FAMILIA PROFESIONAL:	**Informática y Comunicaciones**
CICLOS FORMATIVOS:	**Desarrollo de Aplicaciones Multiplataforma, Desarrollo de Aplicaciones Web**
MÓDULO:	**Programación**

UNIDAD 6:
RELACIONES ENTRE CLASES Y POLIMORFISMO

CONTENIDOS

AUTORES: **Fernando Rodríguez Alonso**
Sonia Pasamar Franco

ÍNDICE DE CONTENIDOS

1. ASOCIACIONES ENTRE CLASES

Una **asociación** es una relación estructural que especifica que los objetos de una clase están conectados con los objetos de otra clase. Esta asociación es binaria, pues representa una relación entre dos objetos distintos.

1.1. TIPOS DE ASOCIACIONES

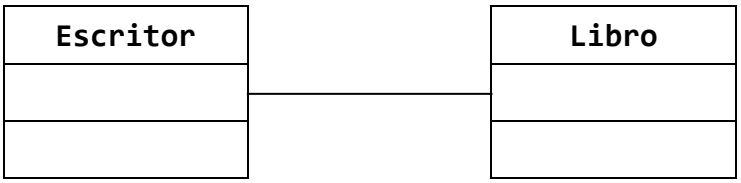
Según el **ciclo de vida de los objetos relacionados**, una asociación se puede clasificar en: asociación simple, asociación de agregación o asociación de composición.

1.1.1. Asociación Simple

En una **asociación simple**, los dos objetos relacionados entre sí existen de forma independiente.

Por tanto, la creación o destrucción de uno de los objetos implica únicamente la creación o destrucción de la relación que existe entre ellos, pero nunca significa la creación o destrucción del otro objeto. Se dice que no hay una relación fuerte entre ambos objetos. La asociación simple responde a una relación que indica que un objeto de la clase A usa un objeto de la clase B y puede que viceversa también.

Por ejemplo, un escritor (un objeto de la clase `Escritor`) puede tener cero, uno o varios libros (objetos de la clase `Libro`) y cada libro pertenece únicamente a un escritor. Si se destruye un escritor, no significa que se destruyan sus libros, ya que estos libros se pueden utilizar para realizar otros cálculos. Más claro queda el caso contrario: si se elimina un libro de un escritor, no significa que ese escritor deje de existir. Es más, puede que no exista ningún libro por parte de un escritor y, sin embargo, puede existir ese escritor.



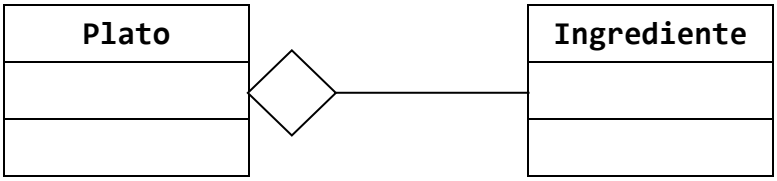
1.1.2. Asociación de Agregación

En una **asociación de agregación**, un objeto de una clase (llamada componente) es parte de un objeto de otra clase (llamada compuesto).

Este tipo de asociación se denomina también composición débil. Se da una subordinación conceptual del tipo “todo/parte” o bien “tiene un”. Así pues, se trata de un caso particular de la asociación simple en la que hay una cierta relación de ensamblaje, ya sea física o lógica, entre los objetos.

Los dos objetos relacionados entre sí existen de forma independiente. Es decir, la destrucción del objeto compuesto no implica la destrucción de los objetos componentes, ni viceversa.

Por ejemplo, un plato (un objeto de la clase `Plato`) está compuesto por cero, uno o varios ingredientes (objetos de la clase `Ingrediente`) y un ingrediente puede formar parte de varios platos. Si se elimina un plato, los ingredientes de este plato no se eliminan, ya que pueden estar en otros platos. Si se elimina un ingrediente de un plato, este plato no se elimina pero queda modificado.



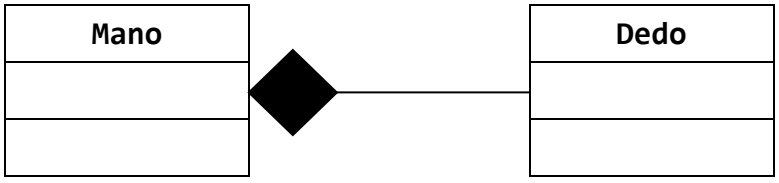
1.1.3. Asociación de Composición

La **asociación de composición** es un caso particular de la asociación de agregación:

- Un objeto de una clase (llamada componente) es parte de un objeto de otra clase (llamada compuesto). Cada componente solo puede estar presente en un único compuesto.
- La vida o existencia del objeto componente debe coincidir con la vida o existencia del objeto compuesto. Dicho de otro modo, la destrucción del objeto compuesto conlleva la destrucción de sus objetos componentes.

Este tipo de asociación se denomina también composición fuerte. Los dos objetos relacionados entre sí existen de forma dependiente: la destrucción del objeto compuesto implica la destrucción de los objetos componentes, y viceversa.

Un ejemplo muy visual de asociación de composición es el caso de la mano y su relación con los dedos. Si se elimina una mano, implícitamente se eliminan todos los dedos que la componen. Pero si se elimina un dedo de una mano, no se elimina toda la mano sino únicamente ese dedo.



1.2. PROPIEDADES DE LAS ASOCIACIONES

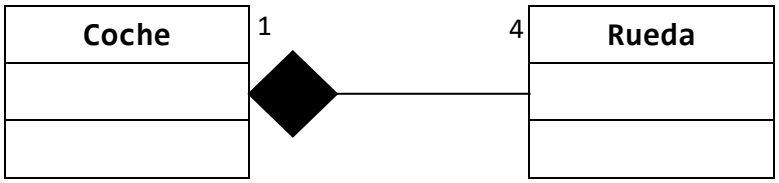
Las principales propiedades de las asociaciones son tres: cardinalidad, navegabilidad y rol.

1.2.1. Cardinalidad

La **cardinalidad** (también denominada multiplicidad) es el número de objetos de una clase con el que se puede relacionar un objeto de la otra clase de la relación.

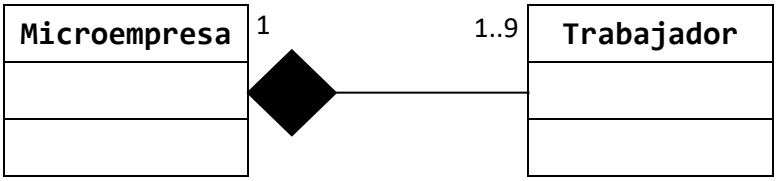
La cardinalidad se indica mediante modificadores que se añaden encima o debajo de la línea que representa la relación. Hay diferentes notaciones para indicar la cardinalidad de una asociación:

- **Número Exacto.** Indica que un objeto de la clase A tiene que estar relacionada exactamente con el número indicado de objetos de la clase B.



