

## Caso práctico



En las últimas semanas, **María** y **Juan** han avanzado muchísimo a lo largo de su recorrido por las **estructuras de almacenamiento** y las **interfaces gráficas**, pero **María** es consciente de que aún quedan cosas por ver en lo que respecta a la **Programación Orientada a Objetos**. Aún recuerda que cuando aprendió a escribir sus propias clases, con sus **atributos** y sus **métodos**, se quedaron muchos conceptos sin terminar de aclarar y que serían estudiados más adelante: utilización de la

**herencia**, creación de **interfaces**, **clases abstractas**, **jerarquías de clases**, etc.

Ambos saben que faltan unos cuantos conceptos por asimilar y que sin duda les van a proporcionar más herramientas a la hora de desarrollar sus proyectos. En realidad, muchas de estas nociones ya las han intuido al trabajar con las bibliotecas de clases de la **API** de Java y en cierto modo ya las han utilizado. Parece que ha llegado el momento de formalizar algunos de estos conocimientos para poder emplearlos en sus programas.



**Materiales formativos de FP Online propiedad del Ministerio de Educación, Cultura y Deporte.**

[Aviso Legal](#)

# 1.- Relaciones entre clases.

Cuando estudiaste el concepto de clase, ésta fue descrita como una especie de mecanismo de definición (plantillas), en el que se basaría el entorno de ejecución a la hora de construir un objeto: un **mecanismo de definición de objetos**.

Por tanto, a la hora de diseñar un conjunto de clases para modelar el conjunto de información cuyo tratamiento se desea automatizar, es importante establecer apropiadamente las posibles relaciones que puedan existir entre unas clases y otras.

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** de otra, o bien una **generalización**, 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).

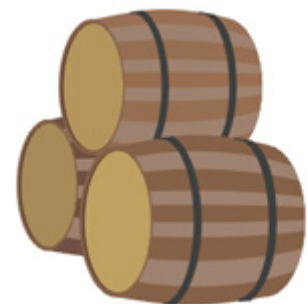


Se pueden distinguir diversos tipos de relaciones entre clases:

- ✓ **Cientela.** 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 **cientela** 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 escribes 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** y **ocultación** de información.

En el caso de la relación de **herencia** también la has visto ya, pues seguro que has utilizado unas clases que derivaban de otras, sobre todo, en el caso de los objetos que forman parte de las **interfaces gráficas**. Lo más probable es que hayas tenido que declarar clases que derivaban de algún componente gráfico (**JFrame**, **JDialog**, etc.).

Podría decirse que tanto la **composición** como la **anidación** son casos particulares de **cientela**, 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.



## Autoevaluación

¿Cuál crees que será la relación entre clases más habitual?

- ☐ Clientela.
- ☐ Anidación o anidamiento.
- ☐ Herencia.
- ☐ Entre las clases no existen relaciones. Son entidades aisladas en el sistema y sin relaciones con el exterior.

## 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.



Por ejemplo, si describes una entidad **País** compuesta por una serie de atributos, entre los cuales se encuentra una lista de comunidades autónomas, podrías decir que los objetos de la clase **País** contienen varios objetos de la clase **ComunidadAutonoma**. Por otro lado, los objetos de la clase **ComunidadAutonoma** podrían contener como atributos objetos de la clase **Provincia**, la cual a su vez también podría contener objetos de la clase **Municipio**.

Como puedes observar, 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.

Recuperando algunos de los ejemplos de clases que has utilizado en otras unidades:

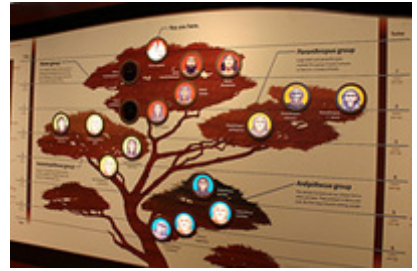
- ✓ Una clase **Rectángulo** podría contener en su interior dos objetos de la clase **Punto** para almacenar los vértices inferior izquierdo y superior derecho.
- ✓ Una clase **Empleado** podría contener en su interior un objeto de la clase **DNI** para almacenar su DNI/NIE, y otro objeto de la clase **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.

### Ejercicio resuelto

¿Podría decirse que la relación que existe entre la clase **Ave** y la clase **Loro** es una relación de composición?

## 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**).
- ✓ Un **cocodrilo** es un **reptil** (heredará atributos como por ejemplo el **número de dientes**).

