**UNIDAD 6: Utilización avanzada de clases**

**1.- Relaciones entre clases**

En algunos casos es posible que no exista relación alguna entre unas clases y otras, pero lo más habitual es que sí exista: una clase puede ser una [**especialización**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#tba24b69a-5374-7de6-b0fb-87bfc41a4d38) de otra, o bien una [**generalización**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#t3f6ed979-11db-98b3-9f98-c5d09392ff87), o una clase contiene en su interior objetos de otra, o una clase utiliza a otra, etc. Es decir, que entre unas clases y otras habrá que definir cuál es su relación (si es que existe alguna).

Se pueden distinguir diversos tipos de relaciones entre clases:

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

La relación de **clientela** la llevas utilizando desde que has empezado a programar en Java, pues desde tu clase principal (clase con método **main**) has estado declarando, creando y utilizando objetos de otras clases. Por ejemplo: si utilizas un objeto **String** dentro de la clase principal de tu programa, éste será cliente de la clase **String** (como sucederá con prácticamente cualquier programa que se escriba en Java). Es la relación fundamental y más habitual entre clases (la utilización de unas clases por parte de otras) y, por supuesto, la que más vas a utilizar tú también. De hecho ya la has estado utilizando y lo seguirás haciendo.

La relación de **composición** es posible que ya la hayas tenido en cuenta si has definido clases que contenían (tenían como atributos) otros objetos en su interior, lo cual es bastante habitual. Por ejemplo, si implementas una clase donde alguno de sus atributos es un objeto de tipo **String**, ya se está produciendo una relación de tipo **composición** (tu clase “tiene” un **String**, es decir, está compuesta por un objeto **String** y por algunos elementos más).

La relación de **anidamiento** (o **anidación**) es quizá menos habitual, pues implica declarar unas clases dentro de otras (**clases internas** o **anidadas**). En algunos casos puede resultar útil para tener un nivel más de [**encapsulamiento**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#td9561cea-2f16-ad0b-3c1d-ea34ebf05d94) y [**ocultación**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#tf08b236b-902a-debb-e720-c5695a80483a) de información. Un ejemplo típico de anidamiento es el de las **clases anónimas**, que suelen utilizarse en contextos donde hay que definir **manejadores de eventos** como puede ser el caso de las **interfaces gráficas de usuario**.

En el caso de la relación de **herencia** se trata de unas clases que derivan de otras. Un ejemplo en el que se produce habitualmente es en el caso de los objetos que forman parte de las **interfaces gráficas**, donde un componente hereda propiedades de sus ascendientes. Más delante lo verás al declarar componentes gráficos que hereden de algún otro componente (**JFrame**, **JDialog**, etc.).

Podría decirse que tanto la **composición** como la **anidación** son casos particulares de **clientela**, pues en realidad en todos esos casos una clase está haciendo uso de otra (al contener atributos que son objetos de la otra clase, al definir clases dentro de otras clases, al utilizar objetos en el paso de parámetros, al declarar variables locales utilizando otras clases, etc.).

A lo largo de la unidad, irás viendo distintas posibilidades de implementación de clases haciendo uso de todas estas relaciones, centrándonos especialmente en el caso de la **herencia**, que es la que permite establecer las relaciones más complejas.

Final del formulario

**1.1.- Composición**

Cuando en un sistema de información, una determinada entidad A contiene a otra B como una de sus partes, se suele decir que se está produciendo una relación de **composición**. Es decir, el objeto de la clase A contiene a uno o varios objetos de la clase B.

La **composición** puede encadenarse todas las veces que sea necesario hasta llegar a objetos básicos del lenguaje o hasta tipos primitivos que ya no contendrán otros objetos en su interior. Ésta es la forma más habitual de definir clases: mediante otras clases ya definidas anteriormente. Es una manera eficiente y sencilla de gestionar la reutilización de todo el código ya escrito. Si se definen clases que describen entidades distinguibles y con funciones claramente definidas, podrán utilizarse cada vez que haya que representar objetos similares dentro de otras clases.

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

Una forma sencilla de plantearte si la relación que existe entre dos clases A y B es de **composición** podría ser mediante la expresión idiomática “**tiene un**”: “la clase A tiene uno o varios objetos de la clase B”, o visto de otro modo: “Objetos de la clase B pueden formar parte de la clase A”.

Algunos ejemplos de composición podrían ser:

* Un **coche** tiene un **motor** y tiene cuatro **ruedas**.
* Una **persona** tiene un **nombre**, una **fecha de nacimiento**, una **cuenta bancaria** asociada para ingresar la nómina, etc.
* Un **cocodrilo** bajo investigación científica que tiene un número de **dientes** determinado, una **edad**, unas **coordenadas** de ubicación geográfica (medidas con GPS), etc.
* Una clase **Rectangulo** podría contener en su interior dos objetos de la clase **Punto** para almacenar los vértices inferior izquierdo y superior derecho.
* Una clase **Empleado** podría contener en su interior un objeto de la clase **<acronym title="Documento Nacional de Identidad.">DNI</acronym>** para almacenar su DNI/NIF, y otro objeto de la clase **CuentaBancaria** para guardar la cuenta en la que se realizan los ingresos en nómina.
* Una clase **JFrame** (**javax.Swing.JFrame**) de la **interfaz gráfica** contiene en su interior referencias a objetos de las clases **JRootPane**, **JMenuBar** o **JLayeredPane**, pues contiene menús, paneles, etc.

Principio del formulario

Final del formulario

**1.2.- Herencia**

El mecanismo que permite crear clases basándose en otras que ya existen es conocido como **herencia**. Como ya has visto en unidades anteriores, Java implementa la herencia mediante la utilización de la palabra reservada **extends**.

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. Al hacer esto se posibilita la reutilización de todos los atributos y métodos de la clase que se ha utilizado como **base** (**clase padre** o **superclase**), sin la necesidad de tener que escribirlos de nuevo.

Una **subclase** hereda todos los miembros de su **clase padre** (atributos, métodos y clases internas). Los **constructores** no se heredan, aunque se pueden invocar desde la **subclase**.

Algunos ejemplos de herencia podrían ser:

* Un **coche** es un **vehículo** (heredará atributos como la **velocidad máxima** o métodos como **parar** y **arrancar**).
* Un **empleado** es una **persona** (heredará atributos como el **nombre** o la **fecha de nacimiento**).
* Un **rectángulo** es una **figura geométrica** en el plano (heredará métodos como el cálculo de la **superficie** o de su **perímetro**).

En este caso la expresión idiomática que puedes usar para plantearte si el tipo de relación entre dos clases A y B es de herencia podría ser “**es un**”: “la clase A es un tipo específico de la clase B” (**especialización**), o visto de otro modo: “la clase B es un caso general de la clase A” (**generalización**).

Otros ejemplos de herencia, relacionados con las interfaces gráficas, que veremos más adelante en otra unidad, podrían ser:

* Una **ventana** en una **aplicación gráfica** puede ser una clase que herede de **JFrame** (componente **Swing**: **javax.swing.JFrame**), de esta manera esa clase será un marco que dispondrá de todos los métodos y atributos de **JFrame** más aquéllos que tú decidas incorporarle al rellenarlo de componentes gráficos.
* Una **caja de diálogo** puede ser un tipo de **JDialog** (otro componente **Swing: <strong>javax.swing.JDialog</strong>**).

En Java, la clase **Object** (dentro del paquete **java.lang**) define e implementa el comportamiento común a todas las clases (incluidas aquellas que tú escribas). Como recordarás, ya se dijo que en Java cualquier clase deriva en última instancia de la clase **Object**.

Todas las clases tienen una **clase padre**, que a su vez también posee una **superclase**, y así sucesivamente hasta llegar a la clase **Object**. De esta manera, se construye lo que habitualmente se conoce como una **jerarquía de clases**, que en el caso de Java tendría a la clase **Object** en la raíz.

Final del formulario

**1.3.- ¿Herencia o composición?**

Cuando escribas tus propias clases, debes intentar tener claro en qué casos utilizar la **composición** y cuándo la **herencia**:

* **Composición**: cuando una clase está formada por objetos de otras clases. En estos casos se incluyen objetos de esas clases, pero no necesariamente se comparten características con ellos (no se heredan características de esos objetos, sino que directamente se utilizarán sus atributos y sus métodos). Esos objetos incluidos no son más que atributos miembros de la clase que se está definiendo.
* **Herencia**: cuando una clase cumple todas las características de otra. En estos casos la clase derivada es una **especialización** (o **particularización**, **extensión** o **restricción**) de la clase base. Desde otro punto de vista se diría que la clase base es una **generalización** de las clases derivadas.

