Tema 7: Utilización avanzada de clases

Contenido

1. Relación entre clases

Se pueden distinguir diversos tipos de relaciones entre clases:

- **Clientela**. Cuando una clase utiliza objetos de otra clase (por ejemplo, al pasarlos como parámetros a través de un método).
- **Composición**. Cuando alguno de los atributos de una clase es un objeto de otra clase.
- **Anidamiento**. Cuando se definen clases en el interior de otra clase.
- **Herencia**. Cuando una clase comparte determinadas características con otra (clase padre), añadiéndole alguna funcionalidad específica (especialización).

2. Composición

Para indicar que una clase contiene objetos de otra clase no es necesaria **ninguna sintaxis especial**. Cada uno de esos objetos no es más que un atributo y, por tanto, debe ser declarado como tal:

```
class <nombreClase> {
   [modificadores] <NombreClase1> nombreAtributo1;
   [modificadores] <NombreClase2> nombreAtributo2;
…
}
```

EJERCICIO1: EjemploComposiciónRectangulo

Crea la clase **Punto** con atributos decimales X e Y.

Crea la clase **Rectangulo** teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase **Punto, vertice1 y vertice2**, en lugar de cuatro elementos de tipo **double**). Intenta reescribir los constructores:

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros, **x1**, **y1**, **x2**, **y2**, que cree un rectángulo con los vértices (x1, y1) y (x2, y2).
3. Un constructor con dos parámetros, **punto1**, **punto2**, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros, **base** y **altura**, que cree un rectángulo donde el vértice inferior izquierdo esté ubicado en la posición (0,0) y que

tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor copia.
6. Añade dos métodos: CalcularSuperficie() y CalcularPerimetro()

En la clase **Principal**

1. Crea cinco rectángulos, cada uno con uno de los constructores.
2. Calcula el área y el perímetro de los mismos.

3. Herencia

Sintaxis:

```
[modificador] class ClasePadre {
    // Cuerpo de la clase
    …
}

[modificador] class ClaseHija extends ClasePadre {
// Cuerpo de la clase
…
}
```

EJERCICIO2: EjemploClaseHerenciaPersona

Crea las clases Persona, Alumno y Profesor.

```
public class Persona {
    String nombre;
    String apellidos;
    String fechaNacim; //GregorianCalendar fechaNacim;
    …
}
public class Alumno extends Persona {
    String grupo;
    double notaMedia;
    …
}
public class Profesor extends Persona {
    String especialidad;
    double salario;
    …
}
```

Dadas las clases **Persona**, **Alumno** y **Profesor** que has utilizado anteriormente, implementa métodos **get** y **set** en la clase **Persona** para trabajar con sus tres atributos y en las clases **Alumno** y **Profesor** para trabajar con sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para **Persona** van a ser heredados en **Alumno** y en **Profesor**.

Redefinición de métodos hererados

Métodos sobrescritos. Anotación @Override

Métodos sobrescritos. Anotación @Override
Un método en una clase hija sobreescribe a un método de la clase padre cuando lleva exactamente su misma cabecera; de forma que un objeto instanciado como objeto de la clase hija aplicará el método sobreescrito llegado el caso y no el de la clase padre.

Redefine el método getNombre de tal forma que devuelva "ALUMNO: nombre del alumno" o "PROFESOR: nombre del profesor"

Ampliación de métodos heredados: uso de super

Un método en una clase hija sobreescribe y amplía a un método de la clase padre cuando lleva exactamente su misma cabecera y llama al método de la clase padre utilizando el super.metodoDeLaClasePadre; de forma que un objeto instanciado como objeto de la clase hija aplicará el método sobreescrito y ampliado llegado el caso, es decir el de la clase padre y la parte ampliada de la clase hija.

Ejercicio 3:

Define el método mostrar en las clases Persona, Alumno y Profesor, de manera que se muestren los datos personales de alumno o profesor, pero el método mostrar en las clases hijas debe **ampliar** el de la clase padre, aprovechando la definición de éste.

