MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN

Utilización avanzada de clases II. Clases abstractas, interfaces y polimorfismo

13

ÍNDICE

OBJETIVOS

Utilizar clases abstractas.

Utilizar métodos abstractos.

Utilizar interfaces.

Diferenciar entre interfaz y clase abstracta.

Utilizar polimorfismo.

Utilizar clases genéricas.

/ 1. Introducción y contextualización práctica

En este tema, hablaremos sobre las clases abstractas, qué son y cómo utilizarlas, además de los métodos abstractos.

También vamos a hablar sobre las interfaces, para qué sirven y qué las diferencia de las clases abstractas.

Seguidamente, veremos el concepto de polimorfismo y cómo podemos utilizarlo en nuestros programas, ya que tiene mucho potencial.

Por último, vamos a ver qué son las clases genéricas y la gran utilidad que estas pueden llegar a generar.

Todos son **conceptos complementarios a los estudiados en el tema anterior,** por lo que te resultarán muy familiares.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

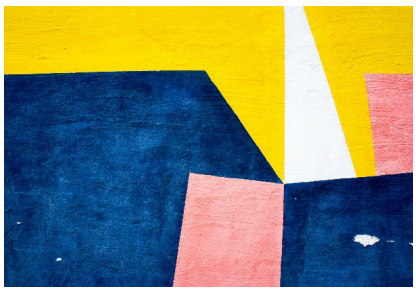

Fig. 1. Abstracción.

Audio intro. "¿Clases sin utilidad?"
https://bit.ly/33119xI

/ 2. Clases abstractas

Cuando ya tenemos cierto manejo de la utilización de la herencia, llegaremos a un punto en el que empezaremos a definir clases, que van a contener funcionalidades comunes de otras, de las que queremos que hereden, pero que no tendrá sentido instanciar un objeto de ellas.

Por ejemplo, podemos crear las clases *Mamífero y Reptil,* ambas con una serie de propiedades comunes a todos los mamíferos y reptiles, para luego crear las clases *Perro, Gato y Serpiente,* que heredarán de las anteriores. Ahora nos preguntamos, ¿declararemos objetos de las clases *Mamífero y Reptil,* o solo de las clases *Perro, Gato y Serpiente?* La respuesta está bastante clara, y es que solo vamos a necesitar objetos de *Perro, Gato y Serpiente,* ya que serán animales concretos, mientras que crear un mamífero o un reptil puede ser confuso, ya que no sabemos a qué tipo de animal se van a referir.

En este ejemplo, las clases *Mamífero y Reptil* van a ser clases abstractas, clases de las que no vamos a instanciar nunca un objeto, y que no tendría sentido.

Cuando declaramos una clase abstracta, estamos indicado que de esa clase no se van a poder instanciar objetos, pero sí que se va a poder heredar de ellas. Esto nos servirá para agrupar una serie de funcionalidades comunes.

Para declarar una clase abstracta en Java, lo hacemos mediante la palabra reservada abstract, delante del nombre de la clase en cuestión.

A partir de ese momento, no se podrán instanciar objetos de dicha clase, aunque sí se podrán implementar los constructores en una clase abstracta.

Se pueden declarar tantas clases abstractas como se necesite.

Una clase abstracta podrá heredar de otra clase sin inconveniente alguno.

```
/**
 * Clase que representa un mamífero. Esta clase es abstracta.
 * @author Francis
 */
public abstract class Mamifero
{
```

Código 1. Ejemplo de una clase abstracta.

/ 3. Métodos abstractos

Dentro de las clases abstractas, podremos declarar métodos como venimos haciendo, es decir, podrán devolver un valor, no devolverlo, tener una cierta cantidad de parámetros, etc., pero también vamos a poder declarar métodos abstractos.

Los métodos abstractos son métodos que cumplen las siguientes restricciones:

1. **No se implementan.** Esto quiere decir que no se van a programar sus cuerpos, ni aunque devuelvan un valor, simplemente, se declarará su cabecera y se terminará con punto y coma.