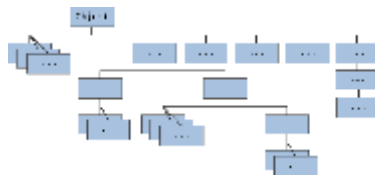
En este caso la expresión idiomática que puedes usar para plantearte si el tipo de relación entre dos clases A y B es de herencia podría ser **"es un"**: "la clase A es un tipo específico de la clase B" (**especialización**), o visto de otro modo: "la clase B es un caso general de la clase A" (**generalización**).

Recuperando algunos ejemplos de clases que ya has utilizado en otras unidades:

- ✓ 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** mas aquellos que tú decidas incorporarle al rellenarlo de componentes gráficos.
- ✓ Una **caja de diálogo** puede ser un tipo de **JDialog** (otro componente **Swing**: `javax.swing.JDialog`).

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.



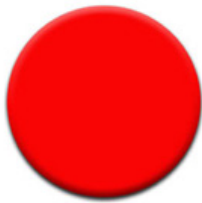
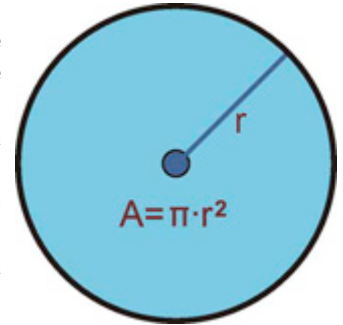
### Ejercicio resuelto

Cuando escribas una clase en Java, puedes hacer que herede de una determinada clase padre (mediante el uso de **extends**) o bien no indicar ninguna herencia. En tal caso tu clase no heredará de ninguna otra clase Java. ¿Verdadero o Falso?

### 1.3.- ¿Herencia o composición?

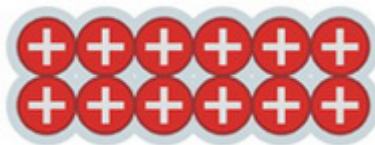
Cuando escribas tus propias clases, debes intentar tener claro en qué casos utilizar la **composición** y cuándo la **herencia**:

- ✓ **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 **Círculo** 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 continuas 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 heredaré 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 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.

### Caso práctico



**María** es consciente de que las relaciones que pueden existir entre dos clases pueden ser de **clientela**, **composición**, **herencia**, etc. También sabe que una forma muy práctica de distinguir entre la necesidad de una composición de un herencia suele ser mediante las preguntas “¿Es la **clase A** un tipo de **clase B**?” o “¿**Tiene la clase A** elementos de la **clase B**”? Normalmente, ese método le suele funcionar a la hora de decidirse por la **composición** o por la **herencia**.

Pero, ¿qué hay que hacer para establecer una relación de composición? ¿Es necesario indicar algún modificador al definir las clases? En tal caso, ¿se indicaría en la clase continente o en la contenida? ¿Afecta de alguna manera al código que hay que escribir? En definitiva, ¿cómo se indica que una clase contiene instancias de otra clase en su interior?

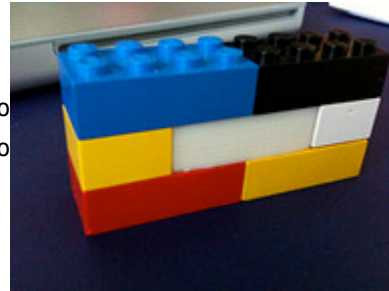
Mientras **María** piensa en voz alta, **Ada** se acerca con una carpeta en la mano y se la entrega: – “Bueno, aquí tienes algunas clases básicas que nos hacen falta para el proyecto de la **Clínica Veterinaria**. A ver qué tal os quedan...”.