Por ejemplo, imagina que dispones de una clase **Punto** (ya la has utilizado en otras ocasiones) y decides definir una nueva clase llamada **Círculo**. Dado que un punto tiene como atributos sus coordenadas en plano (**x1**, **y1**), decides que es buena idea aprovechar esa información e incorporarla en la clase **Circulo** que estás escribiendo. Para ello utilizas la **herencia**, de manera que al derivar la clase **Círculo** de la clase **Punto**, tendrás disponibles los atributos **x1** e **y1**. Ahora solo faltaría añadirle algunos atributos y métodos más como por ejemplo el **radio** del círculo, el cálculo de su **área** y su **perímetro**, etc.

En principio parece que la idea pueda funcionar pero es posible que más adelante, si continúas construyendo una **jerarquía de clases**, observes que puedas llegar a conclusiones incongruentes al suponer que un círculo es una especialización de un punto (un tipo de punto). ¿Todas aquellas figuras que contengan uno o varios puntos deberían ser tipos de punto? ¿Y si tienes varios puntos? ¿Cómo accedes a ellos? ¿Un rectángulo también tiene sentido que herede de un punto? No parece muy buena idea.

Parece que en este caso habría resultado mejor establecer una relación de **composición**. Analízalo detenidamente: ¿cuál de estas dos situaciones te suena mejor?

1. “**Un círculo es un punto** (su centro)”, y por tanto heredará las coordenadas **x1** e **y1** que tiene todo punto. Además tendrá otras características específicas como el **radio** o métodos como el cálculo de la **longitud** de su perímetro o el cálculo de su **área**.
2. “**Un círculo tiene un punto** (su centro)”, junto con algunos atributos más como por ejemplo el **radio**. También tendrá métodos para el cálculo de su **área** o de la longitud de su **perímetro**.

Parece que en este caso la **composición** refleja con mayor fidelidad la relación que existe entre ambas clases. Normalmente suele ser suficiente con plantearse las preguntas “**¿A es un tipo de B?**” o“ ¿**A contiene elementos de tipo B?**”.

**2.- Composición.**

**2.1.- 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 <nombreClase> {

[modificadores] <NombreClase1> nombreAtributo1;

[modificadores] <NombreClase2> nombreAtributo2;

…

}

En unidades anteriores has trabajado con la clase **Punto**, que definía las coordenadas de un punto en el plano, y con la clase **Rectangulo,**que definía una figura de tipo rectángulo también en el plano a partir de dos de sus **vértices** (**inferior izquierdo** y **superior derecho**). Tal y como hemos formalizado ahora los tipos de relaciones entre clases, parece bastante claro que aquí tendrías un caso de **composición**: “**un rectángulo contiene puntos**”. Por tanto, podrías ahora redefinir los atributos de la clase **Rectangulo**(cuatro **números reales**) como dos objetos de tipo **Punto**:

class Rectangulo {

private Punto vertice1;

private Punto vertice2;

…

}

Ahora los métodos de esta clase deberán tener en cuenta que ya no hay cuatro atributos de tipo **double**, sino dos atributos de tipo **Punto** (cada uno de los cuales contendrá en su interior dos atributos de tipo **double**).

Final del formulario

Principio del formulario

Final del formulario

**2.2.- Uso de la composición (I). Preservación de la ocultación.**

Como ya has observado, la relación de **composición** no tiene más misterio a la hora de implementarse que simplemente declarar **atributos** de las clases que necesites dentro de la clase que estés diseñando.

Ahora bien, cuando escribas clases que contienen objetos de otras clases (lo cual será lo más habitual) deberás tener un poco de precaución con aquellos métodos que devuelvan información acerca de los **atributos** de la clase (métodos “**obtenedores**” o de tipo **get**).

Como ya viste en la unidad dedicada a la creación de clases, lo normal suele ser declarar los **atributos** como **privados** (o **protegidos**, como veremos un poco más adelante) para ocultarlos a los posibles **clientes** de la clase (otros objetos que en el futuro harán uso de la clase). Para que otros objetos puedan acceder a la información contenida en los **atributos,**o al menos a una parte de ella, deberán hacerlo a través de **métodos que sirvan de interfaz**, de manera que sólo se podrá tener acceso a aquella información que el creador de la clase haya considerado oportuna. Del mismo modo, los **atributos** solamente serán modificados desde los métodos de la clase, que decidirán cómo y bajo qué circunstancias deben realizarse esas modificaciones. Con esa metodología de acceso se tenía perfectamente separada la parte de manipulación interna de los atributos de la **interfaz** con el exterior.

Hasta ahora los métodos de tipo **get**devolvían **tipos primitivos**, es decir, copias del contenido (a veces con algún tipo de modificación o de formato) que había almacenado en los **atributos**, pero los atributos seguían “a salvo” como elementos privados de la clase. Pero, a partir de este momento, al tener objetos dentro de las clases y no sólo tipos primitivos, es posible que en un determinado momento interese devolver un **objeto completo**.

Ahora bien, 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**!

Para evitar ese tipo de situaciones (ofrecer al exterior referencias a objetos privados) puedes optar por diversas alternativas, procurando siempre evitar la devolución directa de un atributo que sea un objeto:

* Una opción podría ser devolver siempre tipos primitivos.
* Dado que esto no siempre es posible, o como mínimo poco práctico, otra posibilidad es crear un nuevo objeto que sea una copia del atributo que quieres devolver y utilizar ese objeto como valor de retorno. Es decir, **crear una copia del objeto** especialmente para devolverlo. De esta manera, el código cliente de ese método podrá manipular a su antojo ese nuevo objeto, pues no será una referencia al atributo original, sino un nuevo objeto con el mismo contenido.

Por último, debes tener en cuenta que es posible que en algunos casos sí se necesite realmente la referencia al atributo original (algo muy habitual en el caso de atributos estáticos). En tales casos, no habrá problema en devolver directamente el atributo para que el código llamante (cliente) haga el uso que estime oportuno de él.

**Debes evitar por todos los medios la devolución de un atributo que sea un objeto (estarías dando directamente una referencia al atributo, visible y manipulable desde fuera), salvo que se trate de un caso en el que deba ser así.**

Principio del formulario

Final del formulario

**2.2.1.- Uso de la composición (II). Llamadas a constructores.**

Otro factor que debes considerar, a la hora de escribir clases que contengan como atributos objetos de otras clases, es su comportamiento a la hora de instanciarse. Durante el proceso de creación de un objeto (**constructor**) de la clase contenedora habrá que tener en cuenta también la creación (llamadas a **constructores**) de aquellos objetos que son contenidos.

**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, el código cliente de la clase podría tener acceso a ella sin necesidad de pasar por la interfaz de la clase (volveríamos a dejar abierta una **puerta pública** a algo que quizá sea privado).

Además, si el **objeto parámetro** que se pasó al **constructor** formaba parte de otro objeto, esto podría ocasionar un desagradable efecto colateral si esos objetos son modificados en el futuro desde el código cliente de la clase, ya que no sabes de dónde provienen esos objetos, si fueron creados especialmente para ser usados por el nuevo objeto creado o si pertenecen a otro objeto que podría modificarlos más tarde. Es decir, correrías el riesgo de estar “compartiendo” esos objetos con otras partes del código, sin ningún tipo de control de acceso y con las nefastas consecuencias que eso podría tener: cualquier cambio de ese objeto afectaría a partes del programa supuestamente independientes, que entienden ese objeto como suyo.

**En el fondo los objetos no son más que variables de tipo referencia a la zona de memoria en la que se encuentra toda la información del objeto en sí mismo. Esto es, puedes tener un único objeto y múltiples referencias a él. Pero sólo se trata de un objeto, y cualquier modificación desde una de sus referencias afectaría a todas las demás, pues estamos hablando del mismo objeto.**

Recuerda también que sólo se crean objetos cuando se llama a un constructor (uso de **new**). Si realizas asignaciones o pasos de parámetros, no se están copiando o pasando copias de los objetos, sino simplemente de las referencias, y por tanto se tratará siempre del mismo objeto.

Se trata de un efecto similar al que sucedía en los métodos de tipo **get**, pero en este caso en sentido contrario (en lugar de que nuestra clase “regale” al exterior uno de sus atributos objeto mediante una referencia, en esta ocasión se “adueña” de un parámetro objeto que probablemente pertenezca a otro objeto y que es posible que en el futuro haga uso de él).