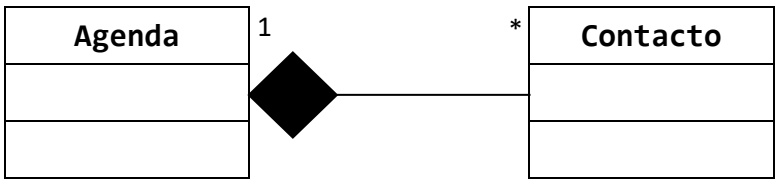
En este ejemplo, un coche tiene exactamente 4 ruedas y una rueda solo puede pertenecer a un coche.

- **Rango de Valores.** Indica el número mínimo y máximo de objetos de la clase B con los que tiene que estar relacionada un objeto de la clase A.



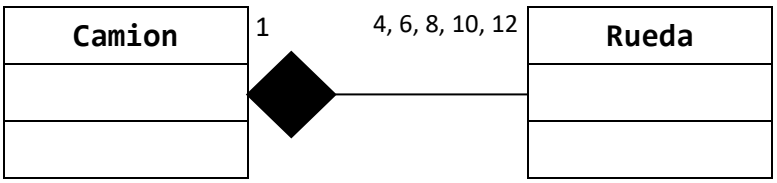
En este ejemplo, se considera que una microempresa tiene menos de diez trabajadores, que no existe ninguna empresa sin ningún trabajador (aunque sea el propio propietario) y que un trabajador solo puede estar en una microempresa. En otro contexto, esta cardinalidad podría ser diferente.

- **Muchos.** Utiliza el símbolo asterisco (*) para indicar que el número de objetos relacionados es "muchos" o "infinito". Es decir, con el asterisco se dice que un objeto de la clase A puede estar relacionado con cualquier número de objetos de la clase B (o incluso con ninguno) y que no se conoce este número. Escribir * es equivalente a indicar el rango `0..*`.



En este ejemplo, una agenda puede estar vacía (cero contactos) o tener muchos contactos (no se sabe el límite en este contexto), mientras que un contacto sólo puede estar en una agenda.

- **Rango de Valores No Consecutivos.** Separando números exactos y/o rangos por comas, se pueden hacer rangos no consecutivos.



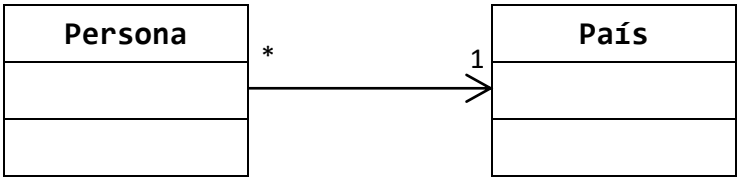
En este ejemplo, un camión puede tener 4, 6, 8, 10 o 12 ruedas y una rueda solo puede pertenecer a un camión.

1.2.2. Navegabilidad

La **navegabilidad** (también denominada direccionalidad) tiene sentido para asociaciones binarias e indica cuál de los dos objetos de la relación conoce al otro.

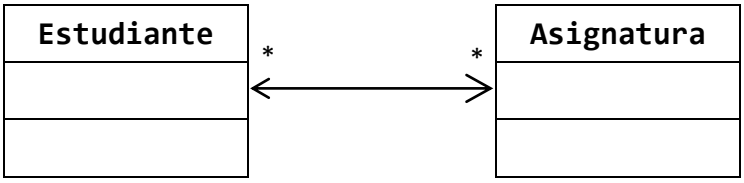
Hay dos opciones de navegabilidad:

- **Unidireccional.** Solo uno de los dos objetos tiene constancia de la existencia del otro objeto. Se representa mediante una punta de flecha en un extremo de la asociación.



En este ejemplo, una persona vive en un país, pero no hace falta almacenar qué personas viven en un país concreto, es decir, ningún país interactúa con ninguna persona.

- **Bidireccional.** Los dos objetos tienen constancia de la existencia del otro objeto. Se representa mediante una línea simple sin puntas de flecha, o también dibujando una punta de flecha en cada extremo de la asociación. Este tipo de navegabilidad es el más frecuente.



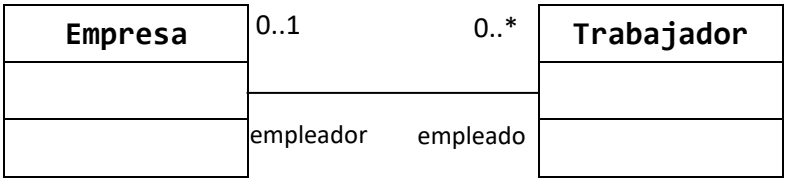
En este ejemplo, un estudiante se puede matricular en cero o varias asignaturas y una asignatura puede tener a cero o varios estudiantes. Interesa saber, para cada asignatura, la lista de estudiantes matriculados, y de cada estudiante, la lista de asignaturas que cursa.

1.2.3. Rol

A diferencia de la cardinalidad y la navegabilidad, el rol no modifica el comportamiento de una asociación, sino que añade información contextual que debe ayudar al lector a comprender la relación.

El **rol** es una etiqueta que se pone encima o debajo en cada lado de la asociación para nombrar a las clases participantes de una manera diferente, es decir, para indicar el rol de cada clase. Si la asociación se entiende sin poner los roles, entonces no hace falta escribirlos.

En este ejemplo, una empresa tiene cero o muchos trabajadores y un trabajador pertenece a una empresa o a ninguna. Además, cada empresa actúa con el rol de empleador y cada trabajador tiene el rol de empleado.



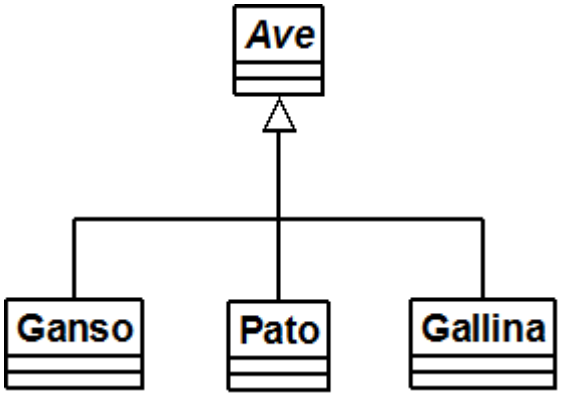
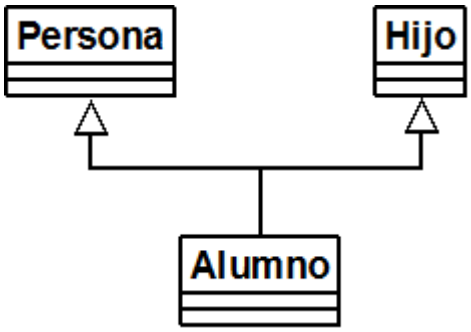
2. HERENCIA DE CLASES

La **herencia** es un mecanismo de la orientación a objetos que permite definir una nueva clase a partir de otra clase o de una jerarquía de clases preexistente, evitando con ello el rediseño, la modificación y verificación de la parte ya implementada. De esta forma, se facilita la creación de objetos partiendo de otros ya existentes previamente.

