UNIDAD 8: UTILIZACIÓN AVANZADA DE CLASES

Profesor: José Ramón Simó Martínez

Contenido

1. Introducción

En la anterior unidad introducimos los elementos que componen la **Programación Orientada a Objetos** (POO). Sin embargo, dejamos para la presente los que marcan la importancia de la POO: la **Herencia** y el **Polimorfismo**.

En esta unidad estudiaremos todos los aspectos de implementación de la herencia y el polimorfismo en la POO en el lenguaje Java. Empezaremos con una aproximación a las relaciones que se pueden establecer entre las clases. A continuación, profundizaremos en aquellas que establecen relaciones jerárquicas con la herencia simple, el uso de clases abstractas y finales. También introduciremos el concepto de interfaz como solución a la herencia múltiple en Java. Continuaremos con la técnica del polimorfismo y su importancia en la POO. Finalizaremos presentando la jerarquía de la API de Java y el uso de alguna de sus clases e interfaces más comunes.

Al terminar esta unidad deberás ser capaz de:

- Escribir programas estableciendo relaciones de asociación entre clases.
- Escribir programas estableciendo relaciones de herencia entre clases.
- Comprender y aplicar el concepto de herencia y polimorfismo.
- Crear clases abstractas e interfaces y conocer las ventajas de su uso.
- Sobrecargar y sobrescribir métodos de la superclase.
- Conocer la jerarquía de la API de Java.
- Comparar y ordenar objetos de nuestra propia clase.
- Desarrollar programas en Java aplicando técnicas avanzadas del paradigma orientado a objetos.

2. Relaciones entre clases

Las relaciones entre clases son cruciales en la programación orientada a objetos. Así como los conceptos como clases y objetos en la programación orientada a objetos se crean para modelar entidades del mundo real, las relaciones entre clases en la programación orientada a objetos se crean para modelar las relaciones entre entidades del mundo real que representan estas clases.

En nuestro primer contacto con la POO hemos comprendido la importancia del concepto de clase para modelar los objetos (y conceptos) del mundo real. Sin embargo, los objetos no están aislados unos de otros, sino que mantienen, de una forma u otra, relaciones entre ellos.

En Java se pueden modelar principalmente dos tipos de relaciones entre clases:

- **Asociación**: una clase contiene o usa objetos de otra clase. Se conoce como relación **HAS-A** (en inglés). Se conocen dos tipos de asociación:
    - **Agregación**: una clase puede existir independientemente de la otra. Se conoce como relaciones débiles.
    - **Composición**: una la existencia de una clase depende de la existencia de la otra. Se conoce como relaciones fuertes.
- **Herencia**: una clase es una subcategoría de otra clase. Se conoce como relación **IS-A** (en inglés)

En este apartado estudiaremos cómo se representan en Java las relaciones de asociación y en otros apartados entraremos en profundidad con la relación de herencia entre clases.

> **Nota**
>
> En el módulo de Entornos de Desarrollo (ED) se estudia con más detalle los conceptos teórico-prácticos del paradigma orientado a objeto, como por ejemplo los diagramas UML. En esta unidad se dará por entendido que el estudiante ha adquirido dichos conocimientos del módulo de ED.

2.1. Asociación con Java

La relación más simple es la de asociación. En el caso de que una clase haga uso o contenga a otra decimos que tiene una relación de asociación.

En Java podemos representar este tipo de relación según sea:

- **Unidireccional:** una clase usa o contiene a otra, pero no a la inversa. También se dice que una clase puede ver a otra.
- **Bidireccional:** ambas clases hacen referencia una a otra. Es decir, ambas se ven.

En el siguiente ejemplo podemos ver tanto la representación UML como el código correspondiente en Java de una relación unidireccional:


```
// Fichero B.java
public class B {
	private int atributoB;

	public B(){
		this.atributoB = 0;
	}
}
// Fichero A.java
public class A {
	private int atributoA;
	private B b1; // La clase A tiene una referencia a la clase B.

	public A(){
		this.atributoA = 0;
	}
}
```

Podemos ver en el código que para representar que la clase A puede ver a la clase B, añadimos a la clase A un atributo de tipo B.

En el anterior ejemplo la cardinalidad la relación es uno. Si quisiéramos representar una cardinalidad de uno a muchos simplemente declararíamos una lista de objetos de la clase relacionada:


```
// Fichero B.java
public class B {
	private int atributoB;

	public B(){
		this.atributoB = 0;
	}
}

// Fichero A.java
public class A {
	private int atributoA;

	// La clase A tiene uno o más objetos de B
	private ArrayList<B> b1;

	public B(){
		this.atributoA = 0;
	}
}
```

En la representación bidireccional debemos añadir una referencia del objeto referenciado a cada clase:

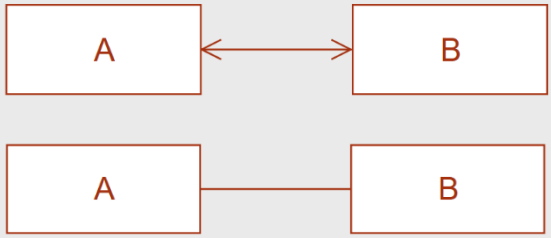
```
// Fichero B.java
public class B {
	private int atributoB;

	// Referencia a la clase A
	private A a1;

	public B(){
		this.atributoB = 0;
	}
}
// Fichero A.java
public class A {
	private int atributoA;

	// Referencia a la clase B
	private B b1;

	public A(){
		this.atributoA = 0;
	}
}
```



2.2. Agregación con Java

Una agregación es una asociación (pero no viceversa) entre dos clases de manera que una clase contiene uno o más elementos de la otra con la que está relacionada. Sin embargo, en Java la implementación de una agregación es la misma que una asociación.

Para la siguiente representación en UML se aplica el mismo ejemplo de código Java que hemos visto en la asociación unidireccional:

2.3. Composición con Java

Una composición es una asociación (pero no viceversa) entre dos clases de manera que una clase contiene un o más elementos de la otra con la que está relacionada. Se considera que se establece una relación de tipo fuerte, ya que la existencia de un objeto de una clase depende de la existencia del objeto de la otra clase que lo contiene.