Constructores y herencia

Si la clase **Persona** tuviera un constructor de este tipo:

```
public Persona (String nombre, String apellidos, GregorianCalendar fechaNacim){

        this.nombre= nombre;

        this.apellidos= apellidos;

                this.fechaNacim=fechaNacim;

        //this.fechaNacim= (GregorianCalendar) fechaNacim.clone();

    }
```

Podrías llamarlo desde un constructor de una clase derivada (por ejemplo, **Alumno**) de la siguiente forma:

```
public Alumno (String nombre, String apellidos, GregorianCalendar fechaNacim, String grupo, double notaMedia){
    super (nombre, apellidos, fechaNacim);
    this.grupo= grupo;
    this.notaMedia= notaMedia;
}
```

Ejercicio 4:

Escribe un constructor para la clase Profesor y otro para la clase Alumno, del modo anterior.

En la clase **Principal** crea tres objetos: una Persona, un Alumno y un Profesor y aplícales el método mostrar.

4. Clases abstractas

Las **clases abstractas** se declaran mediante el modificador **abstract**:

```
[modificador_acceso] abstract class nombreClase  [herencia] [interfaces] {
    …
}
```

Cuando trabajes con **clases abstractas** debes tener en cuenta:

- Una **clase abstracta** sólo puede usarse para crear nuevas clases derivadas. No se puede hacer un **new** de una **clase abstracta**. Se produciría un **error de compilación**.
- Una **clase abstracta** puede contener **métodos totalmente definidos** (**no abstractos**) y **métodos sin definir** (**métodos abstractos**).

EJERCICIO5: EjemploClaseAbstractaPersona.

Copia el proyecto anterior y haz la clase Persona abstracta

Métodos abstractos

Un método se declara como abstracto mediante el uso del modificador **abstract** (como en las **clases abstractas**):

[modificador_acceso] **abstract** <tipo> <nombreMetodo> ([parámetros]) [excepciones];

Estos métodos tendrán que ser **obligatoriamente redefinidos** (en realidad "definidos", pues aún no tienen contenido) en las **clases derivadas**. Si en una **clase derivada** se deja algún **método abstracto sin implementar**, esa **clase derivada** será también una **clase abstracta**.

Cuando una clase contiene un método abstracto tienen que declararse como abstracta obligatoriamente.

Ejercicio 6:

Basándote en la jerarquía de clases **Persona**, **Alumno**, **Profesor**, crea un método abstracto llamado mostrar para la clase **Persona**. Dependiendo del tipo de persona (alumno o profesor) el método mostrar tendrá que mostrar unos u otros datos personales (habrá que hacer implementaciones específicas en cada clase derivada).

Una vez hecho esto, implementa completamente las tres clases (con todos sus atributos y métodos) y utilízalas en un pequeño programa de ejemplo que acabarás de diseñar en la clase **Principal**.

Trata de crear un objeto Persona.

Crea un objeto de tipo **Alumno** y otro de tipo **Profesor**, con información y muestre **esa información en la pantalla a través del método mostrar.**

Clases y métodos finales

Una clase declarada como final no puede ser heredada, es decir, **no puede tener clases derivadas**. La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas):

[modificador_acceso] **final** class nombreClase [herencia] [interfaces]

Un **método** también puede ser declarado como **final**, en tal caso, ese método no podrá ser redefinido en una **clase derivada**:

[modificador_acceso] **final** <tipo> <nombreMetodo> ([parámetros]) [excepciones]

Ejercicio 7:
Crea un método en la clase Persona, que pueda ser común a todos los objetos de tipo Persona y decláralo **final**.
Trata de ampliar ese método en la clase Alumno.
Trata de sobreesccribir ese método en la clase Alumno.

5. Iterfaces

Una **interfaz** en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar.

Ejercicio 8: EjemploInterfazImprimible

Crea una interfaz en Java cuyo nombre sea **Imprimible** y que contenga algunos métodos útiles para mostrar el contenido de una clase:

1. Método **devolverContenidoString**, que crea un String con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves:

"{<nombre_atributo_1>=<valor_atributo_1>,
<nombre_atributo_n>=<valor__atributo_n>}".

2. Método **devolverContenidoArrayList**, que crea un ArrayList de String con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve.

Implementación de interfaces

Todas las clases que implementan una determinada **interfaz** están obligadas a proporcionar una **definición (implementación) de los métodos de esa interfaz**, adoptando el modelo de comportamiento propuesto por ésta.