La herencia se usa con mucha frecuencia en el desarrollo de software porque permite la reutilización de código y la extensibilidad de funcionalidades.

Existen dos tipos de herencia de clases:

- **Herencia Simple.** La definición de una nueva clase se realiza partiendo de otra única clase ya existente. No se permite heredar de más de una clase. Java sólo permite la herencia simple.
- **Herencia Múltiple.** La definición de una nueva clase se realiza partiendo de varias clases ya existentes. Se permite heredar de varias clases. C++ permite la herencia múltiple.

EJEMPLO DE HERENCIA SIMPLE	EJEMPLO DE HERENCIA MÚLTIPLE
	

2.1. SUPERCLASES Y SUBCLASES

La herencia también es una relación de generalización/especialización entre una clase general y otra clase específica. A la clase general se la denomina **superclase**, **clase padre** o **clase base** y a la clase específica se la denomina **subclase**, **clase hija** o **clase derivada**.

Cuando se realiza herencia entre dos clases:

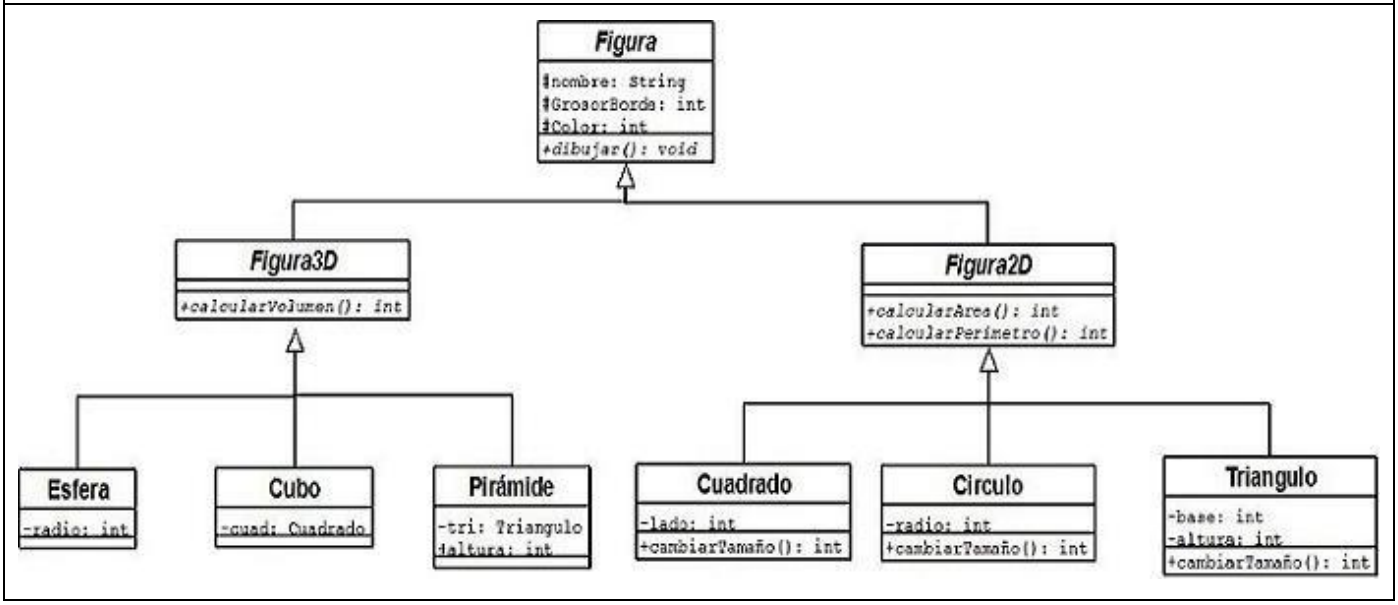
- La subclase hereda todos los miembros (atributos y métodos) de la superclase.
- La subclase podrá acceder a los miembros de la superclase dependiendo del nivel de acceso (público, protegido o privado) utilizado en la definición de cada miembro de la superclase.

  Si se aplica el principio de encapsulación, la subclase no podrá acceder directamente a los atributos privados de la superclase (con métodos *getters*/*setters* definidos sí tendría acceso a ellos) pero sí podrá acceder a los métodos públicos de la superclase.

  Además, la subclase podrá acceder a los miembros de la superclase que hayan sido definidos como públicos, protegidos o a nivel de paquete (si la subclase está en el mismo paquete que la superclase).

- La subclase puede incluir sus propios atributos y sus propios métodos (que serán específicos de la subclase).
- La subclase puede modificar (redefinir o sobrescribir) cualquier método heredado de la superclase con su propia implementación.
- Lo que es común a ambas clases queda comprendido en la superclase y lo que es específico queda restringido a la subclase.
- Puede haber varias subclases que hereden de la misma superclase. Cualquier clase puede convertirse en superclase mediante la herencia definiendo las subclases necesarias. Así se puede definir una jerarquía de clases con varios niveles.

EJEMPLO DE JERARQUÍA DE CLASES



Java utiliza la siguiente sintaxis para definir una nueva subclase mediante herencia a partir de una superclase ya existente:

```
class ClaseHija extends ClasePadre {
    ...
}
```

La clase **`java.lang.Object`** es la clase base de toda la jerarquía de clases de Java. Cualquier clase que se defina indicando que no hereda de ninguna otra clase, por defecto, heredará de esta clase `Object`. Esta clase proporciona, entre otros, los métodos `toString`, `equals` y `getClass`, los cuales pueden ser sobrescritos en la nueva clase.

2.2. CONSTRUCTORES ESPECIALES

Cuando existe una jerarquía de clases definidas con herencia y se crea un objeto de una subclase en memoria, se invocará el constructor de esta subclase. En este constructor, primero se deberá llamar al constructor de la superclase y después se ejecutará el resto del código del constructor de la subclase.

En Java existen dos constructores especiales: **`this`** y **`super`**. Ambos constructores se invocan siempre en la primera línea de código de cualquier otro constructor de la clase.

El uso de `this` y `super` es excluyente en un constructor de una clase: o se utiliza `this` o se utiliza `super`.

2.2.1. this

El constructor **`this`** representa a cualquier constructor de la propia clase. Cuando se utiliza `this`, realiza una llamada al constructor de la misma clase cuyos parámetros formales coincidan en número, orden y tipo con los parámetros actuales indicados en la llamada `this`.