2. Este tipo de métodos solo pueden existir **dentro de una clase abstracta.**

3. Los métodos abstractos tendrán que ser **sobrescritos obligatoriamente** por las clases que hereden de ellas.

Esto nos va a servir para que cuando tengamos clases que hereden de la clase abstracta y tengan que implementar los métodos abstractos, cada una lo haga de una forma diferente, es decir, vamos a tener el mismo método base, pero con diferentes implementaciones.

Una clase abstracta puede no contener ningún método abstracto, solamente métodos 'normales', pero en el caso de que se necesite un método abstracto en una clase, hará que esta ya tenga que ser abstracta, forzosamente.

Para declarar un método abstracto, deberemos usar la palabra reservada *abstract* antes de indicar qué devuelve, de la siguiente forma:

[public/private] abstract [tipo / void] nombreMetodo([parámetros])

```java
/**
 * Clase que representa un mamífero. Esta clase es abstracta.
 * @author Francis
 */
public abstract class Mamifero
{
    /**
     * Método abstracto que indica como se desplaza
     */
    public abstract void desplazarse();
}
```

Código 2. Ejemplo de un método abstracto.

Audio 1. "Heredando de una clase abstracta"
https://bit.ly/3brHz1v

/ 4. Caso práctico 1: "¿Qué le ocurre a mi código?"

Planteamiento: Pilar y José están realizando una práctica en la que tienen que realizar un método que no tiene implementación, es decir, un método abstracto. Lo han definido dentro de la clase principal.

"¿Qué pasa aquí, por qué me está dando error este método?", le pregunta Pilar a José. José, igual de confundido que Pilar, no sabe qué ocurre: "Este método está bien definido, no entiendo nada", se dice José.

Nudo: ¿Están Pilar y José definiendo de forma correcta ese método? ¿En qué crees que están fallando?

Desenlace: Cuando empezamos a utilizar los métodos abstractos, es normal perderse un poco. Este tipo de métodos son algo especiales, ya que no tienen una implementación definida, al contrario de lo que estábamos acostumbrados hasta ahora.

Un error típico es el de querer implementar un método abstracto dentro de la clase principal de nuestro proyecto. Esto va a provocar un error, bastante lógico por otro lado, y es que ya hemos visto que si definidos un método abstracto, esa clase debe ser abstracta, obligatoriamente, y ¿la clase principal puede ser abstracta? La respuesta es un contundente **no.**

La clase principal es donde vamos a empezar a ejecutar nuestro proyecto, por lo que se debe ejecutar de manera obligatoria, mientras que las clases abstractas, al no poder instanciar objetos de ellas, no se ejecutarán nunca, así que aquí se encuentra el error que están cometiendo nuestros amigos, el de poner un método abstracto en la clase principal.

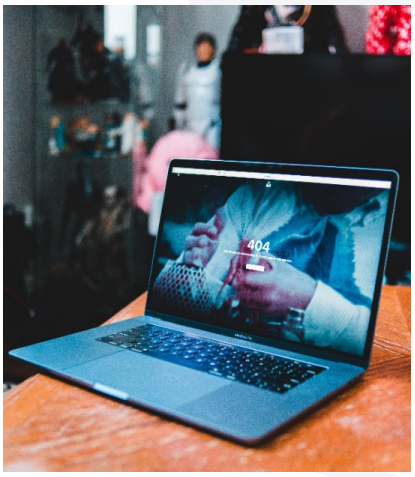

Fig. 2. Los errores hay que investigarlos.

/ 5. Polimorfismo

El polimorfismo es otro de los pilares de la programación dirigida a objetos y va ligado de forma muy estrecha a la herencia.

En Java, para poder utilizar el polimorfismo, debemos tener una serie de objetos que tengan una raíz común, es decir, que hereden de la misma clase padre.

El hecho de que una serie de objetos hereden de una misma clase padre va a traducirse en que van a ser compatibles, ya que, en definitiva, todos "son un objeto" de la misma clase padre.

Cuando utilizamos polimorfismo, vamos a poder enviar exactamente el mismo mensaje a varios objetos, ya que serán compatibles con él al heredar de la misma clase.

Existen dos tipos de polimorfismo:

- Polimorfismo **estático,** o en tiempo de compilación.
- Polimorfismo **dinámico,** o en tiempo de ejecución. También se le puede conocer con el nombre de ligadura dinámica.

