Clases Avanzado

https://bronze-infinity-741.notion.site/Clases-Avanzado-b0fe5d2eac3544e79fa980dcc420bafb?pvs=4

▼ Relaciones entre clases.

Tipos:

- Especialización → La especialización es la creación de una nueva clase a partir de una clase existente, con funcionalidad adicional o modificada.
- Generalización → La generalización es el proceso de abstracción de características comunes de varias clases en una clase genérica o de nivel superior.

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

▼ Composición.

La *composición* se da cuando una clase contiene algún atributo que es una referencia a un objeto de otra clase.(“tiene un”)

▼ Sintaxis de la composición.

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

```
class Rectangulo {
     private Punto vertice1;
     private Punto vertice2;
…
}
```

▼ Uso de la composición (I). Preservación de la ocultación.

> Cuando vayas a devolver un objeto habrás de obrar con mucha precaución. Si en un método de la clase devuelves directamente un objeto que es un atributo, estarás ofreciendo directamente una **referencia**
> a un objeto atributo que probablemente has definido como privado. ¡De esta

> forma estás **volviendo a hacer público un atributo que inicialmente era privado**!

Debemos crear una **copia del objeto** para devolverlo, a menos que sea estático. Ejemplos:

```
public Punto getVertice1 () {  // Creación de un nuevo punto extrayendo sus atributos
     double x, y;
     Punto p;
     x= this.vertice1.getX();
     y= this.vertice1.getY();
     p= new Punto (x,y);

     return p;
}

public Punto getVertice1 () { // Utilizando el constructor copia de Punto (si es que está definido)
     Punto p;
     p= new Punto (this.vertice1); // Uso del constructor copia
     return p;
}
```

▼ Uso de la composición (II). Llamadas a constructores.

El constructor de la clase contenedora debe invocar a los constructores de las clases de los objetos contenidos.

> En este caso hay que tener cuidado con las referencias a **objetos que se pasan como parámetros**para rellenar el contenido de los atributos. Es conveniente hacer una **copia** de esos objetos y utilizar esas copias para los atributos pues si se utiliza la referencia que se ha pasado como parámetro.

```
public Rectangulo (Punto vertice1, Punto vertice2) {
     this.vertice1= new Punto (vertice1.getX(), vertice1.getY() );
     this.vertice2= new Punto (vertice2.getX(), vertice2.getY() );
}
```

▼ Clases anidadas o internas.

Se pueden distinguir varios tipos de **clases internas**:

- **Clases internas estáticas** (o **clases anidadas**), declaradas con el modificador `static`.
- **Clases internas miembro**, conocidas habitualmente como **clases internas**. Declaradas al máximo nivel de la clase contenedora y no estáticas.
- **Clases internas locales**, que se declaran en el interior de un bloque de código (normalmente dentro de un método).
- **Clases anónimas**, similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y, al no tener nombre, no tendrán constructores). Se suelen usar en la **gestión de eventos** en las **interfaces gráficas**.

```
class ClaseContenedora {
…
     static class ClaseAnidadaEstatica {
…
     }
     class ClaseInterna {
…
     }
```

▼ Herencia.

El concepto de herencia es algo bastante simple y sin embargo muy potente: cuando se desea definir una nueva clase y ya existen clases que, de alguna manera, implementan parte de la funcionalidad que se necesita, es posible crear una nueva clase derivada de la que ya tienes.(”es un”)

▼ Sintaxis de la herencia.

```
/**
 * Clase que representa a una persona
 */
public class Persona {
     private String nombre;
     private String apellidos;
     private LocalDate fechaNacimiento;
     …
}
/**
 * Clase que representa a un alumno
 */
public class Alumno extends Persona {
     private String grupo;
     private double notaMedia;
     …
}
```

▼ Acceso a miembros heredados.

Cuadro de niveles accesibilidad a los atributos de una clase

	Misma clase	Otra clase del mismo paquete	Subclase (aunque sea de diferente paquete)	Otra clase (no subclase) de diferente paquete
`public`	X	X	X	X
`protected`	X	X	X	
Sin modificador (paquete)	X	X		
`private`	X			

▼ Utilización de miembros heredados . Atributos.(I)

1. Utilización de miembros heredados . Métodos.(II)

▼ Redefinición de métodos heredados.