En Java podemos representar la composición del modelo UML como se indica en el siguiente ejemplo:


```
// Fichero B.java
public class B {
	private int atributoB;

	public B(){
		this.atributoB = 0;
	}
}

// Fichero A.java
public class A {
	private int atributoA;

	private B b1; // Referencia a la clase B

	public B(){
		this.atributoA = 0;
		// Se instancia el objeto de B en el constructor de la clase A
		this.b1 = new B();
	}
}
```

Para reflejar una composición en Java tenemos que indicar que el objeto de la clase a la que se hace referencia se instancia en el constructor. En el ejemplo, cuando instanciemos la clase A a su vez se instanciará un objeto de la clase B. De esta forma cuando un objeto de la clase A deje de existir a su vez dejará de existir el objeto de la clase B con el que estaba relacionado.

En resumen:

- Tanto la agregación como la composición son asociaciones, pero no a la inversa.
- Tanto la agregación como la composición son unidireccionales, mientras que la asociación puede ser bidireccional.
- La implementación en Java es la misma para representar tanto una asociación como una agregación. La diferencia es a nivel lógico y de modelado.
- Una composición es una agregación, pero no a la inversa.

3. Herencia

Definición: La herencia en la POO es un mecanismo que permite adquirir las características de otras clases, esto es, sus atributos y métodos. En consecuencia, podemos establecer relaciones jerárquicas entre las clases de nuestro proyecto.

Esto supone muchas ventajas para nuestro proyecto software, entre las cuales están las siguientes:

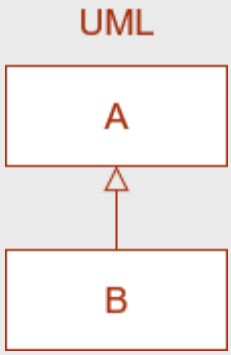
- Reutilización del código.
- Extender los requisitos funcionales de manera relativamente sencilla.
- Reducir los costes de desarrollo y mantenimiento.
- Facilitar las pruebas y la documentación.
- Seguridad de los datos.

A continuación, vamos a estudiar cómo el lenguaje Java permite implementar este mecanismo.

3.1. Herencia simple entre clases

En Java se utiliza la palabra reservada *extends* para indicar que una clase hereda de otra:

```
// Fichero A.java
public class A {
      private int atributoA;
      public A(){}
      public void metodoA() {}
}
// Fichero B.java
public class B extends A {
      private int atributoB;
      public B(){}
      public void metodoB() {}
}
```



En el anterior ejemplo decimos que la clase B extiende a la clase A. Dicho de otra forma, la clase B **hereda** los atributos y métodos de la clase A. No obstante, cabe tener en cuenta los modificadores de acceso que permiten mantener el encapsulamiento de las clases tal y como estudiamos en la unidad anterior. En el ejemplo anterior la clase B hereda los métodos públicos de A, pero no sus atributos ya que estos son privados.

Recordemos la tabla resumen sobre los modificadores de acceso que vimos en la unidad anterior:

Modificador	Clase	Paquete	Subclase	Otros
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
(default)	Sí	Sí	No	No
private	Sí	No	No	No

Como podemos ver en esta tabla, una solución al ejemplo anterior para que clase B pueda heredar de la clase A sería declarar los atributos de B como *public*, aunque como ya dijimos esto no es para nada una buena práctica. Por tanto, nos queda utilizar el modificador *protected*:

```
// Fichero A.java
public class A {
	protected int atributoA;

	public A(){}

	public void metodoA() {}
}
```

Ahora la clase B sí que podrá heredar el atributoA de la clase A.

De todas formas, no es necesario que una clase herede todos los componentes de otra. Podemos elegir, por tanto, qué atributos y métodos hereda (o no) una clase de otra indicando los modificadores de acceso adecuados:

```
// Fichero A.java
public class A {
	protected int atributoA; // se hereda
	private String otroAtributoA; // no se hereda

	public A(){}
	public void metodoA() {} // se hereda
	private void otroMetodoA() {} // no se hereda
}
```

Vamos a tener en cuenta las siguientes consideraciones con el uso de los modificadores de acceso en general y en la herencia en particular:

- En general:
  - Declarar los atributos como *private* y los métodos como *public*.
  - Crear los *getters* y *setters* necesarios (ambos siempre *public*) para acceder a los atributos de la clase.

- Si queremos que un atributo sea de solo de lectura, crear solo su *getter*.
- Si queremos que un atributo sea de solo de escritura, crear solo su *setter*.
- Lo métodos que no pertenezcan a la API de nuestra clase, declararlos como *private*.
- Al declarar métodos *public* o *protected* nos comprometemos a mantenerlos durante el tiempo de mantenimiento de nuestra clase (o librería).

- En la herencia:
  - El modificador *protected* usarlo en caso de que queramos publicar nuestro método solo para las clases que heredan y no sea usado como parte de la API de nuestra librería.
  - El modificador *protected* limitarlo para los métodos y no para atributos, que deberían ser aconsejablemente *private*; accederemos a los atributos de la clase heredada a través de sus *getters* y *setters*.