El siguiente ejemplo muestra el uso del constructor `this` en Java:

```
public class Persona {
    private String nombre;
    private int edad;

    public Persona() {
        this("Fulanito", 18);
    }

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

2.2.2. super

El constructor **`super`** representa a cualquier constructor de la superclase. Cuando se utiliza `super`, realiza una llamada de manera explícita al constructor de la superclase cuyos parámetros formales coincidan en número, orden y tipo con los parámetros actuales indicados en la llamada `super`:

- Si no hay que pasar ningún parámetro, no es necesario llamar al constructor de la superclase con `super`, ya que se ejecuta automáticamente el constructor por defecto de la superclase.
- Se debe invocar `super` de forma explícita cuando el constructor de la superclase espera recibir uno o más parámetros.
- La llamada al constructor de la superclase debe realizarse con `super` en la primera línea de código del constructor de la subclase.

El siguiente ejemplo muestra el uso del constructor `super` en Java:

```
public class Deportista extends Persona {
    private String deporte;

    public Deportista() {
        this("Menganito", 25, "Tenis");
    }

    public Deportista(String nombre, int edad, String deporte) {
        super(nombre, edad);
        this.deporte = deporte;
    }
}
```

2.3. MODIFICADORES DE CLASES, ATRIBUTOS Y MÉTODOS

Java dispone de distintos **modificadores** que pueden aplicarse a una clase, un atributo y/o un método: `static`, `final`, `public`, `protected`, `private`, `abstract` y `native`.

2.3.1. Modificadores de Clase

Los **modificadores de clase** son:

- **`public`.** Es un modificador de visibilidad. Si se pone delante de una clase, esta clase podrá ser utilizada fuera del paquete donde está definida. Si no se usa este modificador, por defecto será `package`, lo que significa que la clase solo podrá ser utilizada por las clases que están en el mismo paquete donde está definida.
- **`abstract`.** Indica que una clase es abstracta, es decir, que contiene al menos un método sin implementación (o método abstracto). Una clase abstracta sirve exclusivamente para definir subclases de ella (o una jerarquía de clases) mediante herencia y no se puede instanciar objetos de este tipo de clases.
- **`final`.** Impide definir subclases de esta clase mediante herencia. Con este modificador, no se permitirá crear subclases de esta clase.

2.3.2. Modificadores de Atributos y Métodos

La **visibilidad u ocultación de un miembro de una clase** se define en relación con el resto de clases. Una clase siempre puede acceder a todos los atributos y métodos que se han definido en ella.

Los **modificadores de atributos y métodos** son los modificadores de visibilidad, que permiten controlar el acceso a los miembros de una clase desde otras clases y desde las propias clases que heredan de ella:

- **`public`.** Cuando un miembro de una clase se define público, la clase y cualquier otra clase pueden acceder a dicho miembro para utilizarlo. No conviene que los atributos sean públicos.
- **`protected`.** Cuando un miembro de una clase se define protegido, la clase y cualquier otra clase descendiente (a través de herencia) de esta clase pueden acceder a dicho miembro para utilizarlo.
- **`private`.** Cuando un miembro de una clase se define privado, sólo la clase puede acceder a dicho miembro para utilizarlo. Para acceder a un atributo privado desde fuera de la clase, se deberá utilizar un método público de acceso a dicho atributo (*setter* o *getter*).
- **`package`.** Cuando no se pone ningún modificador de los anteriores a un miembro de una clase, la clase y cualquier otra clase definida en el mismo paquete que ésta puede acceder a dicho miembro para utilizarlo. Si una clase hereda miembros de tipo `package`, podrá hacer uso de ellos siempre que la superclase y las subclases estén en el mismo paquete.

2.3.3. Modificadores Exclusivos de Atributos

Los **modificadores exclusivos de atributos** son:

- **`static`.** Cuando un atributo de una clase se define `static`, esto implica que el valor de dicho atributo será el mismo para todos los objetos instanciados de esa clase. Si se modifica ese valor,

se cambia para todos los objetos. Esto se debe a que dicho valor no está repetido para cada objeto, sino que existe una única copia.

Sirve para definir atributos de clase (atributos estáticos), los cuales serán compartidos por todos los objetos creados de la clase.

- **`final`.** Cuando un atributo de una clase se define `final`, esto obliga a darle un valor inicial. Esta inicialización solo puede realizarse en dos sitios: en la declaración del atributo dentro de la clase o en el constructor de la clase. Si se opta por inicializarlo en el constructor, el compilador obligará a inicializar este atributo en todos los constructores de la clase.

  Sirve para definir atributos constantes, cuyos valores, una vez inicializados, no podrán ser modificados durante la ejecución del programa.

2.3.4. Modificadores Exclusivos de Métodos

Los **modificadores exclusivos de métodos** son:

- **`static`.** Cuando un método de una clase se define `static`, se permitirá utilizar dicho método con el nombre de la clase y sin tener que instanciar un objeto de la clase.

  Sirve para definir métodos de clase (métodos estáticos), los cuales serán compartidos por todos los objetos creados de la clase. Se suelen utilizar para acceder a atributos estáticos.

- **`final`.** Cuando un método de una clase se define `final`, esto impide que cualquier subclase que lo herede mediante herencia sobrescriba o redefina este método. Con este modificador, no se permitirá sobrescribir o redefinir un método en las subclases definidas con herencia.

- **`abstract`.** Cuando un método de una clase se define `abstract`, esto indica que este método será abstracto, es decir, no tendrá cuerpo o implementación en esta clase.

  Sirve para definir operaciones genéricas comunes para varios tipos. Normalmente, cuando se define un método abstracto en una superclase, se indica únicamente la cabecera del método (nombre, parámetros, tipo de retorno y otros modificadores) y en cada subclase se añade la implementación del método correspondiente, la cual dependerá específicamente de la propia subclase.

- **`native`.** Cuando un método de una clase se define `native`, esto indica que el cuerpo o implementación de este método está codificado en otro lenguaje de programación (distinto de Java).

2.4. SOBRESCRITURA O REDEFINICIÓN DE MÉTODOS

La **sobrescritura o redefinición de métodos** solamente es posible cuando hay herencia de clases. Cuando se codifica una subclase mediante herencia a partir de una superclase, la subclase puede sobrescribir o redefinir los métodos heredados de la superclase.

Para sobrescribir o redefinir un método, se deben cumplir las siguientes reglas:

- El nombre del método debe ser el mismo tanto en la superclase como en la subclase.
- Los parámetros del método deben ser los mismos (en número, orden y tipo) tanto en la superclase como en la subclase. Si no coinciden, ya no se hace referencia al método de la superclase sino que sería otro método distinto, por lo que habría sobrecarga del método.

- El tipo de datos de retorno del método debe ser el mismo tanto en la superclase como en la subclase.
- El modificador de visibilidad del método de la subclase no debe ser más restrictivo que el modificador de visibilidad del método de la superclase. Por ejemplo:
  - Si el método se define `public` en la superclase, en la subclase únicamente podrá definirse `public`.
  - Si el método se define `protected` en la superclase, en la subclase podrá definirse `protected` o `public`.
- Si el método se define como método de clase (`static`) en la superclase, en la subclase también deberá definirse como método de clase (`static`).
- Las excepciones que el pueda lanzar el método deben ser las mismas tanto en la superclase como en la subclase.

El cuerpo o implementación del método sobrescrito en la subclase solamente afectará a la propia subclase y a las clases descendientes que hereden de esta. Este método sobrescrito no afectará en ningún caso a la superclase ni a las clases ascendientes.

Si un método se define con el modificador `final` en la superclase, este método no podrá ser sobrescrito en ninguna subclase ni en clases descendientes.

Cuando hay sobrescritura o redefinición de métodos en Java, la referencia `super` permite acceder desde la subclase al método de la superclase.

El siguiente ejemplo muestra la sobrescritura de un método en Java:

```
public class Empleado {
    private String nombre;

    public Empleado(String nombre) {
        this.nombre = nombre;
    }

    public double calcularSalario(int numeroHorasExtra) {
        return 1400.25 + numeroHorasExtra * 19.95;
    }
}
```

```
public class JefeDepartamento extends Empleado {
    private String departamento;

    public JefeDepartamento(String nombre, String departamento) {
        super(nombre);
        this.departamento = departamento;
    }

    @Override
    public double calcularSalario(int numeroHorasExtra) {
        return 1600.75 + numeroHorasExtra * 24.95;
    }
}
```

3. CLASES ABSTRACTAS E INTERFACES

Cuando se define una nueva clase en Java:

- Se puede realizar herencia simple indicando que la nueva clase hereda de otra clase:

  ```
  public class JefeDepartamento extends Empleado
  ```

- Se puede añadir un comportamiento definido previamente indicando que la nueva clase implementa una o varias interfaces:

  ```
  public class Botella implements Comparable
  public abstract class Forma2D implements Modificable, Dibujable
  ```

3.1. MÉTODOS ABSTRACTOS

Un **método abstracto** se define con el modificador `abstract` y tiene las siguientes características:

- Cuando se define un método abstracto en una clase, solo se indica la cabecera o signatura del método (incluyendo nombre, parámetros, tipo de retorno y otros modificadores). No tiene cuerpo o implementación.
- No se puede definir un método abstracto en una clase que no sea abstracta. Es decir, un método abstracto sólo puede existir dentro de una clase abstracta. Dicho de otra manera, si una clase contiene al menos un método abstracto, forzosamente la clase será una clase abstracta.
- Al realizar herencia de una clase abstracta, todos los métodos abstractos de esta clase deberán sobrescribirse o redefinirse obligatoriamente en cada subclase no abstracta. Si una subclase no implementa un método abstracto heredado de la superclase, esto forzará a que la subclase sea también abstracta.

3.2. CLASES ABSTRACTAS

Una **clase abstracta** se define con el modificador `abstract` y tiene las siguientes características:

- Cuando se define una clase abstracta, ésta debe contener al menos un método abstracto. Si no es así, no tiene sentido definir la clase como abstracta.
- Una clase abstracta puede contener atributos.
- Una clase abstracta puede contener métodos que no sean abstractos. Al realizar herencia de una clase abstracta, la subclase no está obligada a implementar los métodos heredados que no sean abstractos: puede sobrescribirlos o no, según convenga.
- Se pueden declarar objetos del tipo de una clase abstracta, pero no se pueden instanciar objetos de ninguna clase abstracta.

Normalmente, una clase abstracta se utiliza como superclase de una jerarquía de clases a diseñar y en ella se incluyen características y comportamiento comunes de todas las clases de esta jerarquía. Cada subclase (que no sea abstracta) que hereda de la clase abstracta tendrá la obligación de implementar, sobrescribir o redefinir los métodos abstractos heredados. Así, la subclase concretará o especificará el comportamiento común definido en la superclase.

3.3. INTERFACES

Una **interfaz** define un protocolo de comportamiento y proporciona un formato común para implementarlo en las clases. Una interfaz contiene exclusivamente métodos abstractos y atributos constantes. Es decir, todos los métodos definidos en una interfaz son abstractos, son métodos con cabecera pero sin cuerpo o implementación.

Normalmente, una interfaz se utiliza para definir un comportamiento abstracto (sin implementación) que podría ser común en el futuro para diferentes clases que no están relacionadas (situadas en distintas jerarquías de clases sin relaciones de herencia).

El concepto de interfaz se ido ampliando con el tiempo. A partir de Java 8, una interfaz puede contener además métodos por defecto, métodos estáticos y tipos anidados. A partir de Java 9, una interfaz puede incluir también métodos privados.

Una interfaz **tiene en común con una clase** lo siguiente:

- Puede contener métodos. Implícitamente estos métodos son abstractos, por lo que no hace falta definirlos con el modificador `abstract`.
- Se escribe en un fichero con extensión `.java` que debe denominarse exactamente igual que la interfaz.
- Al compilar el programa, el código intermedio *bytecode* de la interfaz se guarda en un fichero `.class`.

Una interfaz **se diferencia de una clase** en lo siguiente:

- Si contiene atributos, estos deberán ser públicos, estáticos y constantes (`public static final`) y estar inicializados con un valor en el momento de su declaración.
- No contiene constructores.
- No se pueden crear o instanciar objetos a partir de una interfaz.
- Una interfaz puede heredar de una o varias interfaces. Se permite la herencia múltiple de interfaces en Java.