Aunque un método sea sobrescrito o redefinido, aún es posible **acceder** a él a través de la referencia **super**, aunque sólo se podrá acceder a métodos de la clase padre y no a métodos de clases superiores en la jerarquía de herencia.

Los métodos **estáticos** o de clase **no** pueden ser sobrescritos. Los originales de la clase base permanecen inalterables a través de toda la jerarquía de herencia.

```
@Override
public String getApellidos () {
     return “Alumno: “ + apellidos;
}
```

▼ Ampliación de métodos heredados.

super es similar a this, pero para la clase padre.

```
/**
 * Representación en forma de String del contenido del objeto Alumno
 * Aprovecha el método toString de la clase Persona mediante una llamada a
 * super.toString(). Es decir, se está ampliando la funcionalidad de la clase
 * Persona.
 * @return
*/
@Override
public String toString () {
    StringBuilder resultado;

    // Llamada al método “toString” de la superclase
    resultado= new StringBuilder (super.toString());

    // A continuación añadimos la información “especializada” de esta subclase
    resultado.append("\n");
    resultado.append ("Grupo: ").append(this.grupo).append("\n");
    resultado.append ("Nota media: ").append(String.format("%6.2f", this.notaMedia));

    return resultado.toString();
}
```

`super()` para invocar al constructor de la superclase

▼ Constructores y herencia.

La utilización del método **super**() para llamar al constructor de la clase padre sólo puede hacerse desde la **primera línea de código** de un constructor (con la única **excepción** de que exista antes una llamada a otro constructor de la clase mediante **this**()).

```
/**
 * Constructor de la clase Persona
 * @param nombre            Nombre de la persona
 * @param apellidos         Apellidos de la persona
 * @param fechaNacimiento   Fecha de nacimiento de la persona
 */
public Persona (String nombre, String apellidos, LocalDate fechaNacimiento) {
     this.nombre= nombre;
     this.apellidos= apellidos;
     this.fechaNacimiento= fechaNacimiento;
```

```
}
/**
 * Constructor de la clase Alumno
 * @param nombre            Nombre del alumno
 * @param apellidos         Apellidos del alumno
 * @param fechaNacimiento   Fecha de nacimiento del alumno
 * @param grupo             Grupo al que pertenece el alumno
 * @param notaMedia         Nota media del alumno
 */
public Alumno (String nombre, String apellidos, LocalDate fechaNacimiento, String grupo, double notaMedia) {
     super (nombre, apellidos, fechaNacimiento);
     this.grupo= grupo;
     this.notaMedia= notaMedia;
}
```

▼ Creación y utilización de clases derivadas.

▼ La clase `Object` en Java.

Método	Descripción
`Object ()`	Constructor.
`clone ()`	Método **clonador**: crea y devuelve una copia del objeto ("clona" el objeto).
`boolean equals (Object obj)`	Indica si el objeto pasado como parámetro es igual a este objeto.
`void finalize ()`	Método llamado por el **recolector de basura** cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
`int hashCode ()`	Devuelve un **código hash** para el objeto.
`toString ()`	Devuelve una representación del objeto en forma de `String`.

▼ Herencia múltiple.

En Java no existe la herencia múltiple de clases.

▼ Clases abstractas.

Clases que **nunca** serán **instanciadas**, sino que proporcionan un marco o modelo a seguir por sus clases derivadas dentro de una jerarquía de herencia.

1. Declaración de una clase abstracta.

```
/**
* Clase Persona * Clase que representa a una persona
*/
public abstract class Persona {
        protected String nombre;
        protected String apellidos;
        protected LocalDate fechaNacimiento;
        ...
}
```

2. Métodos abstractos.

   - Un **método abstracto** implica que la clase a la que pertenece tiene que ser **abstracta**, pero eso no significa que todos los métodos de esa clase tengan que ser abstractos.