Para entender mejor estos posibles efectos podemos continuar con el ejemplo de la clase **Rectangulo** que contiene en su interior dos objetos de la clase **Punto**. En los constructores del rectángulo habrá que incluir todo lo necesario para crear dos instancias de la clase **Punto** evitando las referencias a parámetros (haciendo copias).

Principio del formulario

Final del formulario

**2.3.- Clases anidadas o internas**

En algunos lenguajes, es posible definir una clase dentro de otra clase (**clases 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**.

Las **clases anidadas**, como miembros de una clase que son (miembros de **claseExterna**), pueden ser declaradas con los modificadores **public**, **protected**, **private** o **de paquete**, como el resto de miembros.

Las **clases internas** (no estáticas) tienen acceso a otros miembros de la clase dentro de la que está definida aunque sean privados (se trata en cierto modo de un miembro más de la clase), mientras que las anidadas (estáticas) no.

Las **clases internas** se utilizan en algunos casos para:

* **Agrupar** clases que sólo tiene sentido que existan en el entorno de la clase en la que han sido definidas, de manera que se oculta su existencia al resto del código.
* Incrementar el nivel de **encapsulación** y **ocultamiento**.
* Proporcionar un **código fuente más legible y fácil de mantener** (el código de las **clases internas** y **anidadas** está más cerca de donde es usado).

En Java es posible definir **clases internas** y **anidadas**, permitiendo todas esas posibilidades. Aunque para los ejemplos con los que vas a trabajar no las vas a necesitar por ahora.

**3.- Herencia**

Como ya has estudiado, la **herencia** es el mecanismo que permite definir una nueva clase a partir de otra, pudiendo añadir nuevas características, sin tener que volver a escribir todo el código de la clase base.

La clase de la que se hereda suele ser llamada [**clase base, clase padre** o **superclase**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#t3aee18b7-2a01-637a-7aa3-f353e3cfdc52). A la clase que hereda se le suele llamar [**clase hija, clase derivada** o **subclase**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#t23f8515d-ca4a-0a20-4312-f78696043634).

Una clase derivada puede ser a su vez **clase padre** de otra que herede de ella y así sucesivamente dando lugar a una **jerarquía de clases**, excepto aquellas que estén en la parte de arriba de la jerarquía (sólo serán **clases padre**) o en la parte de abajo (sólo serán **clases hijas**).

Una **clase hija** no tiene acceso a los miembros **privados** de su **clase padre**, tan solo a los **públicos** (como cualquier parte del código tendría) y los **protegidos** (a los que sólo tienen acceso las **clases derivadas** y las del mismo **paquete**). Aquellos miembros que sean privados en la clase base también habrán sido heredados, pero el acceso a ellos estará restringido al propio funcionamiento de la **superclase** y sólo se podrá acceder a ellos si la **superclase** ha dejado algún medio indirecto para hacerlo (por ejemplo a través de algún método).

Todos los miembros de la **superclase**, tanto atributos como métodos, son heredados por la subclase. Algunos de estos miembros heredados podrán ser **redefinidos** o **sobrescritos** (**overriden**) y también podrán añadirse nuevos miembros. De alguna manera podría decirse que estás “ampliando” la **clase base** con características adicionales o modificando algunas de ellas (proceso de **especialización**).

**Una clase derivada extiende la funcionalidad de la clase base sin tener que volver a escribir el código de la clase base.**

Final del formulario

**3.1.- Sintaxis de la herencia**

En Java la **herencia** se indica mediante la palabra reservada **extends**:

Imagina que tienes una clase **Persona** que contiene atributos como **nombre**, **apellidos** y **fechaNacimiento**:

public class Persona {

private String nombre;

private String apellidos;

private LocalDate fechaNacimiento;

…

}

Es posible que, más adelante, necesites una clase **Alumno** que compartirá esos atributos (dado que todo alumno es una persona, pero con algunas características específicas que lo **especializan**). En tal caso tendrías la posibilidad de crear una clase **Alumno** que repitiera todos esos atributos o bien **heredar** de la clase **Persona**:

public class Alumno extends Persona {

private String grupo;

private double notaMedia;

…

}

A partir de ahora, un objeto de la clase **Alumno**contendrá los atributos **grupo** y **notaMedia** (propios de la clase **Alumno**), pero también **nombre**, **apellidos** y **fechaNacimiento** (propios de su **clase base** **Persona**y que por tanto ha heredado).Principio del formulario

Final del formulario

**3.2.- Acceso a miembros heredados**

Como ya has visto anteriormente, no es posible acceder a miembros **privados** de una superclase. Para poder acceder a ellos podrías pensar en hacerlos **públicos**, pero entonces estarías dando la opción de acceder a ellos a cualquier objeto externo y es probable que tampoco sea eso lo deseable. Para ello se inventó el modificador **protected** (**protegido**) que permite el **acceso desde clases heredadas**, pero no desde fuera de las clases (estrictamente hablando, desde fuera del **paquete**), que serían como miembros **privados**.

En la unidad dedicada a la utilización de clases ya estudiaste los posibles modificadores de acceso que podía tener un miembro: **sin modificador** (acceso **de paquete**), **público**, **privado** o **protegido**. Aquí tienes de nuevo el resumen:

|  | **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 |  |  |  |

Si en el ejemplo anterior de la clase **Persona** se hubieran definido sus atributos como **private**:

public class Persona {

private String nombre;

private String apellidos;

…

}

Al definir la clase **Alumno** como heredera de **Persona**, no habrías tenido acceso a esos atributos, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar esos atributos como **protected**:

public class Persona {

protected String nombre;

protected String apellidos;

…

}

Principio del formulario

Final del formulario

**3.3.- Utilización de miembros heredados (I). Atributos**

Los **atributos heredados** por una clase son, a efectos prácticos, iguales que aquellos que sean definidos específicamente en la nueva **clase derivada**.

En el ejemplo anterior la clase **Persona** disponía de tres atributos y la clase **Alumno**, que heredaba de ella, añadía dos atributos más. Desde un punto de vista funcional podrías considerar que la clase **Alumno** tiene cinco atributos: tres por ser **Persona** (**nombre**, **apellidos**, **fechaNacimiento**) y otros dos más por ser **Alumno**(**grupo** y **notaMedia**).

Principio del formulario

Final del formulario

**3.3.1- Utilización de miembros heredados (II). Métodos**

Del mismo modo que se heredan los **atributos**, también se heredan los **métodos**, convirtiéndose a partir de ese momento en otros **métodos** más de la **clase derivada**, junto a los que hayan sido definidos específicamente.

En el ejemplo de la clase **Persona**, si dispusiéramos de métodos **get** y **set** para cada uno de sus tres atributos (**nombre**, **apellidos**, **fechaNacimiento**), tendrías seis métodos que podrían ser heredados por sus **clases derivadas**. Podrías decir entonces que la clase **Alumno**, derivada de **Persona**, tiene diez métodos:

-Seis por ser **Persona** (**getNombre, getApellidos, getFechaNacimiento, setNombre, setApellidos, setFechaNacimien to**).

-Otros cuatro más por ser **Alumno** (**getGrupo**, **setGrupo**, **getNotaMedia**, **setNotaMedia**).

Sin embargo, sólo tendrías que definir esos cuatro últimos (los **específicos**) pues los **genéricos** ya los has heredado de la **superclase**.

Principio del formulario

Final del formulario

**3.4.- Redefinición de métodos heredados**

Una clase puede **redefinir** algunos de los métodos que ha heredado de su **clase base**. En tal caso, el nuevo método (**especializado**) sustituye al **heredado**. Este procedimiento también es conocido como de **sobreescritura de métodos**.

En cualquier caso, 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 redefinidos** pueden **ampliar su accesibilidad** con respecto a la que ofrezca el método original de la **superclase**, pero **nunca restringirla**. Por ejemplo, si un método es declarado como **protected** o **de paquete**en la clase base, podría ser redefinido como **public** en una clase derivada.