```
public interface InterfazHija extends InterfazPadre
public interface SubInterfaz extends SuperInterfaz1, SuperInterfaz2
```

Al trabajar con clases e interfaces, hay que tener en cuenta que:

- Las clases no heredan las interfaces. Las clases implementan las interfaces.
- Una clase puede implementar una o varias interfaces. Cuando una clase implementa una interfaz, la clase tiene la obligación realizar una de las dos opciones:
  - Implementar todos los métodos abstractos de la interfaz.
  - Definir la propia clase como clase abstracta y no implementar todos los métodos abstractos de la interfaz (y seguir conteniendo al menos un método abstracto).

Se puede declarar un objeto con el nombre de una interfaz, pero no se puede instanciar indicando el nombre de la interfaz. De este modo, si se define un objeto cuyo tipo es una interfaz, posteriormente se le podrá asignar un objeto que sea una instancia de una clase que implemente dicha interfaz.

Esto significa que, utilizando interfaces como tipos, se puede aplicar el polimorfismo a clases que no están relacionadas mediante herencia, pero sí por el mecanismo de implementación de una interfaz.

4. POLIMORFISMO

En programación orientada a objetos, el **polimorfismo** se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

En lenguajes basados en clases y con un sistema de tipos de datos fuerte (como Java y C++), existen dos maneras de poder utilizar objetos polimórficos:

1) **Mediante una jerarquía de clases.** Una jerarquía de clases permite que los objetos de estas clases compartan una raíz común y proporciona la compatibilidad de tipos de datos necesaria para que sea posible utilizar una misma variable de referencia (que podrá apuntar a objetos de diversas subclases de la jerarquía) para enviar el mismo mensaje (o un grupo de mensajes) a los objetos que se tratan de manera polimórfica.

   Es decir, un objeto de una clase se podrá comportar como un objeto de cualquiera de sus clases descendientes de la jerarquía porque la referencia al objeto podrá apuntar a objetos de sus clases derivadas.

2) **A través de interfaces.** Dos clases que implementan la misma interfaz se pueden tratar de forma idéntica, como si fuera un mismo tipo de objeto, el tipo definido por la interfaz. Así, dos objetos de estas dos clases podrán intercambiarse mensajes en tiempo de ejecución, y además con dependencias mínimas entre ellos.

   Las interfaces se utilizan para lograr la necesaria concordancia de tipos que hace posible el polimorfismo. Una clase que implementa una interfaz sólo obtiene su tipo de datos y la obligación de implementar sus métodos, no copia comportamiento ni atributos.

A menudo, cuando una clase hereda de otra y además implementa una o varias interfaces, se favorece una utilización más amplia del polimorfismo y se evita la necesidad de la herencia múltiple.

Existen dos grandes **tipos de polimorfismo**:

- **Polimorfismo Estático o ad-hoc (a medida).** Los tipos a los que se aplica el polimorfismo deben ser explícitos y declarados antes de poder ser utilizados.
- **Polimorfismo Dinámico o Paramétrico.** El código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, se puede aplicar el polimorfismo a todo tipo de datos compatible.

4.1. POLIMORFISMO ESTÁTICO O AD-HOC

El **polimorfismo estático** (también denominado *ad-hoc* o a medida) hace referencia a funciones que pueden tener una serie de implementaciones distintas según los tipos de los parámetros aplicados.

Cuando se aplica a la orientación a objetos, se conoce como **sobrecarga de operadores** o **sobrecarga de métodos**. En este caso, la sobrecarga permite definir múltiples funciones o métodos con el mismo nombre, pero con número, orden o tipos de parámetros distintos.

Este tipo de polimorfismo se resuelve o determina en **tiempo de compilación**. El traductor (compilador o intérprete) se encarga automáticamente de invocar la función o método correcto entre las disponibles en la sobrecarga. Esto permite analizar el código para optimizarlo y ejecutarlo más rápido.

El siguiente ejemplo muestra una sobrecarga de operadores en Java:

```
public static void main(String[] args) {
    System.out.println(2 + 3);              // sumar dos números enteros
    System.out.println(5.7 + 4.1);          // sumar dos números reales
    System.out.println("buenos" + " días"); // concatenar dos cadenas de texto
}
```

El siguiente ejemplo muestra una sobrecarga de métodos en Java:

```
// Rellena un vector con números enteros aleatorios comprendidos
// entre un mínimo y un máximo.
public static void rellenar(int[] vector, int minimo, int maximo) {
    Random aleatorio = new Random();
    for (int pos = 0 ; pos < vector.length ; pos++) {
        vector[pos] = aleatorio.nextInt(maximo – minimo + 1) + minimo;
    }
}
```

```
// Rellena un vector con números reales aleatorios comprendidos
// entre un mínimo y un máximo.
public static void rellenar(double[] vector, double minimo, double maximo) {
    Random aleatorio = new Random();
    for (int pos = 0 ; pos < vector.length ; pos++) {
        vector[pos] = aleatorio.nextDouble() * (maximo – minimo) + minimo;
    }
}
```

```
// Rellena un vector con booleanos aleatorios.
public static void rellenar(boolean[] vector) {
    Random aleatorio = new Random();
    for (int pos = 0 ; pos < vector.length ; pos++) {
        vector[pos] = aleatorio.nextBoolean();
    }
}
```

```
// Rellena un vector con caracteres aleatorios comprendidos entre A y Z.
public static void rellenar(char[] vector) {
    Random aleatorio = new Random();
    int minimo = 'A', maximo = 'Z', codigo;
    for (int pos = 0 ; pos < vector.length ; pos++) {
        codigo = aleatorio.nextInt(maximo – minimo + 1) + minimo;
        vector[pos] = (char) codigo;
    }
}
```

4.2. POLIMORFISMO DINÁMICO O PARAMÉTRICO

El **polimorfismo dinámico** (también denominado paramétrico) hace referencia a funciones y tipos de datos que se pueden codificar genéricamente para que pueda manejar valores de manera idéntica sin depender de su tipo de datos.

Cuando se aplica a la orientación a objetos, estas funciones se denominan **métodos genéricos** y estos tipos de datos se denominan **clases genéricas**. Ambos forman la base de la **programación genérica**:

- La programación genérica se define como un estilo de programación en el que los métodos se codifican en términos de tipos de datos (clases) que se especificarán más adelante y que luego se utilizarán cuando se necesiten para tipos (clases) específicos suministrados como parámetros.
- Este enfoque permite escribir métodos y tipos de datos (clases) comunes que difieren solo en el conjunto de tipos sobre los que pueden operar, reduciendo así la duplicación de código.

Este tipo de polimorfismo se resuelve o determina en **tiempo de ejecución**. Cuando se envía un mensaje a un objeto, el programa se encarga de averiguar a qué clase pertenece ese objeto instanciado y de invocar el método correspondiente de dicha clase. Esto da flexibilidad para codificar pero ejecuta el programa un poco más lento.

Para que exista polimorfismo con una jerarquía de clases se tienen que dar tres condiciones:

- Que haya herencia de clases.
- Que haya sobrescritura o redefinición de métodos en las subclases.
- Declarar un objeto con una superclase e instanciarlo con una subclase.

Para que exista polimorfismo a través de una interfaz se tienen que dar tres condiciones:

- Que haya una interfaz.
- Que haya clases que implementen la interfaz (con sobrescritura o redefinición de métodos).
- Declarar un objeto con la interfaz e instanciarlo con una clase que implemente la interfaz.

5. CASOS PRÁCTICOS

5.1. ASOCIACIÓN DE CLASES

El siguiente ejemplo muestra una asociación entre clases, concretamente una composición.

Por un lado, está definida la clase `Fecha` con sus atributos, constructor y método `toString` y, por otro lado, está definida la clase `Persona` que tiene dos atributos: el nombre y la fecha de nacimiento, que es un objeto de la clase `Fecha`. El constructor de la clase `Persona` recibe como parámetros, además del nombre, los valores necesarios para instanciar un objeto de la clase `Fecha`.

En la clase `Principal`, se realiza la instanciación de un objeto de la clase `Persona` y la llamada a su método `toString`.