Estudiaremos estos en los siguientes apartados en mayor profundidad.

Un ejemplo de uso de polimorfismo puede ser el siguiente:

1. **Definimos una clase** *Figura,* donde tendremos algunas propiedades de las figuras y un método para calcular el área. Este método lo definiremos abstracto (definiendo la clase *Figura* abstracta también) ya que cada figura lo implementará de una forma distinta.
2. **Definimos las clase**s *Cuadrado, Triangulo, Rombo y Circulo*. Todas estas clases heredarán de la clase *Figura,* y, por ello, deberán implementar el método para calcular el área, de una forma diferente cada una de ellas.

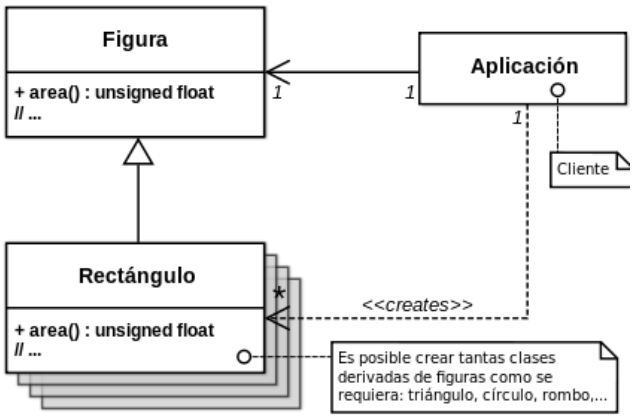

Fig. 3. Ejemplo de polimorfismo.
Fuente: https://es.wikipedia.org/wiki/Polimorfismo_(inform%C3%A1tica)

5.1. Polimorfismo estático o en tiempo de compilación

La palabra polimorfismo significa "que puede tener varias formas", y proviene del griego.

Cuando declaramos un objeto en Java, lo hacemos de la siguiente forma: *Tipo nombreobjeto = new Tipo( );*

Donde tenemos las siguientes partes:

1. El **tipo,** que es el nombre de la clase que va a instanciar nuestro objeto. Esta va a ser la parte estática.
2. El **nombre** de la clase a la que va a apuntar nuestro objeto en tiempo de ejecución. Esta va a ser la parte dinámica.

Cuando utilizamos polimorfismo estático, vamos a tener una jerarquía de clases, donde podrá haber clases normales y clases abstractas, y a su vez, clases que heredan de alguna de ellas.

La idea va a ser que tanto el tipo del objeto como el tipo de la clase con la que instanciamos nuestro objeto van a ser la misma, es decir, si el objeto es de tipo *Cuadrado,* en el *new* tendremos también Cuadrado.

Esto implica que la asignación de tipos se va a realizar en tiempo de compilación.

La consecuencia más notable de esta declaración de tipos en tiempo de compilación es que no podremos llamar a métodos de clases que no sean del mismo tipo (aunque hayan heredado de la misma clase padre), es decir, que sean compatibles.

Cuando usamos polimorfismo, el compilador no va a decidir en tiempo de compilación qué métodos se van a llamar, sino que se tomará esa decisión en tiempo de ejecución, con la llamada **vinculación dinámica**. Estas funciones polimórficas también se denominan **funciones virtuales.**

Fig. 4. Polimorfismo.

5.2. Polimorfismo dinámico o en tiempo de ejecución. Vinculación dinámica

Este tipo de polimorfismo es el que se suele usar normalmente.

Volvemos al mismo caso de antes, cuando declaramos un objeto en Java, lo hacemos de la siguiente forma: *Tipo nombreobjeto = new Tipo( );*

Donde tenemos las siguientes partes:

1. El **tipo,** que es el nombre de la clase que va a instanciar nuestro objeto. Esta va a ser la parte estática.

2. El **nombre** de la clase a la que va a apuntar nuestro objeto en tiempo de ejecución. Esta va a ser la parte dinámica.

Igualmente que en el caso anterior, cuando utilizamos polimorfismo dinámico, vamos a tener una jerarquía de clases, donde podrá haber clases normales y clases abstractas, y a su vez, clases que heredan de alguna de ellas.