**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*.**

En el ejemplo de la clase **Alumno**, podrían redefinirse algunos de los métodos heredados. Por ejemplo, imagina que el método **getApellidos** devuelva la cadena “Alumno:” junto con los apellidos del alumno. En tal caso habría que rescribir ese método para que realizara esa modificación:

public String getApellidos () {

return “Alumno: “ + apellidos;

}

Cuando sobrescribas un método heredado en Java puedes incluir la **anotación** **@Override**. Esto indicará al compilador que tu intención es **sobrescribir el método de la clase padre**. De este modo, si te equivocas (por ejemplo, al escribir el nombre del método) y no lo estás realmente sobrescribiendo, el compilador producirá un error y así podrás darte cuenta del fallo. En cualquier caso, no es necesario indicar **@Override**, pero puede resultar de ayuda a la hora de localizar este tipo de errores (crees que has sobrescrito un **método heredado** y al confundirte en una letra estás realmente creando un nuevo método diferente). En el caso del ejemplo anterior quedaría:

@Override

public String getApellidos ()

Principio del formulario

Final del formulario

**3.5.- Ampliación de métodos heredados**

Hasta ahora, has visto que para **redefinir** o **sustituir** un **método** de una **superclase** es suficiente con crear otro método en la **subclase** que tenga el mismo nombre que el método que se desea **sobrescribir**. Pero, en otras ocasiones, puede que lo que necesites no sea sustituir completamente el comportamiento del método de la superclase, sino simplemente **ampliarlo**.

Para poder hacer esto necesitas poder **preservar el comportamiento antiguo** (el de la **superclase**) y **añadir el nuevo** (el de la **subclase**). Para ello, puedes invocar desde el método “**ampliador**” de la **clase derivada** al método “**ampliado**” de la clase superior (teniendo ambos métodos el mismo nombre). ¿Cómo se puede conseguir eso? Puedes hacerlo mediante el uso de la referencia **super**.

La palabra reservada **super** es una referencia a la **clase padre** de la clase en la que te encuentres en cada momento (es algo similar a **this**, que representaba una referencia a la **clase actual**). De esta manera, podrías invocar a cualquier método de tu **superclase** (si es que se tiene acceso a él).

Por ejemplo, imagina que la clase **Persona** dispone de un método que permite mostrar el contenido de algunos datos personales de los objetos de este tipo (**nombre**, **apellidos**, etc.). Por otro lado, la clase **Alumno** también necesita un método similar, pero que muestre también su información especializada (**grupo**, **nota media**, etc.). ¿Cómo podrías aprovechar el método de la **superclase** para no tener que volver a escribir su contenido en la subclase?

Podría hacerse de una manera tan sencilla como la siguiente:

/\*\*

\* 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();

}

Este tipo de **ampliaciones de métodos** resulta especialmente útil en el caso de los **constructores**, donde se podría ir llamando a los **constructores** de cada **superclase** encadenadamente hasta el **constructor** de la clase en la **cúspide de la jerarquía** (el **constructor** de la clase **Object**). En el caso de los constructores se usa directamente el método super() para invocar al constructor de la superclase.

El ejemplo anterior para el método toString() sirve para proporcionar una representación como String de los objetos de la clase. La parte que convierte en cadena los atributos comunes de la superclase la haría el método toString() de la superclase, por lo que dentro del método toString() de la subclase lo invocaríamos como super.toString(), y a la cadena que devuelve, le añadiríamos la conversión en cadena de los atributos específicos de la subclase.

Principio del formulario

Final del formulario

**3.6.- Constructores y herencia**

Recuerda que cuando estudiaste los **constructores** viste que un **constructor** de una clase puede llamar a otro **constructor** de la misma clase, previamente definido, a través del método **this()**. En estos casos, la utilización de **this()** sólo podía hacerse en la primera línea de código del **constructor**.

Como ya has visto, un **constructor** de una **clase derivada** puede hacer algo parecido para llamar al **constructor** de su **clase base** mediante el uso del método **super()**. De esta manera, el **constructor** de una **clase derivada** puede llamar primero al **constructor** de su **superclase** para que inicialice los **atributos heredados** y posteriormente se inicializarán los **atributos específicos** de la clase. Nuevamente, esta llamada también **debe ser la primera sentencia de un constructor** (con la única excepción de que exista una llamada a otro constructor de la clase mediante **this()**).

Si no se incluye una llamada a **super()** dentro de un **constructor**, el compilador incluye automáticamente una llamada al constructor por defecto de **clase base**(llamada a super()). Esto da lugar a una **llamada en cadena de constructores de superclase** hasta llegar a la clase más alta de la jerarquía (que en Java es la clase **Object**).

En el caso del **constructor por defecto** (el que crea el compilador si el programador no ha escrito ninguno), el compilador añade lo primero de todo, antes de la inicialización de los atributos a sus valores por defecto, una llamada al constructor de la **clase base** mediante **super()**.

A la hora de destruir un objeto (método **finalize()**) es importante llamar a los finalizadores en el **orden inverso** a como fueron llamados los constructores (**primero se liberan los recursos de la clase derivada y después los de la clase base** mediante la llamada **super.finalize()**).

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

/\*\*

\* 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;

}

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

/\*\*

\* 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;

}

En realidad se trata de otro recurso más para optimizar la **reutilización de código**, en este caso del código del **constructor**, que aunque no es heredado sí puedes invocarlo para no tener que rescribirlo. Al hacerlo recuerda siempre que:

**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()).**

Principio del formulario

Final del formulario

**3.7.- Creación y utilización de clases derivadas**

Ya has visto cómo crear una **clase derivada**, cómo acceder a los **miembros heredados** de las **clases superiores**, cómo redefinir algunos de ellos e incluso cómo invocar a un **constructor** de la **superclase**. Ahora se trata de poner en práctica todo lo que has aprendido para que puedas crear tus propias **jerarquías de clases,**o basarte en clases que ya existan en Java para heredar de ellas, y usarlas de manera adecuada para que tus aplicaciones sean más fáciles de escribir y mantener.

La idea de la **herencia** no es complicar los programas, sino todo lo contrario: **simplificarlos al máximo**. Procurar que haya que escribir la menor cantidad posible de código repetitivo e intentar facilitar en lo posible la realización de cambios (bien para corregir errores bien para incrementar la funcionalidad).

**3.8.- La clase Object en Java**

Todas las clases en Java son descendientes (directos o indirectos) de la clase **Object**. Esta clase define los **estados y comportamientos básicos que deben tener todos los objetos**. Entre estos comportamientos, se encuentran:

* La posibilidad de compararse.
* La capacidad de convertirse a cadenas.
* La habilidad de devolver la clase del objeto.

Entre los métodos que incorpora la clase **Object**, y que por tanto hereda cualquier clase en Java, tienes:

| **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**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#t8f87c969-fb62-def5-5d94-0c48364ee009) para el objeto. |
| **toString ()** | Devuelve una representación del objeto en forma de **String**. |

La clase **Object** representa la **superclase** que se encuentra en la cúspide de la **jerarquía de herencia** en Java. Cualquier clase (incluso las que tú implementes) acaban heredando de ella.

Final del formulario

**3.9.- Herencia múltiple**

En determinados casos podrías considerar la posibilidad de que se necesite **heredar de más de una clase**, para así disponer de los miembros de dos (o más) clases disjuntas (que no derivan una de la otra). La **herencia múltiple** permite hacer eso: recoger las distintas características (atributos y métodos) de clases diferentes formando una nueva clase derivada de varias clases base.

El problema en estos casos es la posibilidad que existe de que se produzcan ambigüedades. Así, si tuviéramos miembros con el mismo identificador en clases base diferentes, ¿qué miembro se hereda? ¿de qué clase padre? Para evitar esto, los compiladores suelen solicitar que ante casos de ambigüedad, se especifique de manera explícita la clase de la cual se quiere utilizar un determinado miembro que pueda ser ambiguo.

Ahora bien, la posibilidad de **herencia múltiple** no está disponible en todos los lenguajes orientados a objetos, ¿lo estará en Java? La respuesta es negativa.

**En Java no existe la herencia múltiple de clases.**

**4.- Clases abstractas**

En determinadas ocasiones, es posible que necesites definir una clase que represente un concepto lo suficientemente abstracto como para que nunca vayan a existir instancias de ella (objetos). ¿Tendría eso sentido? ¿Qué utilidad podría tener?

Imagina una aplicación para un **centro educativo** que utilice las clases de ejemplo **Alumno** y **Profesor**, ambas subclases de **Persona**. Es más que probable que esa aplicación nunca llegue a necesitar objetos de la clase **Persona**, pues serían demasiado genéricos como para poder ser utilizados (no contendrían suficiente información específica). Podrías llegar entonces a la conclusión de que la clase **Persona** ha resultado de utilidad como **clase base** para construir otras clases que hereden de ella, pero no como una **clase instanciable** de la cual vayan a existir objetos. A este tipo de clases se les llama **clases abstractas**.