3.2. Llamada a constructores en la herencia

Cuando instanciamos un objeto de una subclase a través de su constructor, Java primero llama al constructor de su superclase:

```
// Fichero UnaSuperClase.java
public class UnaSuperClase {
	// Constructor por defecto de la superclase
	public UnaSuperClase() {
		System.out.println("Constructor de la super clase…");
	}
}
// Fichero UnaSubClase.java
public class UnaSubClase extends UnaSuperClase{
	// Constructor por defecto de la subclase
	public UnaSubClase() {
		System.out.println("Constructor de la subclase…");
	}
}
// Fichero Test.java
public class Test {
	public static void main(String[] args) {
		UnaSubClase subclase = new UnaSubClase();
	}
}
```

Salida por pantalla:

```
Constructor de la super clase…
Constructor de la subclase…
```

3.3. Acceso a métodos y constructores de la superclase: uso de *super*

Una subclase puede acceder a los métodos de su superclase a través de la palabra reservada **super**.

```
// Fichero UnaSuperClase.java
public class UnaSuperClase {

	// Método de la superclase
	public void metodoSuperClase() {
		System.out.println("Método de la superclase…");
	}
}

// Fichero UnaSubClase.java
public class UnaSubClase extends UnaSuperClase{

	// Método de la subclase
	public metodoSuclase() {
		super.metodoSuperClase(); // llamada al método de la superclase
		System.out.println("Método de la subclase…");
	}
}

// Fichero Test.java
public class Test {
	public static void main(String[] args) {
		UnaSubClase subclase = new UnaSubClase();

		suclase.metodoSubclase();
	}
}
```

Salida por pantalla:

```
Método de la superclase…
Método de la subclase…
```

Otro uso destacado de la palabra reservada **super** es para llamar a **constructores con argumentos** de la superclase:

```
// Fichero UnaSuperClase.java
public class UnaSuperClase {
	private String s;
	private int a;

	// Constructor con parámetros de la superclase
	public UnaSuperClase(String s, int a) {
		this.s = s;
		this.a = a;
	}
}
```

```
// Fichero UnaSubClase.java
public class UnaSubClase extends UnaSuperClase{
	private int b;

	// Constructor por defecto de la subclase
	public UnaSubClase(String s, int a, int b) {
		super(s, a); // Llamada al constructor de la superclase
		this.b = b;
	}
}
```

Nota

En caso querer acceder al constructor de la superclase, super(…) debe ir siempre como primera instrucción dentro del constructor.

```
// super debería ir siempre como primera instrucción
public UnaSubClase(String s, int a, int b) {
	this.b = b;
	super(s, a);  // Error
}
```

3.4. Sobrecarga de métodos de la superclase

En la unidad anterior ya estudiamos la sobrecarga de métodos de una clase. Recordemos que una clase puede redefinir un método de manteniendo el mismo nombre, pero distinta lista de argumentos (y el tipo de retorno da igual si cambia o no).

En la **sobrecarga de métodos** de la superclase la idea es la misma: una subclase puede redefinir un método o métodos de la superclase manteniendo el mismo nombre, pero distinta lista de argumentos (y el tipo de retorno da igual si cambia o no).

Veamos un ejemplo:

```
// Fichero UnaSuperClase.java
public class UnaSuperClase {
	public void saludar(String nombre){
		System.out.println("Hola" + nombre);
	}
}

// Fichero UnaSubClase.java
public class UnaSubClase extends UnaSuperClase{

	// Sobrecarga el método de la clase padre
	public void saludar(String nombre, int nVeces){
		for(int i = 0; i < nVeces; i++) {
			System.out.println("Hola" + nombre);
```

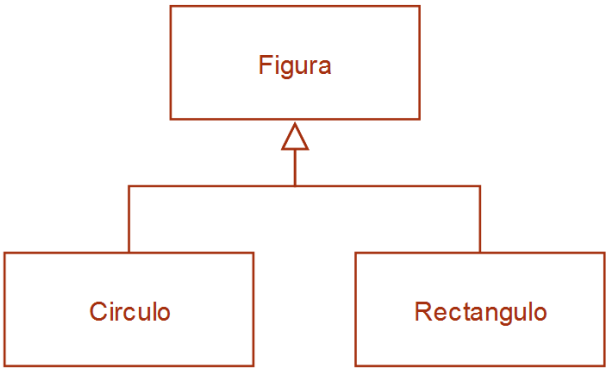
```
            }
      }
}

// Fichero Test.java
public class Test {
      public static void main(String[] args) {
            UnaSubClase subclase = new UnaSubClase();

            subclase.saludar("Ana"); // usa saludar de la clase padre
            subclase.saludar("Pepe", 3); // usa saludar de su propia clase
      }
}
```

3.5. Clases y métodos abstractos

Veamos el siguiente ejemplo de jerarquía de clases en UML:



Ahora ya sabemos que en este ejemplo se representa que tanto la clase Circulo como la clase Rectangulo heredan de una clase llamada Figura. También conocemos los mecanismos para representar dicho ejemplo en Java.

En este punto debemos hacernos la siguiente pregunta, ¿tiene sentido que podamos instanciar un objeto de la clase Figura? En principio, no parece que tenga mucho sentido ya que no sabríamos, por ejemplo, cómo representar gráficamente una figura en general. Tal y como ocurre en la vida real diremos que una figura general es un **objeto abstracto**.

> **Nota**
>
> En el ejemplo anterior decimos que Figura es una clase abstracta, mientras que Circulo y Rectangulo son clases concretas.

Java tiene un mecanismo para crear clases abstractas, es decir, clases de las cuales no podremos instanciar objetos. Para ello, usaremos la palabra reservada *abstract*.

Una **clase abstracta** la implementaremos de la siguiente manera:

```
// Fichero Figura.java
public abstract class Figura {
	// Atributos

	// Constructores

	// Métodos (concretos o abstractos)
}
```

Una clase abstracta puede contener dos tipos de métodos:

- **Concretos**: los que ya conocemos de las clases concretas y que necesitan contener la implementación de lo que hacen.
- **Abstractos**: métodos propios de una clase abstracta. Estos métodos no deben contener ninguna implementación, solo la definición del método. Su implementación, por tanto, deberá ser a cargo de las clases que heredan los métodos de la clase abstracta. Al igual que la clase abstracta, estos métodos también utilizan la palabra clave *abstract*.

Un método abstracto se declara de la siguiente manera:

```
modificador abstract tipoDeVariable metodo1(parámetros);
```

Un ejemplo de implementación de la clase Figura sería el siguiente:

```
// Fichero Figura.java
public abstract class Figura {
	private String nombre;
	// Constructor
	public Figura() {
		this.nombre = "Figura desconocida";
	}

	// Métodos concretos
	public Figura(String nombre) {
		this.nombre = nombre;

	public String getNombre() {
		return this.nombre;
	}

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

	// Métodos abstractos
	public abstract double getArea();
	public abstract double getPerimetro();
}
```

En el anterior ejemplo vemos como se implementan los métodos concretos, pero no los abstractos. Estos se deberán implementar en las clases que heredan de la clase Figura.