- Un **método abstracto** no puede ser **privado** (no se podría implementar, dado que las **clases derivadas** no tendrían acceso a él).
- Los **métodos abstractos** no pueden ser **estáticos**, pues los **métodos estáticos** no pueden ser redefinidos (y los **métodos abstractos** necesitan ser redefinidos).

```
/**
 * Declaración del método abstracto saludar de la clase Persona.
 * Este método en realidad no es instanciable pues aquí no se implementa.
 * @return String que representaría el saludo de una Persona
 */
protected abstract String saludar ();
```

3. Clases y métodos finales.

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

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

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

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

▼ Interfaces.

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.

   Una clase puede adoptar distintos modelos de comportamiento establecidos en diferentes interfaces. Es decir una clase puede implementar varias interfaces.

   1. ¿Clase abstracta o interfaz?

      A partir de ahora podemos hablar de otra posible relación entre clases: la de compartir un determinado comportamiento (interfaz). Dos clases podrían tener en común un determinado conjunto de comportamientos sin que necesariamente exista una relación jerárquica entre ellas. Tan solo cuando haya realmente una relación de tipo "es un" se producirá herencia.

2. Definición de interfaces.

   Una **interfaz** consiste esencialmente en una lista de **atributos finales** constantes y **métodos abstractos** (sin implementar).

   - Puede utilizarse el modificador **public**. Si incluye este modificador la interfaz debe tener el mismo nombre que el archivo .java
   - Si **no** se indica el modificador public, el acceso será por omisión o **"de paquete"**
   - Todos los miembros de la interfaz **(atributos y métodos**) son **public** de manera implícita. **No** es necesario **indicar** el modificador public
   - 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.

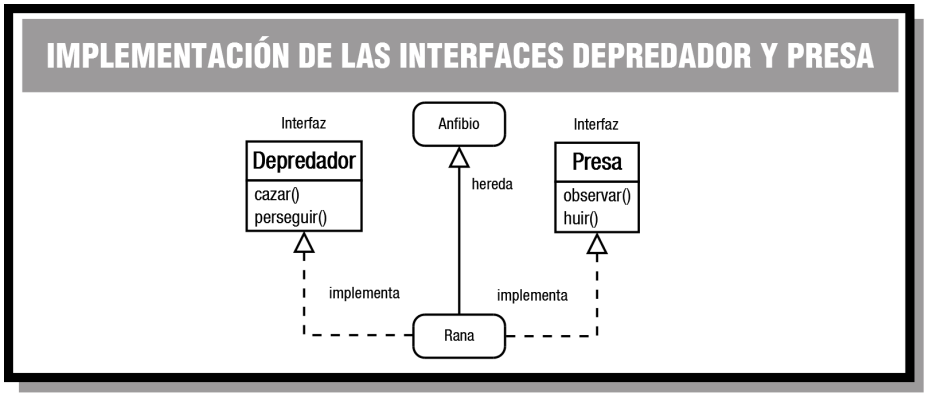
```
[public] interface <NombreInterfaz> {
                [public] [final] <tipo1> <atributo1>= <valor1>;
                [public] [final] <tipo2> <atributo2>= <valor2>;
                ...
                [public] [abstract] <tipo_devuelto1>  <nombreMetodo1> ([lista_parámetros]);
                [public]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

3. Implementación de interfaces.

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

    1. Un ejemplo de implementación de interfaces.

4. Simulación de la herencia múltiple mediante el uso de interfaces.



5. Herencia de interfaces.

   Las interfaces, al igual que las clases, también permiten la herencia. Para indicar que una interfaz hereda de otra se indica nuevamente con la palabra reservada extends. Pero en este caso sí se permite la herencia múltiple de interfaces. Si se hereda de más de una interfaz se indica con la lista de interfaces separadas por comas.

```
public interface InterfazUno {

                // Métodos y constantes de la interfaz Uno

}

public interface InterfazDos {

                // Métodos y constantes de la interfaz Dos

}
```

```
public interface InterfazCompleja extends InterfazUno, InterfazDos {

                // Métodos y constantes de la interfaz compleja

}
```

▼ Polimorfismo.

1. Concepto de polimorfismo.

   El *polimorfismo* ofrece la *posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una de sus subclase*s. Esto te va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus *superclases*.

   El *polimorfismo* puede llevarse a cabo tanto con *superclases* (abstractas o no) como con *interfaces.*

2. Ligadura dinámica.