No obstante, ahora, la idea va a ser que tanto el tipo del objeto como el tipo de la clase con la que instanciamos nuestro objeto **pueden no ser la misma,** es decir, si el objeto es de tipo *Figura,* en el new podremos tener Cuadrado.

En esta ocasión, esto implica que la asignación de tipos se va a realizar en tiempo de **ejecución.**

En este tipo de polimorfismo, la consecuencia más significativa de esta declaración de tipos en tiempo de ejecución es que sí podremos llamar a métodos de clases que no sean del mismo tipo, siempre que hayan heredado de la misma clase padre, es decir, que sean compatibles.

Es muy importante diferenciar que es el compilador el que verifica que los tipos sean correctos, pero cuando hacemos un cambio de tipo dinámico lo hace la propia máquina virtual de Java, consiguiendo así que los tipos se puedan cambiar de forma dinámica en la ejecución del programa.

Fig. 5. Varias herramientas compatibles.

5.3. Comprobación de tipos. La sentencia instaceof

Cuando utilizamos polimorfismo, vamos a tener varias clases que heredan de una misma clase padre, pero con diferente funcionalidad en sus métodos.

Pongamos un ejemplo, imaginemos que tenemos la clase *Animal*, la cual es abstracta y tiene un método *desplazarse,* abstracto también. Declaramos también las clases *Perro, Gato y Serpiente,* todas heredando de la clase *Animal,* e implementamos el método desplazarse indicando que un perro anda, un gato camina y una serpiente repta.

Imaginemos que declaramos varios objetos de las clases *Perro, Gato y Serpiente,* teniendo cada uno sus propios datos.

Ahora, vamos a aplicar una de las mayores ventajas del polimorfismo. Si bien, en la unidad de arrays, vimos que en un array solo podíamos tener datos del mismo tipo, ¿cómo podemos aprovechar el polimorfismo? Pues bien, si declaramos un array de objetos de tipo *Perro,* solo podremos almacenar objetos de ese tipo, pero ¿qué ocurre si creamos un array de objetos de tipo *Animal?*

El hecho de que la clase *Animal* sea abstracta no quiere decir que no se puedan crear arrays que almacenen ese tipo, es más, en un array de datos de tipo *Animal* podremos guardar objetos de cualquiera de las clases que hereden de ella, es decir, podremos guardar en un array de tipo *Animal* objetos de tipo *Perro, Gato y Serpiente.*

Sin embargo, ahora tenemos un problema. Cuando obtengamos un dato de ese array, será un objeto de tipo *Animal,* ¿y si necesitamos llamar a algún método específico de las clases derivadas? Esto lo podremos solucionar con la instrucción *instanceof,* mediante la cual podremos saber de qué tipo es un objeto.

Su uso es el siguiente: *if (nombreobjeto instanceof NombreClase)*

En el caso de que el objeto sea del tipo de la clase que indica la expresión, tomará el valor de verdadero, mientras que si no, valdrá falso.

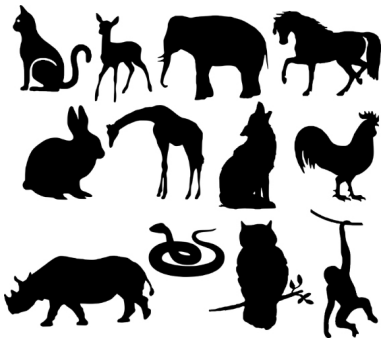

Fig. 6. Comprobación de tipos.

Vídeo 1. "Ejemplo de uso de instanceof"
https://bit.ly/3br2ba5

5.4. Casting entre objetos

Como ya vimos en unidades pasadas, los castings son conversiones de tipo entre variables en los que podría producirse algún tipo de pérdida de información. También vimos que se podían realizar esos *castings* o conversiones entre variables de tipo primitivo, siempre y cuando estas fuesen compatibles.

A continuación, vamos a ver que entre objetos también se van a poder realizar *castings,* y para ello, necesitamos utilizar la herencia y el polimorfismo.

Al igual que en las variables de tipos primitivos, los *castings* entre objetos se van a poder realizar cuando las clases sean compatibles, o lo que es lo mismo, **cuando estemos trabajando con una jerarquía de clases.**

El mayor partido lo vamos a sacar cuando queramos objetos de un tipo en concreto, ya que, si hacemos el *casting* a la clase padre, no vamos a poder utilizar los métodos de las clases hijas.