Por ejemplo, veamos la cómo se implementa la clase Cuadrado que hereda de la clase Figura:

```
// Fichero Figura.java
public class Rectangulo extends Figura {
	private double ancho;
	private double alto;

	// Constructor
	public Rectangulo(double ancho, double alto) {
		super("Rectángulo");
		this.ancho = ancho;
		this.alto = alto;
	}

	@Override
	public double getArea() {
		return this.ancho * this.alto;
	}

	@Override
	public double getPerimetro() {
		return 2.0 * (this.ancho + this.alto);
	}
}
```

Nota

Cuando la subclase implementa los métodos heredados de la clase abstracta se dice que los está sobrescribiendo. El concepto de sobrescritura de métodos es de vital importancia para el polimorfismo, concepto que trataremos en siguientes apartados.

@Override es una etiqueta informativa para el compilador de Java. En este caso, indica indica al compilador que estamos sobrescribiendo los métodos de la interfaz. Esto nos ayudará a evitar errores en la denominación de los métodos sobrescritos.

En el ejemplo de la clase Rectangulo podemos observar que implementamos los métodos abstractos que heredamos de la clase Figura. Atención, en este caso ya son métodos concretos y por tanto no debemos usar la palabra clave *abstract*.

Ahora, implementamos la clase Circulo:

```
// Fichero Circulo.java
public class Circulo extends Figura {
	private double radio;

	// Constructor
	public Circulo(double radio) {
```

```
            super("Círculo");
            this.radio
      }

      @Override
      public double getArea() {
            return Math.PI * radio * radio;
      }

      @Override
      public double getPerimetro() {
            return 2.0 * Math.PI * radio;
      }
}
```

Veamos cómo probar estas clases que hemos creado:

```
// Fichero Test.java
public class Test {
      public static void main(String[] args) {
            Rectangulo r = new Rectangulo(10.0, 2.0);
            Circulo c = new Circulo(2.0);

            double areaCuadrado = r.getArea();
            double areaCirculo = c.getArea();
            System.out.println("Área cuadrado: " + areaCuadrado);
            System.out.println("Área círculo: " + areaCirculo);
}
```

3.6. Clases y métodos finales: uso de *final*

Hay casos en nuestro diseño por el que deseamos que nuestra clase no se pueda heredar. Para ello Java proporciona la palabra clave **final**. Por tanto, podemos restringir el uso de la herencia utilizado las conocidas como *clases finales*.

Su sintaxis es la siguiente:

```
public final class UnaClase {
}
```

Si intentamos hacer esto nos dará error:

```
public class OtraClase extends UnaClase {
}
```

También podemos declarar dentro de una superclase no final un método final. El objetivo es que dicho método no se pueda sobrescribir en las subclases que hereden de dicha superclase. La sintaxis es la siguiente:

```
public final void unMetodoFinal() { }
```

4. Interfaces

Hemos visto cómo la **herencia** permite definir especializaciones (o extensiones) de una clase base que ya existe sin tener que repetir el código de ésta. Este mecanismo da la oportunidad de que la nueva clase especializada (o extendida) disponga de toda la interfaz que tiene su clase base.

También hemos estudiado cómo los **métodos abstractos** permiten establecer una interfaz para marcar las líneas generales de un comportamiento común de superclase que deberían compartir de todas las subclases.

Si llevamos al límite esta idea de interfaz, podrías llegar a tener una clase abstracta donde todos sus métodos fueran abstractos. De este modo estarías dando únicamente el marco de comportamiento, sin ningún método implementado, de las posibles subclases que heredarán de esa clase abstracta.

La idea de una interfaz (o *interface*) es precisamente ésa: disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación (no necesariamente jerárquica).

4.1. Concepto de interfaz

Una interfaz en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes. Estos métodos sin implementar indican un comportamiento, un tipo de conducta, aunque no especifican cómo será ese comportamiento (implementación), pues eso dependerá de las características específicas de cada clase que decida implementar esa interfaz.

En resumen:

- Una interfaz se encarga de establecer qué comportamientos hay que tener (qué métodos), pero no dice nada de cómo deben llevarse a cabo esos comportamientos (implementación).
- En una interfaz solo se indica sólo la forma, no la implementación.
- En cierto modo podrías imaginar el concepto de interfaz como un guion que dice: "éste es el protocolo de comunicación que deben presentar todas las clases que implementen esta interfaz".
- Se proporciona una lista de métodos públicos y, si quieres dotar a tu clase de esa interfaz, tendrás que definir todos y cada uno de esos métodos públicos.
- Los nombre de las interfaces en Java terminan con sufijos del tipo "-able", "-or", "-ente" y cosas del estilo, que significan algo así como capacidad o habilidad para hacer o ser receptores de algo (configurable, serializable, modificable, clonable, ejecutable, etc).

4.2. Definición de interfaces en Java

La declaración de una interfaz en Java es similar a la declaración de una clase, aunque con algunas variaciones:

- Se utiliza la palabra reservada **interface** en lugar de **class**.

- Puede utilizarse el modificador *public*. Si incluye este modificador la interfaz debe tener el mismo nombre que el archivo *.java* en el que se encuentra (exactamente igual que sucedía con las clases). Si no se indica el modificador *public*, el acceso será por omisión o "de paquete" (como sucedía con las clases).
- Todos los miembros de la interfaz (atributos y métodos) son *public* de manera implícita. No es necesario indicar el modificador *public*, aunque puede hacerse.
- Todos los atributos son de tipo *final* y *public* (tampoco es necesario especificarlo), es decir, constantes y públicos. Hay que darles un valor inicial.
- Todos los métodos son abstractos también de manera implícita (tampoco hay que indicarlo). No tienen cuerpo, tan solo la cabecera.