## 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> nombreAtributo  
    [modificadores] <NombreClase2> nombreAtributo  
    ...  
}
```



En unidades anteriores has trabajado con la clase **Punto**, que definía las coordenadas de un punto en el plano, y con la clase **Rectángulo**, 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 **Rectángulo** (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**).



### Autoevaluación

Para declarar un objeto de una clase determinada, como atributo de otra clase, es necesario especificar que existe una relación de composición entre ambas clases mediante el modificador **object**. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

### Ejercicio resuelto

Intenta describir los siguientes los métodos de la clase **Rectángulo** teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase **Punto**, en lugar de cuatro elementos de tipo **double**):

1. Método **calcularSuperficie**, que calcula y devuelve el área de la superficie encerrada por la figura.
2. Método **calcularPerimetro**, que calcula y devuelve la longitud del perímetro de la figura.



## 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.

Para entender estas situaciones un poco mejor, podemos volver al objeto **Rectángulo** y observar sus nuevos métodos de tipo **get**.

### Ejercicio resuelto

Dada la clase **Rectángulo**, escribe sus nuevos métodos **obtenerVertice1** y **obtenerVertice2** para que devuelvan los vértices inferior izquierdo y superior derecho del rectángulo (objetos de tipo **Punto**), teniendo en cuenta su nueva estructura de atributos (dos objetos de la clase **Punto**, en lugar de cuatro elementos de tipo **double**):



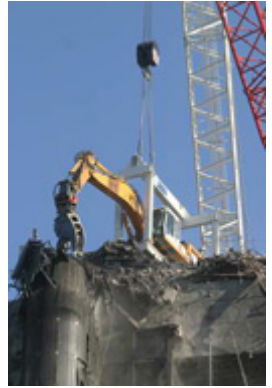
## 2.2.1.- Uso de la composición (II). Llamadas a constructores.

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 el futuro haga uso de él).

Para entender mejor estos posibles efectos podemos continuar con el ejemplo de la clase **Rectángulo** 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).



### Autoevaluación

Si se declaran dos variables objeto **a** y **b** de la clase **X**, ambas son instanciadas mediante un constructor, y posteriormente se realiza la asignación **a=b**, el contenido de **b** será una copia del contenido de **a**, perdiéndose los valores iniciales de **b**. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

### Ejercicio resuelto

Intenta escribir los constructores de la clase Rectángulo teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase Punto, en lugar de cuatro elementos de tipo double):

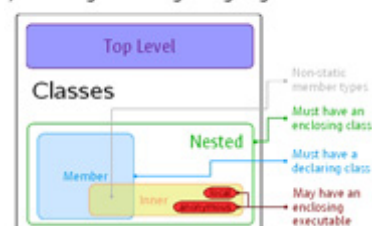
1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros, x1, y1, x2, y2, que cree un rectángulo con los vértices (x1, y1) y (x2, y2).
3. Un constructor con dos parámetros, punto1, punto2, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros, base y altura, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor copia.

## 2.3.- Clases anidadas o internas.

En algunos lenguajes, es posible definir una clase dentro de otra clase (**clases internas**):

```
class claseContenedora {
    // Cuerpo de la clase
    ...
    class claseInterna {
        // Cuerpo de la clase interna
        ...
    }
}
```

Taxonomy of Classes in the Java Programming Language

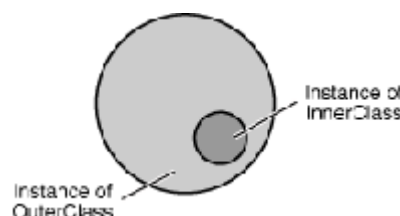


Se pueden distinguir varios tipos de **clases internas**:

- ✓ **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 los **interfaces gráficos**.

Aquí tienes algunos ejemplos:

```
class claseContenedora {
    ...
    static class claseAnidadaEstatica {
        ...
    }
    class claseInterna {
        ...
    }
}
```



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 lo ejemplos con los que vas a trabajar no las vas a necesitar por ahora.

## Para saber más

Si quieres ampliar un poco más sobre las clases internas, puedes echar un vistazo a la siguiente presentación:

[Clases internas.](#) (32 KB)

También puedes consultar el siguiente artículo sobre clases de alto nivel, clases miembro, clases internas y clases anidadas (en inglés):

[Nested Inner, Member, and Top-Level Classes.](#)

### 3.- Herencia.

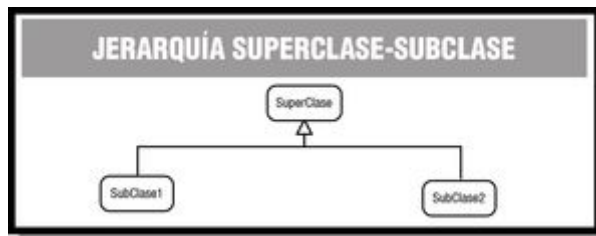
## Caso práctico

**María** ha estado desarrollando junto con **Juan** algunas clases para el proyecto de la **Clínica Veterinaria**. Hasta el momento todo ha ido bien. Han tenido que crear clases que contenían en su interior instancias de otras clases (atributos que eran objetos). Han tenido cuidado con los **constructores** y las **referencias** a los atributos internos y parece que, por ahora, todo funciona perfectamente. Pero ahora necesitan aprovechar algunas de las características que tienen algunas de las clases que ya han escrito, y no quieren tener que volver a escribir todos esos métodos en las nuevas clases. **María** sabe que es una ocasión perfecta para utilizar el concepto de herencia:



- “Si la clase A tiene características en común (atributos y métodos) con la clase B aportando algunas características nuevas, puede decirse que **la clase A es una especialización de la clase B**, ¿no es así?”. – Le pregunta **María** a **Juan**.
- “Así es. Es un caso claro de **herencia**. La clase A hereda de la clase B”. – Contesta **Juan**.
- “De acuerdo. Pues vamos manos a la obra. ¿Cómo indicábamos que una clase heredaba de otra? Creo recordar que se usaba la palabra reservada **extends**. ¿Había que hacer algo más?” – Dice **María** con entusiasmo.
- “Parece que ha llegado el momento de repasar la sintaxis de la **herencia** en Java”.

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**. A la clase que hereda se le suele llamar **clase hija**, **clase derivada** o **subclase**.

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.





## Autoevaluación

Una clase derivada hereda todos los miembros de su clase base, pudiendo acceder a cualquiera de ellos en cualquier momento. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

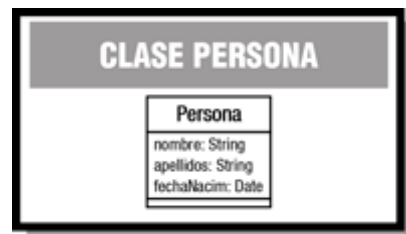
## 3.1.- Sintaxis de la herencia.

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

```
[modificador] class ClasePadre {  
    // Cuerpo de la clase  
    ...  
}  
  
[modificador] class ClaseHija extends ClasePadre {  
    // Cuerpo de la clase  
    ...  
}
```

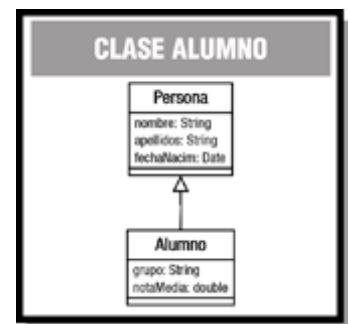
Imagina que tienes una clase **Persona** que contiene atributos como **nombre**, **apellidos** y **fecha de nacimiento**:

```
public class Persona {  
    String nombre;  
    String apellidos;  
    GregorianCalendar fechaNacim;  
    ...  
}
```



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 {  
    String grupo;  
    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 **fechaNacim** (propios de su **clase base Persona** y que por tanto ha heredado).



### Autoevaluación

En Java la herencia se indica mediante la palabra reservada **inherits**. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

## Ejercicio resuelto

Imagina que también necesitas una clase Profesor, que contará con atributos como nombre, apellidos, fecha de nacimiento, salario y especialidad. ¿Cómo crearías esa nueva clase y qué atributos le añadirías?

## 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**.

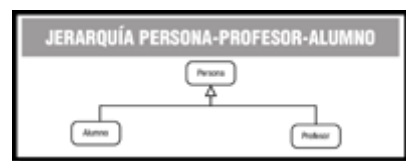
En la unidad dedicada a la utilización de clases ya estudiaste los posibles modificadores de acceso que podía tener un miembro: **sin modificador** (acceso de **paquete**), **público**, **privado** o **protegido**. Aquí tienes de nuevo el resumen:

**Cuadro de niveles accesibilidad a los atributos de una clase**

	Misma clase	Subclase	Mismo paquete	Otro paquete
<b>Sin modificador (paquete)</b>	X		X	X
<b>public</b>	X	X	X	X
<b>private</b>	X			
<b>protected</b>	X	X	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** o bien sin modificador (para que también tengan acceso a ellos otras clases del mismo paquete, si es que se considera oportuno):