Imaginemos que tenemos la misma jerarquía de clases que en el punto 5.3. Es decir, tenemos una clase *Animal* y tres clases hijas, *Perro, Gato y Serpiente*.

Volvemos al caso de crear un array de objetos de tipo *Animal* e introducimos varios objetos de tipo *Perro, Gato y Serpiente*. Al recorrer ese array, mediante la instrucción *instanceof,* podremos saber cuándo tenemos un objeto de cada uno de los tipos, pero ¿qué ocurre si queremos invocar a un método de una de las clases hijas? Pues como tenemos objetos compatibles entre sí, podremos hacer un *casting* a la clase que nos interese.

Al igual que con las variables primitivas, pondremos entre paréntesis el nombre de la clase a la que queremos hacer el *casting*.

```
for(Animal ani : animales)
{
    if( ani instanceof Perro )
    {
        Perro p = (Perro)ani;
        p.metodoPerro();
    }
    if( ani instanceof Gato )
    {
        Gato p = (Gato)ani;
        p.metodoGato();
    }
    if( ani instanceof Serpiente )
    {
        Serpiente p = (Serpiente)ani;
        p.metodoSerpiente();
    }
}
```

Código 3. Ejemplo de casting entre objetos.

/ 6. Interfaces

Una interfaz es **una serie de acciones relacionadas entre sí, que va a poder ejecutar un objeto.**

Concretamente, una interfaz es una clase que únicamente puede tener **variables constantes y métodos sin implementación, pero que no son abstractos.**

Básicamente, en una interfaz, únicamente existen los prototipos de las funcionalidades, es decir, de los métodos, que más tarde se agregarán a otra clase.

Las interfaces proporcionan las siguientes ventajas:

- Organización y estructuración de código.
- Las constantes que se creen en una interfaz estarán disponibles para todas aquellas clases que la usen.
- Se obliga a que las clases que la utilizan usen el mismo método.
- Permiten establecer una relación entre clases que aparentemente no están relacionadas.

Para declarar una interfaz en Java, utilizamos la palabra reservada Interface:

[public / private] Interface NombreInterfaz

El cuerpo de la interfaz irá entre llaves ({ }) y en él se declararán las variables constantes y métodos que necesitemos.

Para poder usar una interfaz, tendremos que "implementarla", esto se hace mediante la palabra reservada *implements*.

```
[public / private] class NombreClase implements NombreInterfaz
```

Al contrario que con la herencia, que únicamente se puede heredar de una clase, con las interfaces, podremos implementar todas las que necesitemos, aunque habrá que implementar los métodos de todas ellas. Para ello, separaremos las interfaces a implementar mediante comas.

```
[public / private] class NombreClase implements NombreInterfaz1, NombreInterfaz2, NombreInterfaz3
```

```
for(Animal ani : animales)
{
    if( ani instanceof Perro )
    {
        Perro p = (Perro)ani;
        p.metodoPerro();
    }
    if( ani instanceof Gato )
    {
        Gato p = (Gato)ani;
        p.metodoGato();
    }
    if( ani instanceof Serpiente )
    {
        Serpiente p = (Serpiente)ani;
        p.metodoSerpiente();
    }
}
```

Código 4. Ejemplo de interfaz.

6.1. Clase abstracta vs interfaz

Cuando hablamos de interfaces, se pueden confundir con las clases abstractas, ya que, vista la definición de una interfaz, estas y las clases abstractas parecen iguales.

Este error es muy común cuando se está aprendiendo el lenguaje, ya que prácticamente parecen lo mismo, salvo por leves diferencias.

Una interfaz se diferencia de una clase abstracta en:

- Todos los métodos que tenga una **interfaz** se van a declarar de forma automática e implícita como **abstractos y públicos.**
- En las clases **abstractas,** podemos implementar **métodos no abstractos.**
- La **interfaz no pertenece a la jerarquía de clases de la herencia.**
- Se podría considerar que la principal diferencia entre una clase abstracta y una **interfaz** es que esta última **no fuerza al programador a usar la herencia** para poder usarla, pudiendo implementar todas las interfaces que se necesite sin ningún tipo de restricción.