Como puedes observar, una interfaz consiste esencialmente en una lista de…

- Atributos finales (constantes) y
- Métodos abstractos (sin implementar).

Su sintaxis, en Java, quedaría entonces:

```
[public] interface <NombreInterfaz> {
       [public] [final] <tipo1> <atributo1>= <valor1>;
       [public] [final] <tipo2> <atributo2>= <valor2>;
       ...
       [public] [abstract] <tipo_devuelto1> <nombreMetodo1> ([lista_parámetros]);
       [public] [abstract] <tipo_devuelto2> <nombreMetodo2> ([lista_parámetros]);
       ...
}
```

4.3. Clases abstractas vs interfaces

En este punto puedes pensar que la idea de clase abstracta e interfaz es la misma. Ciertamente podría ser así, pero veamos a continuación las similitudes y diferencias que existen entre estos dos conceptos:

SIMILITUDES	DIFERENCIAS
No pueden ser instanciadas.	Las Interfaces no pueden contener ninguna implementación.
No pueden ser selladas (*final*).	Las Interfaces no pueden declarar miembros no públicos.
	Las Interfaces no pueden extender clases.

4.4. Ejemplo de creación y uso de una interfaz

Vamos a escribir una interfaz que declare el comportamiento que debe tener todo objeto multimedia que pueda ser reproducido (discos, películas, etc):

```
// Fichero Reproducible.java
interface Reproducible {
      void reproducir();
      void parar();
      void pausar();
}
```

Para utilizar esta interfaz crearemos, por ejemplo, una clase ReproductorMusica:

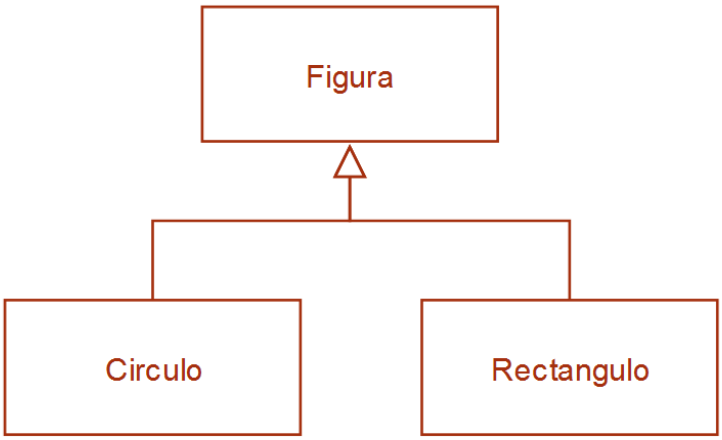
```
public class ReproductorMusica implements Reproducible {
    private boolean estaReproduciendo;
    @Override
    public void reproducir() {
        if (!estaReproduciendo) {
            System.out.println("Reproduciendo la música…");
            estaReproduciendo = true;
        }
    }
    @Override
    public void pausar() {
        if (estaReproduciendo) {
            System.out.println("Pausando la música…");
            estaReproduciendo = false;
        }
    }
    @Override
    public void parar() {
        if (estaReproduciendo) {
            System.out.println("Parando la música…");
            estaReproduciendo = false;
        }
    }
}
```

Nota

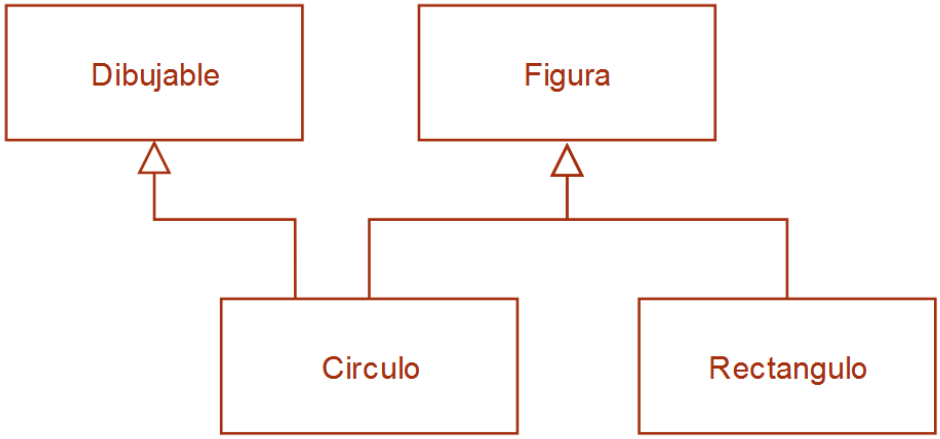
Atención al uso de la etiqueta **@Override** para indicar al compilador que estamos sobrescribiendo los métodos de la interfaz. Esto nos ayudará a evitar errores en la denominación de los métodos sobrescritos.

4.5. Herencia múltiple

Hasta ahora hemos visto los casos de herencia simple. Recordemos el ejemplo de las figuras:



Se puede dar el caso de que en nuestro programa tengamos figuras que se puedan dibujar y otras no. En este ejemplo, queremos indicar que los objetos de la clase Circulo se pueden dibujar mientras que de la clase Rectangulo no:



Este diagrama refleja lo que se conoce como **herencia múltiple**. El concepto de herencia múltiple existe a nivel conceptual, pero no a nivel de implementación en Java de forma directa: no podemos hacer que una clase herede de dos clases diferentes directamente a través de la palabra reservada *extends*:

```
public class Circulo extends Figura, Dibujable {…}
```

Esto se debe a razones de conflictos de nombre de métodos o atributos derivados. Si la clase Figura y Dibujable tienen un método llamado mostrar(), no hay manera de indicarle a la clase Figura si utiliza el método mostrar() de Dibujable o de Figura, ya que lo ha heredado de ambas.

Para solucionar este diseño Java hace uso de las interfaces que estamos tratando en este apartado.