Dada una **interfaz**, cualquier clase puede especificar dicha **interfaz** mediante el mecanismo denominado **implementación de interfaces**. Para ello se utiliza la palabra reservada **implements**:

class NombreClase **implements** NombreInterfaz {

De esta manera, la clase está diciendo algo así como "**la interfaz indica los métodos que debo implementar, pero voy a ser yo (la clase) quien los implemente**".

Es posible indicar varios nombres de **interfaces** separándolos por comas:

class NombreClase **implements** NombreInterfaz1, NombreInterfaz2,... {

Cuando una clase implementa una **interfaz**, tiene que redefinir sus métodos nuevamente con **acceso público**. Con otro tipo de acceso se producirá un **error de compilación**. Es decir, que del mismo modo que no se podían restringir permisos de acceso en la **herencia de clases**, tampoco se puede hacer en la **implementación de interfaces**.

Una vez implementada una **interfaz** en una clase, los métodos de esa interfaz tienen exactamente el mismo tratamiento que cualquier otro método, sin ninguna diferencia, pudiendo ser invocados, heredados, redefinidos, etc

Ejercicio 9:

Haz que las clases Alumno y Profesor implementen la interfaz Imprimible.

NOTA: Vamos a aprovechar que ambas clases son **subclases** de una misma **superclase** (heredan de la misma) y hacer que la interfaz **Imprimible** sea implementada directamente por la **superclase** (**Persona**) y de este modo ahorrarnos bastante código. Así no haría falta indicar explícitamente que **Alumno** y **Profesor** implementan la interfaz **Imprimible**, pues lo estarán haciendo de forma implícita al heredar de una clase que ya ha implementado esa interfaz (la clase **Persona**, que es padre de ambas).

Una vez que los métodos de la **interfaz** estén implementados en la clase **Persona**, tan solo habrá que redefinir o ampliar los métodos de la **interfaz** para que se adapten a cada **clase hija** específica (**Alumno** o **Profesor**), ahorrándonos tener que escribir varias veces la parte de código que obtiene los atributos genéricos de la clase **Persona**.

1. **Clase** Persona**.**

Indicamos que se va a implementar la interfaz **Imprimible**:
public abstract class Persona **implements Imprimible** {
...

Definimos el método **devolverContenidoArrayList** a la manera de cómo debe ser implementado para la clase **Persona**. Podría quedar, por ejemplo, así:

```
public ArrayList devolverContenidoArrayList (){
        ArrayList contenido= new ArrayList ();
        //SimpleDateFormat formatoFecha = new
SimpleDateFormat("dd/MM/yyyy");
        //String stringFecha= formatoFecha.format(this.fechaNacim.getTime());
```

```
        contenido.add(this.nombre);
        contenido.add (this.apellidos);
        contenido.add(stringFecha);

        return contenido;
    }
```

Y por último el método **devolverContenidoString**:
public String devolverContenidoString () { ... }

2. **Clase** Alumno**.**

Esta clase hereda de la clase **Persona**, de manera que heredará los tres métodos anteriores. Tan solo habrá que redefinirlos para que, aprovechando el código ya escrito en la **superclase**, se añada la funcionalidad específica que aporta esta **subclase**.
public class Alumno extends Persona {
...
Como puedes observar no ha sido necesario incluir el **implements Imprimible**, pues el **extends Persona** lo lleva implícito dado que **Persona** ya implementaba ese **interfaz**. Lo que haremos entonces será llamar al método que estamos redefiniendo utilizando la referencia a la **superclase super**.

El método **devolverContenidoArrayList** podría quedar, por ejemplo, así:

```
public ArrayList devolverContenidoArrayList () {

        // Llamada al método de la superclase
ArrayList contenido= super.devolverContenidoArrayList ();

            // Añadimos los atributos específicos
        contenido.add (this.grupo);
        contenido.add (this.notaMedia);

            // Devolvemos el ArrayList relleno
        return contenido;
    }
```

3. **Clase** Profesor**.**

En este caso habría que proceder exactamente de la misma manera que con la clase **Alumno**: redefiniendo los métodos de la interfaz **Imprimible** para añadir la funcionalidad específica que aporta esta **subclase**.