**En algunos casos puede resultar útil disponer de clases que nunca serán instanciadas, sino que proporcionan un marco o modelo a seguir por sus clases derivadas dentro de una jerarquía de *herencia*. Son las *clases abstractas*.**

La posibilidad de declarar **clases abstractas** es una de las características más útiles de los **lenguajes orientados a objetos**, pues permiten dar unas líneas generales de cómo es una clase sin tener que implementar todos sus métodos o implementando solamente algunos de ellos. Esto resulta especialmente útil cuando las distintas **clases derivadas** deban proporcionar los mismos métodos indicados en la clase **base abstracta**, pero su i**mplementación sea específica** para cada **subclase**.

Imagina que estás trabajando en un entorno de **manipulación de objetos gráficos** y necesitas trabajar con **líneas**, **círculos**, **rectángulos**, etc. Estos objetos tendrán en común algunos atributos que representen su estado (**ubicación**, **color del contorno**, **color de relleno**, etc.) y algunos métodos que modelen su comportamiento (**dibujar**, **rellenar con un color**, **escalar**, **desplazar**, **rotar**, etc.). Algunos de ellos serán comunes para todos ellos (por ejemplo la **ubicación** o el **desplazamiento**) y sin embargo otros (como por ejemplo **dibujar**) necesitarán una implementación específica dependiendo del tipo de objeto. Pero, en cualquier caso, todos ellos necesitan esos métodos (tanto un **círculo** como un **rectángulo** necesitan el método **dibujar**, aunque se lleven a cabo de manera diferente). En este caso resultaría muy útil disponer de una **clase abstracta** **objeto gráfico** donde se definirían las **líneas generales** (algunos atributos concretos comunes, algunos métodos concretos comunes implementados y algunos métodos genéricos comunes sin implementar) de un objeto gráfico y más adelante, según se vayan definiendo **clases especializadas** (**líneas**, **círculos**, **rectángulos**), se irán concretando en cada **subclase** aquellos métodos que se dejaron sin implementar en **la clase abstracta**.

Final del formulario

**4.1.- Declaración de una clase abstracta**

Ya has visto que una **clase abstracta** es una clase que no se puede instanciar, es decir, que no se pueden crear objetos a partir de ella. La idea es permitir que otras clases deriven de ella, proporcionando un **modelo genérico** y algunos **métodos de utilidad general**.

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

[modificador\_acceso] abstract class nombreClase [herencia] [interfaces] {

…

}

Una clase puede contener en su interior métodos declarados como **abstract** (métodos para los cuales sólo se indica la cabecera, pero no se proporciona su implementación). En tal caso, la clase tendrá que ser necesariamente también **abstract**. Esos métodos tendrán que ser posteriormente implementados en sus **clases derivadas**.

Por otro lado, una **clase abstracta** también puede contener **métodos totalmente implementados** (**no abstractos**), los cuales serán heredados por sus **clases derivadas** y podrán ser utilizados sin necesidad de definirlos (pues ya están implementados).

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

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

Principio del formulario

Final del formulario

**4.2.- Métodos abstractos**

Un [**método abstracto**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#td5632ffc-786b-1d12-889d-8314451da4f9) es un método cuya implementación no se define, sino que se declara únicamente su **interfaz** (cabecera) para que su cuerpo sea implementado más adelante en una **clase derivada**.

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

[modificador\_acceso] abstract <tipo> <nombreMetodo> ([parámetros]) [excepciones];

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

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

Imagina que tienes una clase **Empleado** genérica para diversos tipos de empleado y tres **clases derivadas**: **EmpleadoFijo** (tiene un salario fijo más ciertos complementos), **EmpleadoTemporal** (salario fijo más otros complementos diferentes) y **EmpleadoComercial** (una parte de salario fijo y unas comisiones por cada operación). La clase **Empleado** podría contener un **método abstracto** **calcularNomina**, pues sabes que se método será necesario para cualquier tipo de empleado (todo empleado cobra una nómina). Sin embargo el cálculo en sí de la nómina será diferente si se trata de un empleado fijo, un empleado temporal o un empleado comercial, y será dentro de las clases especializadas de **Empleado** (**EmpleadoFijo¸** **EmpleadoTemporal**, **EmpleadoComercial**) donde se implementen de manera específica el cálculo de las mismas.

Debes tener en cuenta al trabajar con 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).

Principio del formulario

Final del formulario

**4.3.- Clases y métodos finales**

En unidades anteriores has visto el modificador **final**, aunque sólo lo has utilizado por ahora para **atributos** y **variables** (por ejemplo para declarar **atributos constantes**, que una vez que toman un valor ya no pueden ser modificados). Pero este modificador también puede ser utilizado con clases y con métodos (con un comportamiento que no es exactamente igual, aunque puede encontrarse cierta analogía: no se permite heredar o no se permite redefinir).

**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]

**Si intentas redefinir un método final en una subclase se producirá un *error de compilación*.**

Final del formulario

Además de en la declaración de atributos, clases y métodos, el modificador **final** también podría aparecer acompañando a un método de un parámetro. En tal caso no se podrá modificar el valor del parámetro dentro del código del método. Por ejemplo: **public final metodoEscribir (int par1, <span style="color: #0075be">final</span> int par2)**.

**5.- Interfaces**

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

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

Si llevamos al límite esta idea de **interfaz**, podrías llegar a tener una **clase abstracta** donde todos sus métodos fueran abstractos. De este modo estarías dando únicamente el **marco de comportamiento**, sin ningún método implementado, de las posibles **subclases** que heredarán de esa **clase abstracta**. La idea de una **interfaz** (o **interface**) es precisamente ésa: **disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación** (no necesariamente jerárquica).

Una **interfaz** consiste principalmente en una lista de declaraciones de métodos sin implementar, que caracterizan un determinado comportamiento. Si se desea que una clase tenga ese comportamiento, tendrá que implementar esos métodos establecidos en la **interfaz**. En este caso no se trata de una relación de **herencia** (la clase **A** es una especialización de la clase **B**, o la subclase **A** es del tipo de la superclase **B**), sino más bien una relación "de implementación de comportamientos" (la clase **A** implementa los métodos establecidos en la **interfaz** **B**, o los comportamientos indicados por **B** son llevados a cabo por **A**; pero no que **A** sea de clase **B**).

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieres que lleven a cabo están relacionadas con el hecho de que algunos animales sean **depredadores** (porejemplo: **observar** una **presa**, **perseguirla**, **comérsela**, etc.)osean **presas** (**observar**, **huir**, **esconderse**, etc.). Si creas la clase Leon, esta clase podría implementar una interfaz Depredador, mientras que otras clases como Gacela implementarían las acciones de la interfaz Presa. Por otro lado, podrías tener también el caso de la clase Rana, que implementaría las acciones de la interfaz Depredador(pues es cazador de pequeños insectos), pero también la de Presa (pues puede ser cazado y necesita las acciones necesarias para protegerse).

**5.1.- Concepto de interfaz**

Una **interfaz** en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.

Estos métodos sin implementar indican un **comportamiento**, un tipo de conducta, aunque no especifican cómo será ese **comportamiento** (**implementación**), pues eso dependerá de las características específicas de cada clase que decida implementar esa **interfaz**. Podría decirse que una **interfaz** se encarga de establecer qué **comportamientos** hay que tener (qué **métodos**), pero no dice nada de cómo deben llevarse a cabo esos **comportamientos** (**implementación**). Se indica sólo la **forma**, no la **implementación**.

En cierto modo podrías imaginar el concepto de **interfaz** como un **guión** que dice: "éste es el protocolo de comunicación que deben presentar todas las clases que implementen esta interfaz". Se proporciona una lista de **métodos públicos** y, si quieres dotar a tu clase de esa **interfaz**, tendrás que definir todos y cada uno de esos **métodos públicos**.

En conclusión: **una interfaz se encarga de establecer unas líneas generales sobre los comportamientos (métodos) que deberían tener los objetos de toda clase que implemente esa interfaz, es decir, que no indican lo que el objeto es**(de eso se encarga la clase y sus **superclases**)**, sino acciones (capacidades) que el objeto debería ser capaz de realizar**. Es por esto que el nombre de muchas interfaces en Java termina con sufijos del tipo "-**able**", "-**or**", "**-ente**", , "**-ante**" y cosas del estilo, que significan algo así como **capacidad o habilidad** para hacer o ser receptores de algo (**configurable**, **serializable**, **modificable**, **clonable**, **ejecutable**, **imprimible**, **movible**, **administrador**, **servidor, buscador**, **depredador**, **luchador**, **atacante**, **andante**, etc.), dando así la idea de que se tiene la capacidad de llevar a cabo el conjunto de acciones especificadas en la **interfaz**.