Por tanto, siguiendo el anterior ejemplo, Dibujable dejará de ser una clase y pasará a ser una interfaz:

```
// Dibujable.java
interfaz Dibujable {
      void dibujar();
}

public class Circulo extends Figura implements Dibujable {
      @Override
      public double getArea() {
            return this.ancho * this.alto;
      }

      @Override
      public double getPerimetro() {
            return 2.0 * (this.ancho + this.alto);
      }

      @Override
      public void dibujar() {
            System.out.println("Dibujando círculo…");
      }
}

public class Rectangulo extends Figura {
      @Override
      public double getArea() {
            return this.ancho * this.alto;
      }

      @Override
      public double getPerimetro() {
            return 2.0 * (this.ancho + this.alto);
      }
}
```

Consideraciones para tener en cuenta sobre la herencia múltiple:

- Una clase solo puede heredar como máxima de una clase.
- Una clase puede implementar múltiples interfaces. Por ejemplo:

```
public class A implements interfaceB, interfaceC {…}
```

- Una interfaz puede heredar de varias interfaces y permitir diseños más jerárquicos. Por ejemplo:

```
interface A {…}

interface B {…}

interface C extends A, B {…} // Hereda los métodos de A y B
```

4.6. Métodos *default* y *static*

A partir de Java 8 se pueden definir dos tipos de métodos dentro de la interfaz: **default** y **static**.

4.6.1. Método default

Los métodos definidos como default se implementan en la misma interfaz y, por tanto, todas las clases que implementen la interfaz heredarán este método y su comportamiento. No obstante, también podemos sobrescribir el método para afinar el comportamiento.

La sintaxis es la siguiente:

```
public interface MiInterfaz {
	// Métodos regulares de la interfaz

	// Métodos default
	default void metodoPorDefecto() {
		// Aquí implementación del método
	}
}
```

El objetivo de los métodos default es evitar que cuando una interfaz añada métodos nuevos, haya que actualizar todas las clases que implementen dicha interfaz implementando dichos métodos. Veamos un ejemplo.

Tenemos una interfaz definida de la siguiente manera:

```
public interface Dibujable {
	public void dibujar2D();
}
```

Todos las clases que incluyan la interfaz Dibujable deben implementar el método dibujar2D().

```
public class Rectangulo implements Dibujable {
	public void dibujar2D() { // Implementación…};
}

public class Ciruclo implements Dibujable {
	public void dibujar2D() { // Implementación…};
}

public class Triangulo implements Dibujable {
	public void dibujar2D() { // Implementación…};
}

// … y muchas más clases
```

Después de un tiempo queremos actualizar dicha interfaz de manera que también se pueda dibujar en 3D:

```
public interface Dibujable {
	void dibujar2D();
	void dibujar3D();
}
```

Como consecuencia, todas las clases que hayan implementado dicha interfaz quedan inutilizadas ya que necesitan actualizarse al nuevo método de la interfaz. Para solucionar este problema, en parte, sería declarar el nuevo método dibujar3D() como default e implementarle, por ejemplo, un algoritmo básico para dibujar en 3D:

```
public interface Dibujable {
	void dibujar2D();
	default void dibujar3D() {
		// Aquí implementar algoritmo básico para dibujar en 3D…
	}
}
```

4.6.2. Método static

Por otro lado, los métodos **static** se definen para indicar que el método es propio de la interfaz y no pertenecerá a la API de las clases que implementan dicha interfaz. Para acceder a este método deberemos indicar el nombre de la interfaz y el nombre del método.

La sintaxis es la siguiente:

```
public interface MiInterfaz {
	// Métodos regulares de la interfaz

	// Métodos default
	static void metodoPorDefecto() {
		// Aquí implementación del método
	}
}
```

Utilizaríamos dicho método en las clases que implementan la interfaz de esta manera:

```
MiInterfaz.metodoPorDefecto();
```

La idea detrás de los métodos **static** en una interfaz es la de proporcionar un mecanismo simple que permita agrupar en un mismo lugar métodos relacionados sin tener que crear un objeto nuevo para ser usados.

> **Nota**
>
> La misma idea se puede aplicar con el uso de clases abstractas. No obstante, se debe tener en cuenta las ventajas y desventajas de usar clases e interfaces.

5. Polimorfismo

5.1. Concepto de polimorfismo

El diccionario define polimorfismo en el ámbito de la biología como:

"Propiedad de las especies de seres vivos cuyos individuos pueden presentar diferentes formas o aspectos…"

Este principio se puede aplicar al ámbito de la POO y en lenguajes de programación como Java. La idea parte del concepto de herencia donde una superclase se puede comportar o tomar la forma de las subclases que heredan de ella.

Recordemos que hemos hablado del concepto de sobrescritura de métodos (y el uso de la etiqueta @Override) en anteriores apartados:

- Las subclases opcionalmente sobrescriben métodos de superclases concretas
- Las subclases obligatoriamente sobrescriben los métodos abstractos de la superclase abstracta.

En este caso, hablamos del conocido técnicamente como **polimorfismo de inclusión o herencia**.

> **Nota**
>
> Existen otros tipos de polimorfismos que se pueden aplicar en Java. No obstante, en esta unidad nos centramos en el polimorfismo aplicado a la herencia.

5.2. Ejemplo de polimorfismo

A partir del ejemplo del apartado 3.3 de la jerarquía de clases sobre figuras geométricas, veamos cómo funciona el polimorfismo en una clase de prueba:

```
public class Test {
	public static void main(String[] args) {
		Figura f1 = new Rectangulo(2.0, 5.0); // polimorfismo de herencia
		Figura f2 = new Circulo(2.0);

		double area1 = f1.getArea(); // área del rectángulo!!
		double area2 = f2.getArea(); // ahora área del círculo!!
	}
}
```

Vemos que podemos declarar un objeto Figura asignándole una instancia de un objeto Rectangulo o Circulo. Esto es puro polimorfismo: la clase Figura puede tomar la forma de un Rectangulo o Circulo.