6. Polimorfismo

Polimorfismo en tiempo de compilación (sobrecarga)

Esto sucede si una clase tiene métodos sobrecargados, lo cual hace que la misma llamada a un método actúe de una forma u otra dependiendo de los argumentos que reciba dicho método.

Polimorfismo en tiempo de ejecución (ligadura dinámica)

Esto sucede si una clase sobrescribe algún método de la clase que hereda; esto hará que una misma llamada a un método conlleve la ejecución de métodos diferentes, dependiendo del objeto que llame a dicho método.

EJEMPLO: En EjemploClaseHerenciaPersona

La clase Persona y la clase Alumno (que hereda de Persona). Si dentro de la clase Alumno hemos sobrescrito métodos de la clase Persona, cuando hagamos, por ejemplo, estas instanciaciones de objetos:

Persona per1= new Persona();

Persona per2= new Alumno();

Al hacer estas llamadas al mismo método:

per1.mostrar();

per2.mostrar();

El método que se va a ejecutar es diferente. Esto se debe a que se ejecuta el método asociado a la clase que se indicó en la instanciación y no el método asociado a la clase que se indicó en la declaración del objeto. Esto es lo que se llama **ligadura dinámica** o polimorfismo en tiempo de ejecución.

Para que exista polimorfismo en tiempo de ejecución se tienen que dar 3 condiciones:

- Que haya herencia.
- Que haya sobrescritura de métodos.
- Definir un objeto con una superclase e instanciarlo con una subclase:

  Persona per1=new Alumno();

Ejercicio 10: Repasar EjemploClaseHerenciaPersona o EjemploPolimorfismoPersona

Haz un pequeño programa en Java en el que se declare una variable de tipo **Persona**, se pidan algunos datos sobre esa persona (**nombre**, **apellidos** y si es **alumno** o si es **profesor**), y se muestren nuevamente esos datos en pantalla, teniendo en cuenta que esa variable no puede ser instanciada como un objeto de tipo **Persona** (es una **clase abstracta**) y que tendrás que instanciarla como **Alumno** o como **Profesor**. Recuerda que para poder recuperar sus datos necesitarás hacer uso de la **ligadura dinámica** y que tan solo deberías acceder a métodos que sean de la **superclase**.

Si tuviéramos diferentes variables referencia a objetos de las clases **Alumno** y **Profesor** tendrías algo así:

```
Alumno obj1;
Profesor obj2:
...
// Si se dan ciertas condiciones el objeto será de tipo Alumno y lo tendrás en obj1
System.out.printf ("Nombre: %s\n", obj1.getNombre());
// Si se dan otras condiciones el objeto será de tipo Profesor y lo tendrás en obj2
System.out.printf ("Nombre: %s\n", obj2.getNombre());
```

Pero si pudieras tratar de una manera más genérica la situación, podrías intentar algo así:

```
Persona obj;
// Si se dan ciertas condiciones el objeto será de tipo Alumno y por tanto lo instanciarás como tal
obj = new Alumno (<parámetros>);
// Si se dan otras condiciones el objeto será de tipo Profesor y por tanto lo instanciarás como tal
obj = new Profesor (<parámetros>);
```

De esta manera la variable **obj** podría contener una referencia a un objeto de la **superclase Persona** de **subclase Alumno** o bien de **subclase Profesor** (**polimorfismo**).
Esto significa que independientemente del tipo de **subclase** que sea (**Alumno** o **Profesor**), podrás invocar a métodos de la **superclase Persona** y durante la ejecución se resolverán como métodos de alguna de sus **subclases**:

```
//En tiempo de compilación no se sabrá de qué subclase de Persona será obj.
```

```
//Habrá que esperar la ejecución para que el entorno lo sepa e invoque al método adecuado.
System.out.printf ("Contenido del objeto usuario: %s\n", stringContenidoUsuario);
```

Por último recuerda que debes de proporcionar **constructores** a las **subclases Alumno** y **Profesor** que sean "compatibles" con algunos de los **constructores** de la **superclase Persona**, pues al llamar a un **constructor** de una **subclase**, su formato debe coincidir con el de algún **constructor** de la **superclase** (como debe suceder en general con cualquier método que sea invocado utilizando la **ligadura dinámica**).