Imagínate por ejemplo la clase **Coche**, **subclase** de **Vehículo**. Los coches son **vehículos a motor**, lo cual implica una serie de acciones como, por ejemplo, **arrancar el motor** o **detener el motor**. Esa acción no la puedes heredar de **Vehículo**, pues no todos los vehículos tienen por qué ser a motor (piensa por ejemplo en una clase **Bicicleta**), y no puedes heredar de otra clase pues ya heredas de **Vehículo**. Una solución podría ser crear una **interfaz Arrancable**, que proporcione los métodos típicos de un **objeto a motor** (no necesariamente vehículos). De este modo la clase **Coche** sigue siendo subclase de **Vehículo**, pero también implementaría los comportamientos de la interfaz **Arrancable**, los cuales podrían ser también implementados por otras clases, hereden o no de **Vehículo** (por ejemplo una clase **Motocicleta** o bien una clase **Motosierra**). La clase **Coche** implementará su método **arrancar** de una manera, la clase **Motocicleta** lo hará de otra (aunque bastante parecida) y la clase **Motosierra** de otra forma probablemente muy diferente, pero todos tendrán su propia versión del método **arrancar** como parte de la interfaz **Arrancable**.

Según esta concepción, podrías hacerte la siguiente pregunta: **¿podrá una clase implementar varias interfaces?** La respuesta en este caso sí es afirmativa.

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

Final del formulario

**5.1.1.- ¿Clase abstracta o interfaz?**

Observando el concepto de **interfaz** que se acaba de proponer, podría caerse en la tentación de pensar que es prácticamente lo mismo que una **clase abstracta** en la que **todos sus métodos sean abstractos**.

Es cierto que en ese sentido existe un gran **parecido formal** entre una **clase abstracta** y una **interfaz**, pudiéndose en ocasiones utilizar indistintamente una u otra para obtener un mismo fin. Pero, a pesar de ese gran parecido, existen algunas diferencias, no sólo formales, sino también conceptuales, muy importantes:

* **Una clase no puede heredar de varias clases**, aunque sean abstractas (**herencia múltiple**). Sin embargo sí puede **implementar una o varias interfaces** y además seguir heredando de una clase.
* **Una interfaz no puede definir métodos** (**no implementa su contenido**), tan solo los declara o enumera.
* **Una interfaz puede hacer que dos clases tengan un mismo comportamiento** independientemente de sus ubicaciones en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).
* **Una interfaz permite establecer un comportamiento de clase sin apenas dar detalles**, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la **interfaz**).
* **Las interfaces tienen su propia jerarquía**, diferente e independiente de la jerarquía de clases.

De todo esto puede deducirse que **una clase abstracta proporciona una interfaz disponible sólo a través de la herencia**. Sólo quien herede de esa **clase abstracta** dispondrá de esa **interfaz**. Si una clase no pertenece a esa misma jerarquía (no hereda de ella) no podrá tener esa **interfaz**. Eso significa que para poder disponer de la **interfaz** podrías:

1. Volver a escribirla para esa jerarquía de clases. Lo cual no parece una buena solución.
2. Hacer que la clase herede de la superclase que proporciona la **interfaz** que te interesa, sacándola de su jerarquía original y convirtiéndola en **clase derivada** de algo de lo que conceptualmente no debería ser una **subclase**. Es decir, estarías forzando una relación "**es un**" cuando en realidad lo más probable es que esa relación no exista. Tampoco parece la mejor forma de resolver el problema.

Sin embargo, **una interfaz sí puede ser implementada por cualquier clase**, permitiendo que clases que no tengan ninguna relación entre sí (pertenecen a distintas jerarquías) puedan compartir un determinado comportamiento (una interfaz) sin tener que forzar una relación de herencia que no existe entre ellas.

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**.

**Recomendación**

Si solo vas a proporcionar una lista de **métodos abstractos** (**interfaz**), sin definiciones de métodos ni atributos de objeto, suele ser recomendable definir una **interfaz** antes que **clase abstracta**. Es más, cuando vayas a definir una supuesta **clase base**, puedes comenzar declarándola como **interfaz** y sólo cuando veas que necesitas definir métodos o variables miembro, puedes entonces convertirla en **clase abstracta** (no instanciable) o incluso en una **clase instanciable**.

Final del formulario

**5.2.- Definición de interfaces**

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

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

Como puedes observar, una **interfaz** consiste esencialmente en una lista de **atributos finales constantes** y **métodos abstractos (sin implementar**). Su sintaxis quedaría entonces:

[public] interface <NombreInterfaz> {

[public] [final] <tipo1> <atributo1>= <valor1>;

[public] [final] <tipo2> <atributo2>= <valor2>;

...

[public] [abstract] <tipo\_devuelto1> <nombreMetodo1> ([lista\_parámetros]);

[public] [abstract] <tipo\_devuelto2> <nombreMetodo2> ([lista\_parámetros]);

...

}

Si te fijas, la declaración de los métodos termina en punto y coma, pues no tienen cuerpo, al igual que sucede con los **métodos abstractos** de las **clases abstractas**.

El ejemplo de la interfaz Depredadorque hemos visto antes podría quedar entonces así:

public interface Depredador {

void localizar (Animal presa);

void cazar (Animal presa);

...

}

Serán las clases que implementen esta interfaz (Leon, Leopardo, Cocodrilo, Rana, Lagarto, Hombre, etc.) las que definan cada uno de los métodos por dentro.

Principio del formulario

Final del formulario

**5.3.- Implementación de interfaces**

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

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

class NombreClase implements NombreInterfaz {

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

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

class NombreClase implements NombreInterfaz1, NombreInterfaz2,... {

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

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

En el ejemplo de los depredadores, al definir la clase Leon, habría que indicar que implementa la **interfaz** Depredador:

class Leon implements Depredador {

Y en su interior habría que implementar aquellos métodos que contenga la **interfaz**:

void localizar (Animal presa) {

// Implementación del método localizar para un león

...

}

En el caso de clases que pudieran ser a la vez Depredador y Presa, tendrían que implementar ambas interfaces, como podría suceder con la clase Rana:

class Rana implements Depredador, Presa {

Y en su interior habría que implementar aquellos métodos que contengan ambas **interfaces**, tanto las de Depredador (**localizar**, cazar, etc.) como las de Presa (**observar**, huir, etc.).

Principio del formulario

Final del formulario

**5.3.1.- Un ejemplo de implementación de interfaces**

El trabajo con **interfaces** es algo habitual en el desarrollo de aplicaciones en Java. Es por tanto muy importante comprender correctamente su funcionamiento y la interacción con las distintas **bibliotecas** (**paquetes de clases** e **interfaces**) que proporcionan las API. Estas **clases** e **interfaces** son fundamentales para la creación de aplicaciones y tendrás que utilizarlas en multitud de ocasiones (además, por supuesto, de las que tengas que desarrollar por ti mismo).

Vamos a ver un ejemplo de una **interfaz** proporcionada por la API de Java que puede ser implementada por alguna clase creada por ti dentro de una pequeña aplicación. Hemos escogido la interfaz Comparable. Más adelante, en la unidad de interfaces gráficas, veremos otro ejemplo bastante ilustrativo con la interfaz ActionListener. Esta interfeaz es de gran utilidad para las clases que quieran realizar una determinada acción cada vez que se produzca cierto **evento** en el sistema. Este tipo de **interfaces** se encuentran dentro de los **Event Listeners** u "**oyentes de eventos**" y son útiles para detectar que se ha producido un determinado **evento**[**asíncrono**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#t353dd465-1a13-00fb-de1a-daeb794b035d) durante la ejecución de tu aplicación como pulsación de una tecla, clic de ratón, etc. Son intensivamente utilizadas en el desarrollo de las **interfaces gráficas de usuario**.

Vamos a ver un **ejemplo** lo más sencillo posible: se trata de desarrollar **una pequeña aplicación para comparar personas**.