```
public class Persona {  
    protected String nombre;  
    protected String apellidos;  
    ...  
}
```

Sólo en aquellos casos en los que se desea explícitamente que un miembro de una clase no pueda ser accesible desde una clase derivada debería utilizarse el modificador **private**. En el resto de casos es recomendable utilizar **protected**, o bien no indicar modificador (acceso a nivel de **paquete**).

## Ejercicio resuelto

Rescribe las clases Alumno y Profesor utilizando el modificador `protected` para sus atributos del mismo modo que se ha hecho para su superclase Persona

### 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**, **fecha de nacimiento**) y otros dos más por ser **Alumno** (**grupo** y **nota media**).



## Ejercicio resuelto

Dadas las clases **Alumno** y **Profesor** que has utilizado anteriormente, implementa métodos **get** y **set** en las clases **Alumno** y **Profesor** para trabajar con sus cinco atributos (tres heredados más dos específicos).

### 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**, **fechaNacim**), 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**, **getFechaNacim**, **setNombre**, **setApellidos**, **setFechaNacim**).
- ✓ Oros 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**.



## Autoevaluación

En Java los métodos heredados de una superclase deben volver a ser definidos en las subclases. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

## Ejercicio resuelto

Dadas las clases **Persona**, **Alumno** y **Profesor** que has utilizado anteriormente, implementa métodos **get** y **set** en la clase **Persona** para trabajar con sus tres atributos y en las clases **Alumno** y **Profesor** para manipular sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para **Persona** van a ser heredados en **Alumno** y en **Profesor**.



### 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 **sobrescritura 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 reescribir ese método para 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 ()
```



#### Autoevaluación

Dado que el método `finalize()` de la clase `Object` es `protected`, el método `finalize()` de cualquier clase que tú escribas podrá ser `public`, `private` o `protected`. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

#### Ejercicio resuelto

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, redefine el método `getNombre` para que devuelva la cadena "Alumno: ", junto con el nombre del alumno, si se trata de un objeto de la clase `Alumno` o bien "Profesor ", junto con el nombre del profesor, si se trata de un objeto de la clase `Profesor`.

### 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:

```
public void mostrar () {  
    super.mostrar ();    // Llamada al método “mostrar” de la superclase  
    // A continuación mostramos la información “especializada” de esta subclase  
    System.out.printf ("Grupo: %s\n", this.grupo);  
    System.out.printf ("Nota media: %5.2f\n", this.notaMedia);  
}
```

Este tipo de **ampliaciones de métodos** resultan especialmente útiles por ejemplo 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**).



## Ejercicio resuelto

Dadas las clases **Persona**, **Alumno** y **Profesor**, define un método **mostrar** para la clase **Persona**, que muestre el contenido de los atributos (datos personales) de un objeto de la clase **Persona**. A continuación, define sendos métodos **mostrar** especializados para las clases **Alumno** y **Profesor** que “amplíen” la funcionalidad del método **mostrar** original de la clase **Persona**.

## 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 de la referencia **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 de la palabra **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: los no heredados. 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 del **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 la referencia **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:

```
public Persona (String nombre, String apellidos, GregorianCalendar fechaNacim) {  
    this.nombre= nombre;  
    this.apellidos= apellidos;  
    this.fechaNacim= new GregorianCalendar (fechaNacim);  
}
```

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

```
public Alumno (String nombre, String apellidos, GregorianCalendar fechaNacim, String gru  
    super (nombre, apellidos, fechaNacim);  
    this.grupo= grupo;  
    this.notaMedia= notaMedia;  
}
```

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



### Autoevaluación

Puede invocarse al constructor de una superclase mediante el uso de la referencia `this`.  
¿Verdadero o Falso?

Verdadero. ☒ Falso. ☐

## Ejercicio resuelto

Escribe un constructor para la clase `Profesor` que realice una llamada al constructor de su clase base para inicializar sus atributos heredados. Los atributos específicos (no heredados) sí deberán ser inicializados en el propio constructor de la clase `Profesor`.

### 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 las utilices 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).



#### Para saber más

Puedes echar un vistazo a este vídeo en el que se muestran algunos ejemplos de utilización de la herencia y clases derivadas. Está estructurado en tres partes:

Tutorial Herencia en Java (Vehic...



[Resumen textual alternativo](#)

#### Ejercicio resuelto

En el siguiente enlace puedes encontrar un ejercicio en el que se propone y resuelve un ejemplo de utilización de la herencia en Java. Consiste en la representación de tres tipos de **trabajadores** (o **funcionarios**, como son llamados en el ejemplo) de una empresa mediante la implementación de tres tipos de clases que deben heredar de una clase base común con la que compartirán algunos miembros (por el hecho de ser trabajadores). Los tres tipos de trabajadores son: **Programador**, **Analista** e **Ingeniero**.

