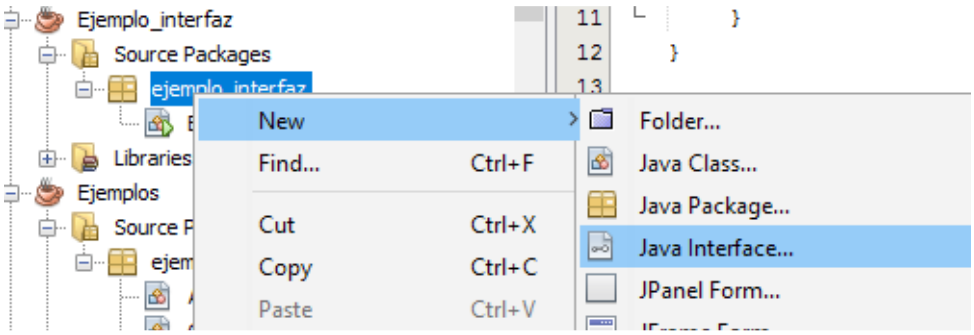
- Una interfaz no se puede instanciar en objetos, solo sirve para implementar clases.
- Una clase puede implementar varias interfaces (separadas por comas).
- Una clase que implementa una interfaz debe de proporcionar implementación para todos y cada uno de los métodos definidos en la interfaz.
- Las clases que implementan una interfaz que tiene definidas constantes pueden usarlas en
- cualquier parte del código de la clase, simplemente indicando su nombre.

Si por ejemplo la clase *Círculo* implementa la interfaz *Figura* la sintaxis sería:

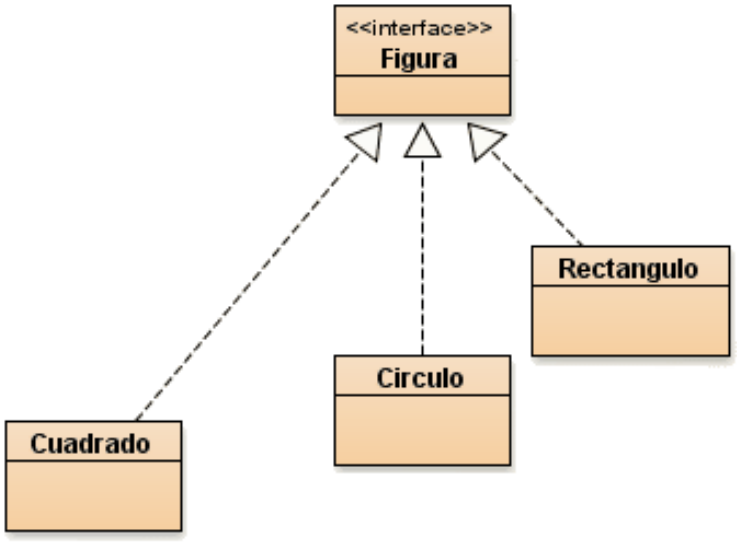
```
public class Circulo implements Figura {
        ...
}
```

Ejemplo

En este ejemplo vamos a crear una interfaz Figura y posteriormente implementarla en varias clases. Para crear una interfaz debemos pinchar con el botón derecho sobre el paquete donde la queramos crear y después NEW > Java Interface.



Vamos a ver un ejemplo simple de definición y uso de interfaz en Java. Las clases que vamos a usar y sus relaciones se muestran en el 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
}
```

```
12  public class Cuadrado implements Figura {
        private float lado;
14
15      public Cuadrado (float lado) {
16          this.lado = lado;
17      }
18
20      public float area() {
21          return lado*lado;
22      }
    }
```

```
12  public class Rectangulo implements Figura {
        private float lado;
        private float altura;
15
16      public Rectangulo (float lado, float altura) {
17          this.lado = lado;
18          this.altura = altura;
19      }
20
22      public float area() {
23          return lado*altura;
        }
24  }
```

```
12  public class Circulo implements Figura {
        private float diametro;
14
15      public Circulo (float diametro) {
16          this.diametro = diametro;
17      }
18
20      public float area() {
21          return (PI*diametro*diametro/4f);
        }
22  }
23
```

```
21      public static void main(String[] args) {
22          // TODO code application logic here
23          Figura cuad1 = new Cuadrado (3.5f);
24          Figura cuad2 = new Cuadrado (2.2f);
25          Figura cuad3 = new Cuadrado (8.9f);
26
27          Figura circ1 = new Circulo (3.5f);
28          Figura circ2 = new Circulo (4f);
29
30          Figura rect1 = new Rectangulo (2.25f, 2.55f);
31          Figura rect2 = new Rectangulo (12f, 3f);
32
33          ArrayList serieDeFiguras = new ArrayList();
34
35          serieDeFiguras.add (cuad1);
36          serieDeFiguras.add (cuad2);
37          serieDeFiguras.add (cuad3);
38
39          serieDeFiguras.add (circ1);
40          serieDeFiguras.add (circ2);
41          serieDeFiguras.add (rect1);
42          serieDeFiguras.add (rect2);
43
44          float areaTotal = 0;
45          Iterator it = serieDeFiguras.iterator(); //creamos un iterador
46
47          while (it.hasNext()){
48              Figura tmp = (Figura)it.next();
49              areaTotal = areaTotal + tmp.area();
50          }
51
52          System.out.println ("Tenemos un total de " + serieDeFiguras.size() + " figuras y su área total es de "  +
53          areaTotal + " uds cuadradas");
54      }
```

El resultado de ejecución podría ser algo así:

```
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)
```

Referencias

Apuntes elaborados a partir de la siguiente documentación:

[1] Apuntes Fernando Barber y Ricardo Ferris. Universidad de Valencia.

[2] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.

[3] Apuntes de Programación de Carlos Cacho y Raquel Torres. Ceedcv.

[4] Apuntes de Programación Edix Digital Workers.