Para ello, incluiremos en el código implements Comparable, que indica que vamos a implementar en la clase la interfaz Comparable. También añadiremos el código necesario para implementar el método public int compareTo(Object o). Ese método es el que implementaremos según nuestro criterio de comparación. Así, por ejemplo, decidimos que vamos a comparar las personas por su edad, aunque podríamos haber elegido otro criterio. De este modo, y para seguir el criterio que se sigue para devolver valores con este método, si la edad de la persona es menor que la de la otra persona que se manda como parámetro al método, entonces se devolverá -1. Si es mayor se devolverá 1. Y si tienen la misma edad se devolverá 0.

Desde el siguiente enlace puedes descargar el ejemplo completo en el que se implementa la interfaz Comparable en un pequeño programa que crea un par de objetos Persona, los instancia y compara.

[Proyecto EjemploInterfazComparable.](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/PROG07_CONT_R62_EjemploInterfazComparable.zip) (18 KB)

**5.4.- Simulación de la herencia múltiple mediante el uso de interfaces**

Una **interfaz** no tiene **espacio de almacenamiento** asociado (no se van a declarar objetos de un tipo de interfaz), es decir, no tiene **implementación**.

En algunas ocasiones es posible que interese representar la situación de que "una clase **X** es de tipo **A**, de tipo **B**, y de tipo **C**", siendo **A**, **B**, **C** **clases disjuntas** (no heredan unas de otras). Hemos visto que sería un caso de **herencia múltiple** que Java no permite.

Para poder simular algo así, podrías definir tres **interfaces** **A**, **B**, **C** que indiquen los comportamientos (métodos) que se deberían tener según se pertenezca a una supuesta clase **A**, **B**, o **C**, pero sin implementar ningún método concreto ni atributos de objeto (sólo interfaz).

De esta manera la clase **X** podría a la vez:

1. Implementar las interfaces **A**, **B**, **C**, que la dotarían de los comportamientos que deseaba heredar de las clases **A**, **B**, **C**.
2. Heredar de otra clase **Y**, que le proporcionaría determinadas características dentro de su taxonomía o jerarquía de objeto (atributos, métodos implementados y métodos abstractos).

En el ejemplo que hemos visto de las interfaces Depredador y **Presa**, tendrías un ejemplo de esto: la clase Rana, que es subclase de Anfibio, implementa una serie de **comportamientos** propios de un Depredador y, a la vez, otros más propios de una Presa. Esos **comportamientos** (**métodos**) no forman parte de la **superclase** Anfibio, sino de las **interfaces**. Si se decide que la clase Rana debe de llevar a cabo algunos otros **comportamientos adicionales**, podrían añadirse a una **nueva interfaz** y la clase Rana **implementaría** una tercera **interfaz**.

De este modo, con el mecanismo "**una herencia pero varias interfaces**", podrían conseguirse resultados similares a los obtenidos con la **herencia múltiple**.

Ahora bien, del mismo modo que sucedía con la **herencia múltiple**, puede darse el problema de la **colisión de nombres** al implementar dos **interfaces** que tengan un **método con el mismo identificador**. En tal caso puede suceder lo siguiente:

* Si los dos métodos tienen **diferentes parámetros** no habrá problema aunque tengan el mismo nombre pues se realiza una **sobrecarga** de métodos.
* Si los dos métodos se diferencian solo en su**valor de retorno que es de un tipo diferente**, se producirá un **error de compilación** (al igual que sucede en la sobrecarga cuando la única diferencia entre dos métodos es ésa).

Si los dos métodos son **exactamente iguales en identificador, parámetros y tipo devuelto**, entonces solamente se podrá **implementar uno de los dos métodos**. En realidad se trata de un solo método pues ambos tienen la misma interfaz (mismo identificador, mismos parámetros y mismo tipo devuelto).

**Recomendación**

La utilización de nombres idénticos en diferentes **interfaces** que pueden ser implementadas a la vez por una misma clase puede causar, además del problema de la **colisión de nombres**,  dificultades de **legibilidad** en el código, pudiendo dar lugar a confusiones. Si es posible intenta evitar que se produzcan este tipo de situaciones.

Final del formulario

Principio del formulario

Final del formulario

**5.5.- 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.

Por ejemplo, dadas las interfaces InterfazUno e **InterfazDos**:

public interface InterfazUno {

// Métodos y constantes de la interfaz Uno

}

public interface InterfazDos {

// Métodos y constantes de la interfaz Dos

}

Podría definirse una nueva **interfaz** que heredara de ambas:

public interface InterfazCompleja extends InterfazUno, InterfazDos {

}

Principio del formulario

Final del formulario

**6.- Polimorfismo**

El **polimorfismo** es otro de los grandes pilares sobre los que se sustenta la **Programación Orientada a Objetos** (junto con la **encapsulación** y la **herencia**). Se trata nuevamente de otra forma más de establecer diferencias entre interfaz e implementación, es decir, entre **el qué** y **el cómo**.

La **encapsulación** te ha permitido agrupar **características** (**atributos**) y **comportamientos** (**métodos**) dentro de una misma unidad (**clase**), pudiendo darles un mayor o menor componente de **visibilidad**, y permitiendo separar al máximo la **interfaz** de la **implementación**. Por otro lado la **herencia** te ha proporcionado la posibilidad de tratar a los objetos como pertenecientes a una **jerarquía de clases**. Esta capacidad va a ser fundamental a la hora de poder manipular muchos posibles objetos de clases diferentes como si fueran de la misma clase (**polimorfismo**).

El **polimorfismo** te va a permitir mejorar la **organización** y la **legibilidad** del código así como la posibilidad de desarrollar aplicaciones que sean más fáciles de ampliar a la hora de incorporar nuevas funcionalidades. Si la implementación y la utilización de las clases es lo suficientemente genérica y extensible será más sencillo poder volver a este código para incluir nuevos requerimientos.

Final del formulario

**6.1.- Concepto de polimorfismo**

El **polimorfismo** consiste en la capacidad de poder utilizar una referencia a un objeto de una determinada clase como si fuera de otra clase (en concreto una **subclase**). Es una manera de decir que una clase podría tener varias (poli) formas (morfismo).

Un método "**polimórfico**" ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en **tiempo de ejecución** en lugar de en **tiempo de compilación**. Para poder hacer algo así es necesario utilizar métodos que pertenecen a una **superclase** y que en cada **subclase** se implementan de una forma en particular. En **tiempo de compilación** se invocará al método sin saber exactamente si será el de una subclase u otra (pues se está invocando al de la **superclase**). Sólo en **tiempo de ejecución** (una vez instanciada una u otra **subclase**) se conocerá realmente qué método (de qué **subclase**) es el que finalmente va a ser invocado.

Esta forma de trabajar te va a permitir hasta cierto punto "desentenderte" del tipo de objeto **específico** (**subclase**) para centrarte en el tipo de objeto **genérico** (**superclase**). De este modo podrás manipular objetos hasta cierto punto "desconocidos" en tiempo de compilación y que sólo durante la ejecución del programa se sabrá exactamente de qué tipo de objeto (**subclase**) se trata.

**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.***

Dada una **superclase** **X**, con un método **m**, y dos **subclases** **A** y **B**, que redefinen ese método **m**, podrías declarar un objeto **O** de tipo **X** que durante la **ejecución** podrá ser de tipo **A** o de tipo **B** (algo desconocido en **tiempo de compilación**). Es decir, podemos tener declarada una referencia **O** de tipo **X** en tiempo de compilación, que en tiempo de ejecución podrá acabar apuntando a un objeto concreto de tipo **A** o de tipo **B**, ya que todo objeto de las subclases "lleva dentro" un objeto de la superclase, que puede ser apuntado sin problemas con la referencia de la superclase. Esto significa que  al invocarse el método **m** de **X** (**superclase**), se estará en realidad invocando al método **m** de **A** o de **B** (alguna de sus **subclases**), sin que sepamos cuál hasta que llegue ese momento a la hora de ejecutar el programa. Por ejemplo:

// Declaración de una referencia a un objeto de tipo X

ClaseX obj; // Objeto de tipo X (superclase)

...

// Zona del programa donde se instancia un objeto de tipo A (subclase) y se le asigna a la referencia obj.

// La variable obj adquiere la forma de la subclase A. (Todo objeto A lleva dentro un objeto de su superclase ClaseX)

obj = new ClaseA ();

...

// Otra zona del programa.

// Aquí se instancia un objeto de tipo B (subclase) y se le asigna a la referencia obj.

// La variable obj adquiere la forma de la subclase B. (Todo objeto A lleva dentro un objeto de su superclase ClaseX)

obj = new ClaseB ();

...