Aquí tienes el enlace al primer vídeo en el que se propone y resuelve el ejemplo (está estructurado en cinco vídeos enlazados):

## Ejercicio Herencia (Mandiosoft 1...



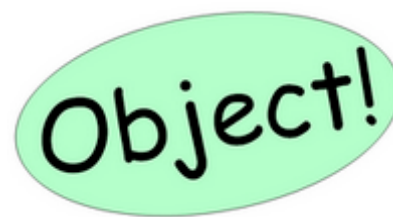
[Resumen textual alternativo](#)



### 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.

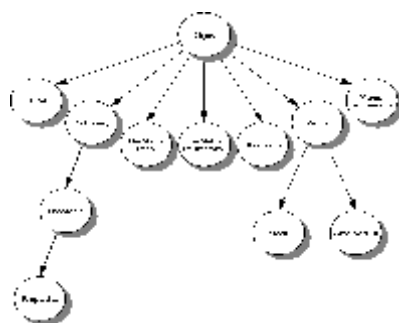


Entre los métodos que incorpora la clase **Object** y que por tanto hereda cualquier clase en Java tienes:

#### Principales métodos de la clase Object

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

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.



### Para saber más

Para obtener más información sobre la clase **Object**, sus métodos y propiedades, puedes consultar la documentación de la [API](#) de **Java** en el sitio web de Oracle.

[Documentación de la clase Object.](#)



### Autoevaluación

**Toda clase Java tiene un método toString y un método finalize. ¿Verdadero o Falso?**

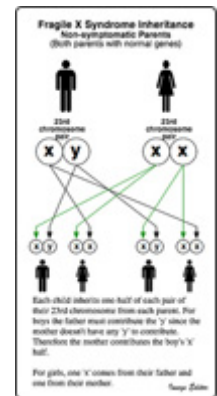
Verdadero. ☐ Falso. ☐

### 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, en tal caso, ¿qué miembro se hereda? 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.

### Caso práctico

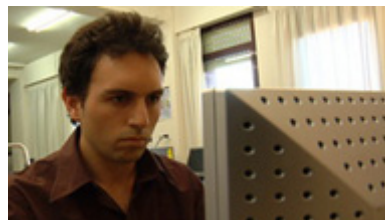
**María** está desarrollando nuevas clases para el proyecto de la **Clínica Veterinaria** y se ha dado cuenta de que, para aprovechar toda la potencia de la **herencia**, le vendría bien definir algunas clases que, sin embargo, nunca va a llegar a instanciar: – “¡Qué raro! Estoy definiendo una clase de la que nunca voy a tener objetos. Voy a instanciar a sus **subclases**, pero nunca a la **superclase**...”. – Piensa **María** en voz alta.

– “¡No te preocupes! En realidad no es tan raro...”. – Le contesta **Juan**, que acaba de llegar.

– “¿Ah no? ¿Tú crees?”.

– “Totalmente. Es más habitual de lo que piensas. Son las **clases abstractas**. Y si has llegado tú misma a la conclusión de que las necesitas, mejor todavía, pues vas a entender con más facilidad su funcionamiento.”.

– “¡Genial! Cuéntame más...”.



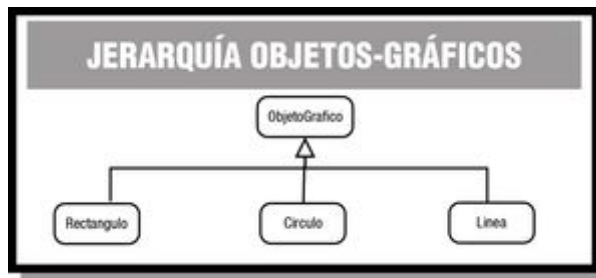
En determinadas ocasiones, es posible que necesites definir una clase que represente un concepto lo suficientemente abstracto como para que nunca vayan a existir instancias de ella (objetos). ¿Tendría eso sentido? ¿Qué utilidad podría tener?

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 **implementación 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**.



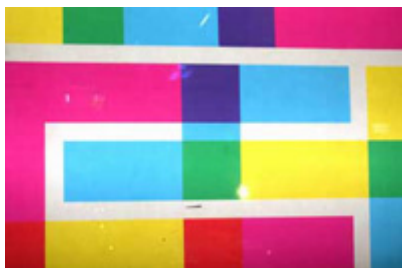
## Autoevaluación

Una clase abstracta no podrá ser nunca instanciada. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

## 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 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)**.



### Autoevaluación

Puede llamarse al constructor de una clase abstracta mediante el operador new. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

### Ejercicio resuelto

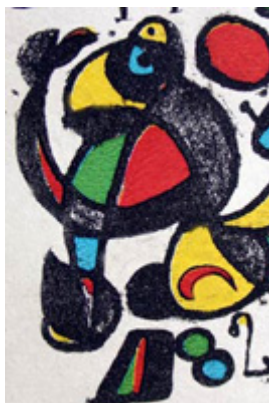
Basándote en la jerarquía de clases de ejemplo (Persona, Alumno, Profesor), que ya has utilizado en otras ocasiones, modifica lo que consideres oportuno para que Persona sea, a partir de ahora, una clase abstracta (no instanciable) y las otras dos clases sigan siendo clases derivadas de ella, pero sí instanciables.