Otra manera de aprovechar el polimorfismo es a través de los parámetros pasados a una función:

```
public class Test {
	public void mostrarArea(Figura f) {
		System.out.println("Área: " + f.getArea());
```

```
            System.out.println("Perímetro: " + f.getPerímetro());
      }

      public static void main(String[] args) {
            Rectangulo r = new Rectangulo(2.0, 5.0);
            Ciruclo c = new Circulo(2.0);
            mostrarInformacion(r);
            mostrarInformacion(c);
      }
}
```

En este caso pasamos como parámetro un objeto de tipo Rectangulo o Circulo a una función que recibe objetos de tipo Figura como parámetro.

Continuamos con más ejemplos para desplegar la potencia del polimorfismo, esta vez haremos uso de los arrays:

```
public class Test {
      public static void main(String[] args) {
            ArrayList<Figura> figuras = new ArrayList<Figura>();

            Figura f1 = new Rectangulo(2.0, 5.0);
            Figura f2 = new Circulo(2.0);

            figuras.add(f1);
            figuras.add(f2);

            // Atención a este fragmento
            for(Figura f : figuras) {
                  System.out.println("Área: " + f.getArea());
                  System.out.println("Perímetro: " + f.getPerímetro());
            }
      }
}
```

En el ejemplo declaramos un ArrayList donde almacenaremos objetos de tipo Figura. Como Rectangulo y Circulo son Figura podemos añadirlos a este array. Luego, la gracia está en *f.getArea()* dentro del bucle: si sacamos un objeto de tipo Rectangulo llamará al getArea() de Rectangulo, y si el objetos es Circulo llamará al getArea() de la clase Circulo.

6. Jerarquía de la API de Java

En la jerarquía de Java existen: clases e interfaces. En el siguiente enlace se presenta el esquema de la jerarquía oficial de la API de Java 8:

https://docs.oracle.com/javase/8/docs/api/java/lang/package-tree.html

> **Nota**
>
> Aunque estemos usando el compilador JDK 17 es perfectamente válido para nuestro propósito educativo.

En este apartado vamos a estudiar el objecto ***Object*** y algunos de sus métodos de uso común. También veremos la interfaz ***Comparable<T>***, la cual nos resultará de bastante utilidad a la hora de comparar nuestras propias clases. Ambos elementos pertenecen al paquete *java.lang* (recuerda que este paquete se añade por defecto a nuestras aplicaciones).

6.1. La clase Object

En la jerarquía de la API de Java todas sus clases son realmente subclases, excepto una: la **clase Object**. Sin embargo, cuando definimos una clase en Java de manera implícita hereda de la clase Object (no hace falta poner un *extend*).

La clase Object está definida en el paquete java.lang, que a estas alturas del curso podrás adivinar que se importa automáticamente cada vez que escribimos un program. En otras palabras, las dos siguientes declaraciones son lo mismo:

```
public class Circulo {
}
public class Circulo extends Object {
}
```

Por otra parte, la clase Object incluye métodos que las subclases pueden usar, sobrecargar o sobrescribir. En el siguiente enlace se describen los métodos de la clase Object:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html

La mayoría de estos métodos se aplican en conceptos avanzados de programación en Java y que en principio no se tratarán en este curso. No obstante, entraremos en detalle con dos métodos que sí se utilizan habitualmente: *toString()* y *equals()*.

6.1.1. Sobrescribir el método toString()

Este método de la clase Object convierte un objeto en una cadena de texto (objeto String) que contiene información sobre dicho objeto. Veamos un ejemplo:

```
// Fichero Test.java
public class Test {
      public static void main(String[] args) {
            Rectangulo r = new Rectangulo(2.0, 5.0);

            System.out.println(r.toString());
}
```

Salida por pantalla:

```
Rectangulo@6d06d69c
```

Este código un valor que muestra el identificador único que Java asigna a un objeto, y podemos deducir que dicho valor no muestra información útil al usuario.

Si añadimos método *toString()* a nuestra clase estaremos sobrescribiendo el método de la clase Object; de esta manera, podemos adaptar la información que queremos mostrar sobre nuestro objeto en concreto:

```
// Fichero Rectangulo.java
public class Rectangulo extends Figura {
      // Añadimos el método toString
      @Override
      public String toString() {
          System.out.println("Area del rectángulo: " + getArea());
          System.out.println("Perímetro del rectangulo: " + getPerimetro());
      }
}
```

Ahora, la salida por pantalla de Test.java sería:

```
Área del rectángulo: 10.0
Perímetro del rectángulo: 14.0
```

Una de las ventajas de sobrescribir el método toString() es que es el método por defecto que utiliza Java para formatear la información de un objeto a String. En el ejemplo Test.java podemos obviar la referencia al método toString() y funcionaría de la misma manera:

```
System.out.println(r);
```

6.1.2. Sobrescribir el método equals()

La clase *Object* también contiene un método *equals()* con la siguiente cabecera:

```
public boolean equals(Object obj)
```

El método toma un parámetro de tipo *Object*, esto es, que podemos incluir cualquier tipo de objeto de Java o que hayamos creado nosotros. Este método ya lo hemos utilizado a lo largo del curso para la comparación de cadenas con la clase *String*. Por ejemplo:

```
String s1 = "cadena";
String s2 = "cadena";
String s3 = new String("cadena");
if (s1.equals(s2) {
    System.out.println("Son iguales");
```

El objetivo de *equals()* es comparar si los datos que contiene el objeto son iguales, y no tanto si ambos objetos son iguales a nivel de referencia en memoria.

> **Nota**
>
> Recordemos que en el caso de los String si comparamos con == no funcionaba correctamente, porque en el ejemplo s1 y s2 son el mismo objeto (s3 sería un objeto diferente). Por tanto, la clase String tiene sobrescrito el método equals() que compara las cadenas de texto de sus objetos (que es el contenido) y por eso funciona.