// Zona donde se utiliza el método m sin saber realmente qué subclase se está utilizando.

// (Sólo se sabrá durante la ejecución del programa)

obj.m () // Llamada al método m (sin saber si será el método m de A o de B). Se asociará o "ligará" al m() adecuado

//durante la ejecución, que será cuando se sabrá si obj apunta a un objeto tipo A o tipo B. (por eso se llama ligadura dinámica).

...

Imagina que estás trabajando con las clases Alumno y Profesor y que en determinada zona del código podrías tener objetos, tanto de un tipo como de otro, pero eso sólo se sabrá según vaya discurriendo la ejecución del programa. En algunos casos, es posible que un determinado objeto pudiera ser de la clase Alumno y en otros de la clase Profesor, pero en cualquier caso serán objetos de la clase Persona. Eso significa que la llamada a un método de la clase Persona (por ejemplo getContenidoString) en realidad será en unos casos a un método (con el mismo nombre) de la clase Alumno y en otros, a un método (con el mismo nombre también) de la clase Profesor. Esto será posible hacerlo gracias a la **ligadura dinámica**.

Final del formulario

**6.2.- Ligadura dinámica**

La conexión que tiene lugar durante una llamada a un método suele ser llamada [**ligadura**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#t892f4849-e206-861c-1a40-ab34f81bbc93), **vinculación** o **enlace** (en inglés **binding**). Si esta **vinculación** se lleva a cabo durante el proceso de compilación, se le suele llamar [**ligadura estática**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#ta2f0b153-c560-9d47-bea7-d47717576a88)(también conocido como **vinculación temprana**). En los lenguajes tradicionales, no orientados a objetos, ésta es la única forma de poder resolver la **ligadura** (en **tiempo de compilación**). Sin embargo, en los **lenguajes orientados a objetos** existe otra posibilidad: la [**ligadura dinámica**](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55845/mod_scorm/content/0/#t916c977f-5376-f92c-44e0-515574eb9244)(también conocida como **vinculación tardía, enlace tardío**o **late binding**).

La **ligadura dinámica** hace posible que sea el **tipo de objeto** instanciado (obtenido mediante el **constructor** finalmente utilizado para crear el objeto) y no el **tipo de la referencia** (el tipo indicado en la declaración de la variable que apuntará al objeto) lo que determine qué versión del método va a ser invocada. El **tipo de objeto** al que apunta la variable de tipo referencia sólo podrá ser conocido durante la **ejecución** del programa y por eso el **polimorfismo** necesita la **ligadura dinámica**.

En el ejemplo anterior de la clase **X** y sus **subclases** **A** y **B**, la llamada al método **m** sólo puede resolverse mediante ligadura dinámica, pues es imposible saber en tiempo de compilación si el método **m** que debe ser invocado será el definido en la subclase **A** o el definido en la subclase B:

// Llamada al método m (sin saber si será el método m de A o de B).

obj.m () // Esta llamada será resuelta en tiempo de ejecución (ligadura dinámica)

Principio del formulario

Final del formulario

**6.3.- Limitaciones de la ligadura dinámica**

Como has podido comprobar, el **polimorfismo** se basa en la utilización de **referencias** de un tipo más "amplio" (**superclases**) que los objetos a los que luego realmente van a apuntar (**subclases**). Ahora bien, existe una importante **restricción** en el uso de esta capacidad, pues el tipo de referencia limita cuáles son los métodos que se pueden utilizar y los atributos a los que se pueden acceder.

**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*.**

Veamos un ejemplo: si dispones de una clase **A** que es subclase de **B** y declaras una variable como referencia un objeto de tipo **B**. Aunque más tarde esa variable haga referencia a un objeto de tipo **A** (**subclase**), los miembros a los que podrás acceder sin que el compilador produzca un error serán los miembros de **A** que hayan sido heredados de **B** (**superclase**). De este modo, se garantiza que los métodos que se intenten llamar van a existir cualquiera que sea la subclase de **B** a la que se apunte desde esa referencia.

En el ejemplo de las clases Persona, **Profesor** y Alumno, el **polimorfismo** nos permitiría declarar variables de tipo Persona y más tarde hacer con ellas referencia a objetos de tipo Profesor o **Alumno**, pero no deberíamos intentar acceder con esa variable a métodos que sean específicos de la clase Profesor o de la clase Alumno, tan solo a métodos que sabemos que van a existir seguro en ambos tipos de objetos (métodos de la **superclase** Persona).

Principio del formulario

Final del formulario

**6.4.- Interfaces y polimorfismo**

Es posible también llevar a cabo el **polimorfismo** mediante el uso de **interfaces**. Un objeto puede tener una referencia cuyo tipo sea una **interfaz**, pero para que el compilador te lo permita, la clase cuyo **constructor** se utilice para crear el objeto deberá implementar esa **interfaz** (bien por sí misma o bien porque la implemente alguna **superclase**). 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**.

Las referencias de tipo **interfaz** permiten unificar de una manera bastante estricta la forma de utilizar objetos que pertenezcan a clases muy diferentes (pero que todas ellas implementan la misma **interfaz**). De este modo podrías hacer referencia a diferentes objetos que no tienen ninguna relación jerárquica entre sí utilizando la misma variable (referencia a la **interfaz**). Lo único que los distintos objetos tendrían en común es que implementan la misma **interfaz**. En este caso sólo podrás llamar a los métodos de la **interfaz** y no a los específicos de las clases.

Por ejemplo, si tenías una variable de tipo referencia a la interfaz Arrancable, podrías instanciar objetos de tipo Coche o **Motosierra** y asignarlos a esa referencia (teniendo en cuenta que ambas clases no tienen una relación de herencia). Sin embargo, tan solo podrás usar en ambos casos los métodos y los atributos de la interfaz Arrancable (por ejemplo arrancar) y no los de Coche o los de Motosierra (sólo los genéricos, nunca los específicos).

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

Final del formulario

**6.5.- Conversión de objetos**

Como ya has visto, en principio no se puede acceder a los **miembros específicos** de una **subclase** a través de una **referencia** a una **superclase**. 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). Se realizará una **conversión implícita o automática** de **subclase** a **superclase** siempre que sea necesario, pues un objeto de tipo **subclase** siempre contendrá toda la información necesaria para ser considerado un objeto de la **superclase**.

Ahora bien, 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**.

Por ejemplo, imagina que tienes una clase **A** y una clase **B**, **subclase** de **A**:

class ClaseA {

public int atrib1;

}

class ClaseB extends ClaseA {

public int atrib2;

}

A continuación declaras una variable referencia a la clase **A** (**superclase**) pero sin embargo le asignas una referencia a un objeto de la clase **B** (**subclase**) haciendo uso del **polimorfismo**:

A obj ; // Referencia a objetos de la clase A

obj = new B() ; // Referencia a objetos clase A, pero apunta realmente a objeto clase B (polimorfismo)

El objeto que acabas de crear como **instancia de la clase B** (**subclase** de **A**) contiene más información que la que la referencia **obj** te permite en principio acceder sin que el compilador genere un error (pues es de clase **A**). En concreto los objetos de la clase **B** disponen de **atrib1** y **atrib2**, mientras que los objetos de la clase **A** sólo de **atrib1**. Para acceder a esa información adicional de la clase especializada (**atrib2**) tendrás que realizar una **conversión explícita** (**casting**):

// Casting del tipo A al tipo B (funcionará bien porque el objeto es realmente del tipo B)

System.out.printf ("obj.atrib2=%d", ((B) obj).atrib2);

Sin embargo si se hubiera tratado de una **instancia de la clase A** y hubieras intentado acceder al miembro **atrib2**, se habría producido una **excepción** de tipo **ClassCastException**:

A obj; // Referencia a objetos de la clase A

obj = new A() ; // Referencia a objetos de la clase A, y apunta realmente a un objeto de la clase A

// Casting del tipo A al tipo B (puede dar problemas porque el objeto es realmente del tipo A):

// Funciona (la clase A tiene atrib1)

System.out.printf ("obj.atrib1=%d", ((B) obj).atrib1);

// ¡Error en ejecución! (la clase A no tiene atrib2). Producirá una ClassCastException.

System.out.printf ("obj.atrib2=%d", ((B) obj).atrib2);

**En Java, el operador instanceof nos sirve para averiguar si un objeto es de una clase determinada. A continuación tienes un ejemplo resuelto bastante interesante sobre su utilidad.**

Principio del formulario

Final del formulario