```
public abstract class Instrumento {
    public void tocarNota (String nota) {
        System.out.printf ("Instrumento: tocar nota %s.\n", nota);
    }
}
```

```
public class Flauta extends Instrumento {
    @Override
    public void tocarNota (String nota) {
        System.out.printf ("Flauta: tocar nota %s.\n", nota);
    }
}

public class Piano extends Instrumento {
    @Override
    public void tocarNota (String nota) {
        System.out.printf ("Piano: tocar nota %s.\n", nota);
    }
}
```

A la hora de declarar una **referencia** a un objeto de tipo instrumento, utilizamos la **superclase** (`Instrumento`):

```
Instrumento instrumento1; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
```

Sin embargo, a la hora de instanciar el objeto, utilizamos el **constructor** de alguna de sus **subclases** (`Piano, Flauta`, etc.):

```
if (<condición>) {
   // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)
   instrumento1= new Piano ();
```

```
}
else if (<condición>) {
   // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)
   instrumento1=  new Flauta ();
} else {
   ...
}
```

Finalmente, a la hora de invocar el método tocarNota, no sabremos a qué versión (de qué subclase) de tocarNota se estará llamando, pues dependerá del tipo de objeto (subclase) que se haya instanciado. Se estará utilizando por tanto la ligadura dinámica:

```
// Interpretamos una nota con el objeto instrumento1
// No sabemos si se ejecutará el método tocarNota de Piano o de Flauta
// (dependerá de la ejecución)
instrumento1.tocarNota ("do");   // Ejemplo de ligadura dinámica (tiempo de ejecución)
```

3. Limitaciones de la ligadura dinámica.

   No se puede acceder a los *miembros específicos* de una *subclase* a través de una *referencia* a una superclase. Sólo se pueden utilizar los miembros declarados en la *superclase*, aunque la definición que finalmente se utilice en su ejecución sea la de la *subclase*.

4. Interfaces y polimorfismo.

   > Un objeto cuya referencia sea de tipo interfaz sólo puede utilizar aquellos métodos definidos en la interfaz, es decir, que no podrán utilizarse los atributos y métodos específicos de su clase, tan solo los de la interfaz.

   En el caso de las clases `Persona`, `Alumno` y `Profesor`, podrías declarar, por ejemplo, variables del tipo `Imprimible`:

   ```
   Imprimible objeto ;               // Imprimible es una interfaz y no una clase
   ```

   Con este tipo de referencia podrías luego apuntar a objetos tanto de tipo `Profesor` como de tipo `Alumno`, pues ambos implementan la interfaz `Imprimible`:

   ```
   // En algunas circunstancias podría suceder esto:
   objeto = new Alumno (nombre, apellidos, fecha, grupo, nota) ; // Polimorfismo con interfaces
   ...
   // En otras circunstancias podría suceder esto:
   objeto = new Profesor (nombre, apellidos, fecha, especialidad, salario) ; // Polimorfismo con interfaces
   ...
   ```

   Y más adelante hacer uso de la **ligadura dinámica**:

```
// Llamadas sólo a métodos de la interfaz
String contenido ;
contenido = objeto.devolverContenidoString() ;  // Ligadura dinámica con interfaces
```

5. Conversión de objetos.

> Si deseas tener acceso a todos los métodos y atributos específicos del objeto subclase tendrás que realizar una conversión explícita (casting) que convierta la referencia más general (superclase) en la del tipo específico del objeto (subclase).

Para que puedas realizar conversiones entre distintas clases es obligatorio que exista una relación de herencia entre ellas (una debe ser clase derivada de la otra)

La conversión en sentido contrario (de superclase a subclase) debe hacerse de forma explícita y según el caso podría dar lugar a errores por falta de información (atributos) o de métodos. En tales casos se produce una excepción de tipo ClassCastException.