```
public class Fecha {

    private int dia;
    private int mes;
    private int agno;

    public Fecha(int dia, int mes, int agno) {
        this.dia = dia;
        this.mes = mes;
        this.agno = agno;
    }

    @Override
    public String toString() {
        return "Fecha [día=" + dia + ", mes=" + mes + ", año=" + agno + "]";
    }

}
```

```
public class Persona {

    private String nombre;
    private Fecha fechaNacimiento;

    public Persona(String nombre, int dia, int mes, int agno) {
        this.nombre = nombre;
        this.fechaNacimiento = new Fecha(dia, mes, agno);
    }

    @Override
    public String toString() {
        return "Persona [nombre=" + nombre +
                ", fechaNacimiento=" + fechaNacimiento + "]";
    }

}
```

```java
public class PrincipalAsociacion {

    public static void main(String[] args) {
        Persona persona;
        persona = new Persona("Fulanito", 2, 5, 1994);
        System.out.println(persona.toString());
    }

}
```

5.2. HERENCIA DE CLASES

El siguiente ejemplo muestra la declaración de una clase como clase hija o derivada de otra existente. Además, también se muestra una composición.

La clase `Empleado` deriva de la clase `Persona` que se ha definido en el ejemplo anterior, por lo tanto, esta clase tendrá lo mismo que su clase padre, además de lo que se añada. Aunque al estar los atributos definidos como privados, no se podrá acceder a ellos directamente.

En el constructor, la primera sentencia es la llamada al constructor de la clase padre. Con esta instrucción se inicializan los valores de los atributos declarados en la clase `Persona`.

En el método `toString`, se hace una llamada al método `toString` de la clase padre, que es un método sobrescrito.

En la clase `Principal`, se instancia un objeto para la fecha de alta del empleado y se usa justo con otros literales para instanciar un objeto de la clase `Empleado` y mostrar su estado mediante el método `toString`.

```java
public class Empleado extends Persona {

    private String departamento;
    private Fecha fechaAlta;

    public Empleado(String departamento, Fecha fechaAlta,
            String nombre, int dia, int mes, int agno) {
        super(nombre, dia, mes,  agno);
        this.departamento = departamento;
        this.fechaAlta = fechaAlta;
    }

    @Override
    public String toString() {
        return "Empleado ["+ super.toString() +
                ", departamento=" + departamento +
                ", fechaAlta=" + fechaAlta.toString() + "]";
    }

}
```

```java
public class PrincipalHerencia {

    public static void main(String[] args) {
        Empleado empleado;
        Fecha fAlta = new Fecha(1, 1, 2022);
        empleado = new Empleado("Informática", fAlta, "Fulanito", 2, 5, 1994);
        System.out.println(empleado.toString());
    }

}
```

5.3. IMPLEMENTACIÓN DE UNA INTERFAZ

El siguiente ejemplo muestra la definición de la interfaz `Figura` y su implementación en las clases `Circulo` y `Cuadrado`.

Esta interfaz tiene definidos dos métodos, `calcularArea` y `calcularPerimetro`, que deberán ser implementados en las clases `Circulo` y `Cuadrado`.

En la clase `Principal` se instancia un objeto de la clase `Circulo` y otro de la clase `Cuadrado` y se hace uso de sus métodos.

```java
public interface Figura {

    public double calcularArea();

    public double calcularPerimetro();

}
```

```java
public class Circulo implements Figura {

    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

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

    @Override
    public double calcularPerimetro() {
        return  2 * Math.PI * radio;
    }

}
```

```java
public class Cuadrado implements Figura {

    private double lado;

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

    @Override
    public double calcularArea() {
        return lado * lado;
    }

    @Override
    public double calcularPerimetro() {
        return 4 * lado;
    }

}
```

```
import entrada.Teclado;

public class PrincipalInterfaz {
    public static void main(String[] args) {
        Circulo circulo;
        Cuadrado cuadrado;
        double radio, lado;

        radio = Teclado.leerReal("¿Radio del círculo? ");
        lado = Teclado.leerReal("¿Lado del cuadrado? ");

        circulo = new Circulo(radio);
        cuadrado = new Cuadrado(lado);

        System.out.println("Área círculo: " + circulo.calcularArea());
        System.out.println("Perímetro círculo: " + circulo.calcularPerimetro());

        System.out.println("Área cuadrado: " + cuadrado.calcularArea());
        System.out.println("Perímetro cuadrado: " + cuadrado.calcularPerimetro());
    }
}
```

5.4. USO DE UNA CLASE ABSTRACTA

El siguiente ejemplo muestra la definición de la clase abstracta `ObraArte` y la definición de dos clases derivadas de ésta: `Pintura` y `Escultura`.

Esta clase abstracta tienen definido el método abstracto `calcularPrecio`, que deberá ser implementado en las clases `Pintura` y `Escultura`.

En la clase `Principal`, se instancia un objeto de clase `Pintura` y otro de la clase `Escultura` y se hace uso de sus métodos.

```java
public abstract class ObraArte {

    private static int contador = 0;

    private int codigo;
    private String titulo;
    private String autor;
    private int agnoRealizacion;

    public ObraArte(String titulo, String autor, int agnoRealizacion) {
        ObraArte.contador++;
        this.codigo = ObraArte.contador;
        this.titulo = titulo;
        this.autor = autor;
        this.agnoRealizacion = agnoRealizacion;
    }

    public int getAgnoRealizacion() {
        return this.agnoRealizacion;
    }

    public abstract double calcularPrecio(int agnoVenta);

    @Override
    public String toString() {
        return
            "Código = " + this.codigo +
            ", Título = " + this.titulo +
            ", Autor = " + this.autor +
            ", AñoRealización = " + this.agnoRealizacion;
    }

}
```

```java
public class Pintura extends ObraArte {

    private double anchuraSoporte;
    private double alturaSoporte;

    public Pintura(String titulo, String autor, int agnoRealizacion,
                   double anchuraSoporte, double alturaSoporte) {
        super(titulo, autor, agnoRealizacion);
        if (anchuraSoporte > 0.0 && alturaSoporte > 0.0) {
            this.anchuraSoporte = anchuraSoporte;
            this.alturaSoporte = alturaSoporte;
        }
        else {
            this.anchuraSoporte = 1.95;
            this.alturaSoporte = 0.97;
            System.out.println("El tamaño del soporte no es válido.");
            System.out.println("Asignado el tamaño por defecto: 1.95 x 0.97");
        }
    }

    @Override
    public String toString() {
        return
            "Pintura [" +
            super.toString() +
            ", AnchuraSoporte = " + this.anchuraSoporte +
            ", AlturaSoporte = " + this.alturaSoporte +
            "]";
    }

    @Override
    public double calcularPrecio(int agnoVenta) {
        double areaSoporte = this.anchuraSoporte * this.alturaSoporte;
        return (agnoVenta - this.getAgnoRealizacion()) * areaSoporte;
    }