Vamos a aprender cómo sobrescribir el método *equals()* de la clase *Object* para poder comparar adecuadamente objetos de nuestra propia clase. Tomemos como ejemplo la clase *Rectangulo*:

```
public class Rectangulo extends Figura {
      // Añadimos el método equals()
      @Override
      public boolean equals(Object obj) {
            Rectangulo r = (Rectangulo) obj;

            return this.getArea() == r.getArea();
      }
}
```

Esta es la forma más simple de sobrescribir el método *equals()* en nuestra clase. En caso de utilizarlo para programas sencillos podría valer perfectamente. No obstante, cuando empecemos a utilizar técnicas de Java más avanzadas podremos encontrarnos con ciertos errores al usar esta versión.

Estas son las recomendaciones que dan los creadores de Java sobre la implementación más adecuada del método *equals()*:

- Determinar si el parámetro *Object* es el mismo objeto que el objeto que llama al método. Para ello hacemos la comparación *obj == this*, devolviendo true en ese caso.
- Devolver false el parámetro *Object* es *null*.
- Devolver false si el parámetro *Object* y el objeto que llama al método no son la misma clase.
- Hacer casting del parámetro *Object* al mismo tipo que el objeto que llama al método, solo si ambos son de la misma clase.

Veamos el ejemplo anterior ampliado para seguir estas recomendaciones:

```
public class Rectangulo extends Figura {
      // Atributos y métodos ya implementados en anteriores ejemplos
      // Añadimos el método equals()
      @Override
```

```
	public boolean equals(Object obj) {
		if(obj == this)
			return true;
		else
			if(obj == null)
				return false;
			else
				if(obj.getClass() != this.getClass())
					return = false;

		Rectangulo r = (Rectangulo) obj;

		return this.getArea() == r.getArea();
	}
}
```

Ahora hagamos una prueba:

```
// Fichero Test.java
public class Test {
	public static void main(String[] args) {
		Rectangulo r1 = new Rectangulo(2.0, 5.0);
		Rectangulo r2 = new Rectangulo(2.0, 5.0);
		Rectangulo r3 = new Rectangulo(10.0, 5.0);

		System.out.println(r1.equals(r2));
		System.out.println(r1.equals(r3));
	}
}
```

Salida por pantalla:

```
true
false
```

6.2. La interfaz *Comparable<T>*

Otra forma de poder comparar dos objetos es haciendo uso del método compareTo de la interfaz Comparable<T>. Esta interfaz se define de la siguiente manera:

```
public interface Comparable<T> {
	public int compareTo(T obj);
}
```

Nota

El *<T>* hace referencia al concepto de programación genérica, que básicamente indica que podemos sustitur la T por cualquier objeto sobre el que queramos implementar la comparación.

El método compareTo(), a diferencia del método equals(), devuelve un entero. Normalmente según su valor indicará:

- Devuelve 1: el objeto que llama al método es mayor que el objeto del parámetro.
- Devuelve -1: el objeto que llama al método es menor que el objeto del parámetro.
- Devuelve 0 (cero): ambos objetos son iguales.

Vamos un ejemplo implementando la interfaz en el ejemplo de la clase Rectangulo:

```
public class Rectangulo extends Figura implements Comparable<Rectangulo> {
	// Atributos y métodos ya implementados en anteriores ejemplos
	// Añadimos el método compareTo()
	@Override
	public int compareTo(Rectangulo r) {
		if(this.getArea() > r.getArea())
			return 1;
		else if(this.getArea() < r.getArea())
			return -1;
		else
			return 0;
	}
}
```

Ahora hagamos una prueba:

```
// Fichero Test.java
public class Test {
	public static void main(String[] args) {
		Rectangulo r1 = new Rectangulo(2.0, 5.0);
		Rectangulo r2 = new Rectangulo(2.0, 5.0);
		Rectangulo r3 = new Rectangulo(10.0, 5.0);

		if(r1.compareTo(r2) > 0)
			System.out.println("Es mayor");
		else if(r1.comparteTo(r2) < 0)
			System.out.println("Es menor");
		else
			System.out.println("Son iguales");
	}
}
```

Por otra parte, el uso de la interfaz *Comparable*<T> es de gran utilidad cuando queremos ordenar objetos dentro de una lista. En el caso de listas estáticas utilizaremos el método *sort()* de la clase *Arrays*:

```
// Fichero Test.java
public class Test {
	public static void main(String[] args) {
		Rectangulo rectangulos = new Rectangulo[3];

		rectangulos[0] = new Rectangulo(2.0, 5.0);
		rectangulos[1] = new Rectangulo(2.0, 5.0);
```

```
            rectangulos[2] = new Rectangulo(10.0, 5.0);

            Arrays.sort(rectangulos);
      }
}
```

En el caso de de listas dinámicas, utilizaremos el método el método *sort()* de la clase *Collections*:

```
// Fichero Test.java
public class Test {
      public static void main(String[] args) {
            ArrayList<Rectangulo> rectangulos = new ArrayList<Rectangulo>();

            rectangulos.add(new Rectangulo(2.0, 5.0));
            rectangulos.add(new Rectangulo(2.0, 5.0));
            rectangulos.add(new Rectangulo(10.0, 5.0));

            Collections.sort(rectangulos);
      }
}
```

En ambos casos, el resultado es la ordenación de la lista que hayamos pasado al método *sort()*.

Nota

Es fundamental que para que *Arrays.sort()* y *Collections.sort()* puedan ordenar los objetos, la clase del objeto a ordenar tenga implementado el método *compareTo()* de la interfaz *Comparable*.

Por último, debemos tener en cuenta la relación entre el método *compareTo()* y *equals()*:

- En caso de tener intención de comparar objetos de la clase, es recomendable implementar ambos métodos para que haya consistencia en la comparación de los objetos de dicha clase.

7. Terminología

agregación, api de java, asociación, clases abstractas, clases finales, comparación de objectos, composición, herencia, herencia múltiple, interfaces, interfaz Comparable, jerarquía, métodos abstractos, métodos finales, objeto object, interfaz comparable, polimorfismo, sobrecarga, sobrescritura, subclase, super, superclase, this.

8. Bibliografía

Libro “Java Programming 9th Edition” de Joyce Farrell

Apuntes de José Chamorro del CFGS DAW del IES Sant Vicent Ferrer (Algemesí).

https://docs.oracle.com/