Localiza en la [API](#) de Java algún ejemplo de **clase abstracta**.



## 4.2.- Métodos abstractos.

Un **método abstracto** 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**.



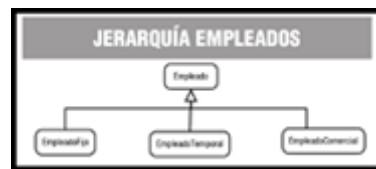
Un método se declara como abstracto mediante el uso del modificador **abstract** (como en las **clases abstractas**):

```
[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).

### Para saber más

Puedes echar un vistazo a este vídeo en el se explica el funcionamiento de las **clases abstractas** y se muestra un ejemplo de creación y utilización:

## Video Tutorial Clases Abstractas



[Resumen textual alternativo](#)



## Autoevaluación

Los métodos de una clase abstracta tienen que ser también abstractos. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

## Ejercicio resuelto

Basándote en la jerarquía de clases Persona, Alumno, Profesor, crea un método abstracto llamado mostrar para la clase Persona. Dependiendo del tipo de persona (alumno o profesor) el método mostrar tendrá que mostrar unos u otros datos personales (habrá que hacer implementaciones específicas en cada clase derivada).

Una vez hecho esto, implementa completamente las tres clases (con todos sus atributos y métodos) y utilízalas en un pequeño programa de ejemplo que cree un objeto de tipo Alumno y otro de tipo Profesor, los rellene con información y muestre esa información en la pantalla a través del método mostrar.

## 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**.



### Autoevaluación

Los modificadores **final** y **abstract** son excluyentes en la declaración de un **método**. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

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, final int par2)**.

### Debes conocer

Dada la gran cantidad de contextos diferentes en los que se puede encontrar el modificador **final**, vale la pena que hagas un repaso de todos los lugares donde puede aparecer y cuál sería su función en cada uno:

[Contextos del modificador final.](#)

## 5.- Interfaces.

### Caso práctico

**María y Juan** continúan con su tarea de desarrollo de clases para el proyecto de la **Clínica Veterinaria**. Ya han utilizado la **herencia** en diversas ocasiones e incluso han escrito alguna **clase abstracta** para luego generar clases especializadas basadas en ella.



Pero ahora se les ha planteado un nuevo problema: tienen pensadas algunas clases entre las que no existe ninguna relación de **herencia**, cada una hereda de unos ancestros diferentes que no tienen nada que ver, pero, sin embargo, sí podrían compartir una buena parte de sus **comportamientos (métodos)**. No es posible hacer que las dos hereden de la misma **clase base** porque hemos dicho que no se parecen en nada a ese respecto (cada una tiene su **clase base**, sin relación entre ellas), y tampoco pueden heredar de una nueva **clase abstracta** que contenga la interfaz de ese comportamiento, pues la **herencia múltiple** no está permitida en Java:

- "¿Qué hacemos entonces?, ¿repetimos la misma **interfaz** en las dos **jerarquías** de clases? No me cuadra tener que hacer eso...". - Le pregunta **María** a **Juan**.

- "A mí tampoco. No me suena muy bien". - Le contesta **Juan**.

- "Así es. Tiene que haber una **solución** más elegante que no nos haga tener que repetir ese código una y otra vez. ¿Alguna idea?".

- "Quizá exista una forma de resolver el problema. ¿Recuerdas que **Ada** nos habló el otro día de las **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 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.

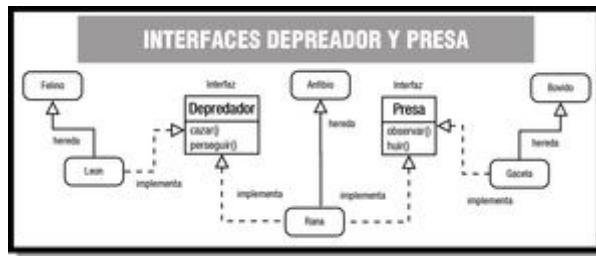
También has estudiado cómo los **métodos abstractos** permiten establecer una **interfaz** para marcar las **líneas generales de un comportamiento común de superclase** que deberían compartir de todas las **subclases**.

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** (por ejemplo: **observar** una **presa**, **perseguirla**, **comérsela**, etc.) o sean **presas** (**observar**, **huir**, **escondersela**, etc.). Si creas la clase **León**, 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**.

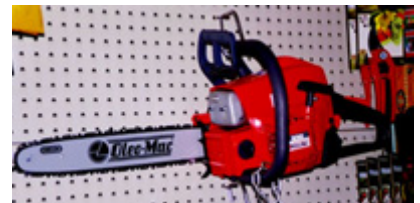