Puede haber ocasiones en las que el uso de una interfaz o de una clase abstracta dé **el mismo resultado.**

Una de las grandes **ventajas** que nos va a proporcionar el uso de las **interfaces** es que **una misma clase podrá heredar de otra y a su vez implementar una serie de interfaces:**

```
class NombreClase extends ClasePadre implements NombreInterfaz1, NombreInterfaz2
```

Cabe destacar que, si solamente se quiere **crear una lista de métodos abstractos,** suele ser más recomendable el uso de una **interfaz** sobre una clase abstracta.

Fig. 7. Clase abstracta vs interfaz.

/ 7. Caso práctico 2: "Emulando la herencia múltiple"

Planteamiento: Pilar y José están trabajando en un proyecto donde la herencia está muy presente. Tienen que realizar varias clases donde algunas heredan de otras, pero se han encontrado con un problema, hay ciertas clases que necesitan tener una serie de funcionalidades comunes, aunque con diferente implementación.

"No puedo hacer que vuelvan a heredar", le dice José a Pilar.

Pilar le responde que ella tampoco sabe cómo hacerlo, ya que únicamente se le viene a la mente la herencia.

Nudo: ¿Qué piensas al respecto? ¿Crees que hay alguna solución para esto? Pilar y José ¿deberán volver a repetir todo el código de las funciones comunes?

Desenlace: Este es uno de problemas más comunes que nos vamos a encontrar cuando trabajamos con herencia: tenemos varias clases que ya han heredado, pero necesitamos que esas clases tengan una funcionalidad común, y como ya han heredado, y en Java no se permite la herencia múltiple, no podremos hacer una clase en la que esté dicha funcionalidad, declararla como abstracta, y hacer que hereden de ella.

Fig. 8. Analizando el problema.

La solución a este problema son las interfaces. Como ya hemos comentado, en una interfaz, se pueden declarar todas las funciones que necesitemos, además, estarán sin implementar, y más tarde, hacer que una clase implemente dicha interfaz, teniendo que implementar forzosamente dichos métodos. Este comportamiento nos va a salvar del hecho de que Java no pueda implementar una herencia múltiple.

/ 8. Tipos genéricos. Clases genéricas

En Java, existe lo que se llaman tipos genéricos o tipos parametrizados. Este tipo de datos tiene gran relevancia, pudiendo llegar a ser muy útil, ya que nos va a permitir **crear funcionalidades donde podamos elegir el tipo de dato con el que se va a operar.**

Concretamente, nos van a permitir **crear plantillas de funcionalidades donde el tipo de dato con el que operen vendrá dado por parámetro.**

Quizás, esta sea su mayor ventaja: van a poder **trabajar directamente con el tipo que se le pase por parámetro,** sin necesidad de hacer *castings* ni que los tipos sean compatibles entre sí.

En el uso de los genéricos, podremos utilizar solamente objetos, es decir, no podremos usar tipos de datos primitivos para aplicar en un genérico.

En el caso de que queramos usar un tipo de **dato primitivo,** tendremos que usar **su equivalente,** como se indica en la siguiente tabla.

TIPO DE DATO PRIMITIVO	EQUIVALENTE
int	Integer
double	Double
boolean	Boolean

Tabla 1. Equivalencias de tipos.

Para declarar un dato genérico, usamos la notación <T>, donde T será el tipo de dato. Para crear una clase genérica, utilizaremos la siguiente sintaxis:

```
[public / private] class NombreClase<T>
```

Podremos crear tantos tipos como queramos:

```
[public / private] class NombreClase<T1, T2>
```

A partir de este momento, podremos utilizar T1 y T2 como tipos válidos para declarar variables dentro de la clase.

```
public class ClaseGenerica<T> {

}
```

Código 5. Clase genérica.

8.1. Métodos genéricos

Además de aplicarlo a las clases, los tipos genéricos se pueden aplicar a los métodos y podremos crear un método genérico **donde el algoritmo sea el mismo para cualquier tipo de dato.**

Para poder crear un método genérico, la clase donde este va a estar no tiene por qué ser genérica y tener los mismos tipos genéricos que va a usar el método.