}
```

```java
public class Escultura extends ObraArte {

    private String material;
    private double peso;

    public Escultura(String titulo, String autor, int agnoRealizacion,
                     String material, double peso) {
        super(titulo, autor, agnoRealizacion);
        boolean validos = false;
        if (material.equals("madera") || material.equals("piedra")
                || material.equals("metal")) {
            if (peso > 0.0) {
                validos = true;
            }
        }
        if (validos) {
            this.material = material;
            this.peso = peso;
        }
        else {
            this.material = "piedra";
            this.peso = 20.0;
            System.out.println("El material o el peso no es válido.");
            System.out.println("Se ha asignado el material por defecto: piedra");
            System.out.println("Se ha asignado el peso por defecto: 20");
        }
    }

    @Override
    public String toString() {
        return
            "Escultura [" +
            super.toString() +
            ", Material = " + this.material +
            ", Peso = " + this.peso +
            "]";
    }

    @Override
    public double calcularPrecio(int agnoVenta) {
        return (agnoVenta - this.getAgnoRealizacion()) * this.peso;
    }

}
```

```
import entrada.Teclado;

public class PrincipalClaseAbstracta {

    public static void main(String[] args) {
        Pintura pintura;
        Escultura escultura;

        int agnoRealizacion;
        String titulo, autor, material;
        double anchuraSoporte, alturaSoporte, peso;

        System.out.println("Datos de pintura:");
        titulo = Teclado.leerCadena("¿Título? ");
        autor = Teclado.leerCadena("¿Autor? ");
        agnoRealizacion = Teclado.leerEntero("¿Año de Realización? ");
        anchuraSoporte = Teclado.leerReal("¿Anchura del Soporte? ");
        alturaSoporte = Teclado.leerReal("¿Altura del Soporte? ");
        pintura = new Pintura(titulo, autor, agnoRealizacion,
                anchuraSoporte, alturaSoporte);

        System.out.println("Datos de escultura:");
        titulo = Teclado.leerCadena("¿Título? ");
        autor = Teclado.leerCadena("¿Autor? ");
        agnoRealizacion = Teclado.leerEntero("¿Año de Realización? ");
        material = Teclado.leerCadena("¿Material? ");
        peso = Teclado.leerReal("¿Peso? ");
        escultura = new Escultura(titulo, autor, agnoRealizacion, material, peso);

        System.out.println(pintura.toString());
        System.out.println(escultura.toString());
    }

}
```

5.5. POLIMORFISMO

El siguiente ejemplo muestra la clase `GaleriaArte`, que hace uso de las clases del ejemplo anterior. Contiene un vector de objetos y métodos para su gestión.

En la clase `Principal` se realiza la gestión de un objeto de la clase `GaleriaArte` mediante un menú de opciones.

```
public class GaleriaArte {

    private ObraArte[] obras;
    private int indice;

    public GaleriaArte(int capacidad) {
        this.obras = new ObraArte[capacidad];
        this.indice = 0;
    }

    @Override
    public String toString() {
        String cadena = "";
        if (this.indice == 0) {
            cadena = "La galería de arte está vacía.";
        }
        else {
            for (int posicion = 0 ; posicion < this.indice ; posicion++) {
                cadena = cadena + "(" + posicion + ") " +
                    this.obras[posicion].toString() + "\n" +
                    "Precio=" + this.obras[posicion].calcularPrecio(2021) + "\n";
            }
        }
        return cadena;
    }

    public boolean insertar(ObraArte obra) {
        boolean insertado = false;
        if (this.indice < this.obras.length) {
            this.obras[this.indice] = obra;
            this.indice++;
            insertado = true;
        }
        return insertado;
    }

    public boolean eliminar(int posicion) {
        boolean eliminado = false;
        if (posicion >= 0 && posicion < this.indice) {
            while (posicion < this.indice - 1) {
                this.obras[posicion] = this.obras[posicion + 1];
                posicion++;
            }
            this.obras[posicion] = null;
            this.indice--;
            eliminado = true;
        }
        return eliminado;
    }

}
```

```
import entrada.Teclado;

public class PrincipalPolimorfismo {

    public static void visualizarMenuOpciones() {
        System.out.println("-----------------------------------------------------------------");
        System.out.println("(0) Salir.");
        System.out.println("(1) Insertar una pintura en la galería de arte.");
        System.out.println("(2) Insertar una escultura en la galería de arte.");
        System.out.println("(3) Eliminar, por posición, una obra de arte de la galería de arte.");
        System.out.println("-----------------------------------------------------------------");
    }

    public static void main(String[] args) {

        GaleriaArte galeria = new GaleriaArte(20);

        int opcion, agnoRealizacion, posicion;
        String titulo, autor, material;
        double anchuraSoporte, alturaSoporte, peso;
        Pintura pintura;
        Escultura escultura;
        do {
            visualizarMenuOpciones();
            opcion = Teclado.leerEntero("¿Opción (0-4)? ");
            switch (opcion) {

            case 0:
                break;
            case 1:
                titulo = Teclado.leerCadena("¿Título? ");
                autor = Teclado.leerCadena("¿Autor? ");
                agnoRealizacion = Teclado.leerEntero("¿Año de Realización? ");
                anchuraSoporte = Teclado.leerReal("¿Anchura del Soporte? ");
                alturaSoporte = Teclado.leerReal("¿Altura del Soporte? ");
                pintura = new Pintura(titulo, autor, agnoRealizacion, anchuraSoporte, alturaSoporte);
                if (galeria.insertar(pintura)) {
                    System.out.println("Se ha insertado la pintura en la galería de arte.");
                }
                else {
                    System.out.println("Error al insertar: galería de arte llena.");
                }
                break;
            case 2:
                titulo = Teclado.leerCadena("¿Título? ");
                autor = Teclado.leerCadena("¿Autor? ");
                agnoRealizacion = Teclado.leerEntero("¿Año de Realización? ");
                material = Teclado.leerCadena("¿Material? ");
                peso = Teclado.leerReal("¿Peso? ");
                escultura = new Escultura(titulo, autor, agnoRealizacion, material, peso);
                if (galeria.insertar(escultura)) {
                    System.out.println("Se ha insertado la escultura en la galería de arte.");
                }
                else {
                    System.out.println("Error al insertar: galería de arte llena.");
                }
                break;
            case 3:
                posicion = Teclado.leerEntero("¿Posición? ");
                if (galeria.eliminar(posicion)) {
                    System.out.println("Se ha eliminado la obra de arte de la galería de arte.");
                }
                else {
                    System.out.println("Error al eliminar: galería de arte vacía o posición incorrecta.");
                }
                break;
            default:
                System.out.println("La opción de menú no es válida.");
                break;
            }
        }
        while (opcion != 0);
    }

}
```