Veamos un ejemplo. Imaginemos que queremos hacer un método que intercambie dos datos de tipo *int:*

```
public void intercambiar(int n1, int n2) {
    int auxiliar = n1;
    n1 = n2;
    n2 = auxiliar;
}
```

Código 6. Método intercambiar.

¿Qué ocurrirá si necesitamos intercambiar dos datos de tipo *String?* ¿O dos datos de tipo *Persona?*

El código para intercambiar cualquier tipo de dato siempre va a ser el mismo, por lo que podremos hacer un método genérico que intercambie dos variables.

```
public class ClaseGenerica<T> {

    public void intercambiar(T n1, T n2) {
        T auxiliar = n1;
        n1 = n2;
        n2 = auxiliar;
    }
}
```

Código 7. Método de intercambiar genérico.

A partir de este momento, podremos intercambiar cualquier tipo de dato.

Vídeo 2. "Creación y uso de usa clase genérica"
https://bit.ly/2DsFLc2

/ 9. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad, hemos podido aprender el concepto de **clase y método abstracto** y cuándo vamos a poder usar cada uno de ellos. También, hemos profundizado sobre el último de los pilares de la programación dirigida a objetos que nos faltaba por conocer, el **polimorfismo.** Hemos visto en qué consiste, cómo podemos usarlo y las ventajas que ello conlleva.

Hemos estudiado, además, la instrucción ***instanceof,*** que nos permitía saber de qué tipo es un objeto que ya tenemos instanciado, y que va de la mano del polimorfismo.

A continuación, comprobamos que, al igual que con las variables de tipo primitivo, con los objetos también podemos realizar ***castings*** entre ellos, siempre y cuando sean compatibles.

Por otro lado, hemos trabajado con las **interfaces,** cómo podemos utilizarlas, sus ventajas y en qué se diferencian de las clases abstractas, ya que no hay que confundirlas.

Por último, hemos visto los **tipos de datos genéricos,** que nos permiten crear tanto clases como métodos sin especificar los datos para, más tarde, poder usarlos con los tipos de datos que necesitemos.

Resolución del caso práctico inicial

Cuando ya tenemos cierto manejo sobre la herencia y la sobrecarga de métodos, vamos a notar que, tarde o temprano, tendremos clases en las que solo tenemos métodos, y de las que van a heredar otras que sobrecargaran dichos métodos.

Fig. 9. La abstracción en los programas es muy importante.

Siguiendo con el caso planteado, de esas clases que comenta Pilar, no vamos a necesitar instanciar un objeto, ya que las que dan una función a dichos métodos son las clases que heredan.

Esto es totalmente normal, y es una buena señal, ya que indica que nuestro dominio de la abstracción mediante herencia es bueno. En este caso, como ya hemos visto a lo largo del tema, estamos hablando de las clases abstractas.

/ 10. Bibliografía

Rodríguez, A. (s. f.). *Sobreescribir métodos en Java. Tipo estático y dinámico. Ligadura. Métodos polimórficos. Ejercicio (CU00690B)*. Recuperado 25 de mayo de 2020, de https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=660:sobreescribir-metodos-en-java-tipo-estatico-y-dinamico-ligadura-metodos-polimorficos-ejercicio-cu00690b&catid=68&Itemid=188

Colaboradores de Wikipedia. (2020, mayo 7). *Polimorfismo (informática)*. Wikipedia, la enciclopedia libre. Recuperado 25 de mayo de 2020, de https://es.wikipedia.org/wiki/Polimorfismo_(inform%C3%A1tica)

Casting de objetos en Java | Disco Duro de Roer. (2017, octubre 9). Recuperado 25 de mayo de 2020, de https://www.discoduroderoer.es/casting-de-objetos-en-java/

Colaboradores de Wikipedia. (2020, mayo 13). *Interfaz (Java)*. Wikipedia, la enciclopedia libre. Recuperado 25 de mayo de 2020, de https://es.wikipedia.org/wiki/Interfaz_(Java)

E. (s. f.). 4.5 *Interfaces | Curso de Introducción a Java*. Recuperado 25 de mayo de 2020, de https://www.mundojava.net/interfaces.html?Pg=java_inicial_4_5.html