En cierto modo podrías imaginar el concepto de **interfaz** como un **guión** que dice: "éste es el protocolo de comunicación que deben presentar todas las clases que implementen esta interfaz". Se proporciona una lista de **métodos públicos** y, si quieres dotar a tu clase de esa **interfaz**, tendrás 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" y cosas del estilo, que significan algo así como **capacidad o habilidad** para hacer o ser receptores de algo (**configurable, serializable, modificable, clonable, ejecutable, administrador, servidor, buscador**, 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 porqué 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.



### Autoevaluación

Una interfaz en Java no puede contener la implementación de un método mientras que una clase abstracta sí. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

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

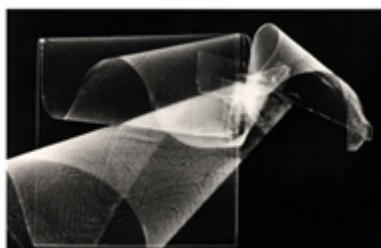
Observando el concepto de **interfaz** que se acaba de proponer, podría caerse en la tentación de pensar que es prácticamente lo mismo que una **clase abstracta** en la que **todos sus métodos sean abstractos**.

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 sólo 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**.





## Autoevaluación

En Java una clase no puede heredar de más de una clase abstracta ni implementar más de una interfaz. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

## 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.

### Resumen textual alternativo

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 **Depredador** que 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 (**León**, **Leopardo**, **Cocodrilo**, **Rana**, **Lagarto**, **Hombre**, etc.) las que definan cada uno de los métodos por dentro.



## Autoevaluación

Los métodos de una interfaz en Java tienen que ser obligatoriamente declarados como **public** y **abstract**. Si no se indica así, se producirá un error de compilación. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

## Ejercicio resuelto

Crea una interfaz en Java cuyo nombre sea **Imprimible** y que contenga algunos métodos útiles para mostrar el contenido de una clase:

1. Método **devolverContenidoString**, que crea un `String` con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves: "{<nombre\_atributo\_1>=<valor\_atributo\_1>, ..., <nombre\_atributo\_n>=<valor\_atributo\_n>}".
2. Método **devolverContenidoArrayList**, que crea un `ArrayList` de `String` con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve.
3. Método **devolverContenidoHashtable**, similar al anterior, pero en lugar de devolver en un `ArrayList` los valores de los atributos, se devuelve en una `Hashtable` en forma de pares (nombre, valor).

## 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 **León**, 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.).



### Autoevaluación

¿Qué palabra reservada se utiliza en Java para indicar que una clase va a definir los métodos indicados por una interfaz?

- ☐ implements.
- ☐ uses.
- ☐ extends.
- ☐ Los métodos indicados por una **interfaz** no se definen en las clases pues sólo se pueden utilizar desde la propia **interfaz**.

## Ejercicio resuelto

Haz que las clases **Alumno** y **Profesor** implementen la interfaz **Imprimible** que se ha escrito en el ejercicio anterior.

### 5.3.1.- Un ejemplo de implementación de interfaces: la interfaz ActionListener.

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 APIs. 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 **ActionListener**.



La **interfaz ActionListener** ya la has utilizado en la unidad dedicada a las **interfaces gráficas**. Las clases que quieran realizar una determinada acción cada vez que se produzca cierto **evento** en el sistema deben implementar esta **interfaz**. 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** durante la ejecución de tu aplicación (pulsación de una tecla, clic de ratón, etc.). Son intensivamente utilizadas en el desarrollo de las **interfaces gráficas de usuario**.

## Recomendación

Es interesante que repases los conceptos vistos en el **modelo de gestión de eventos** explicado en la unidad dedicada las **interfaces gráficas: evento, oyente, gestión de eventos**, etc.

## Para saber más

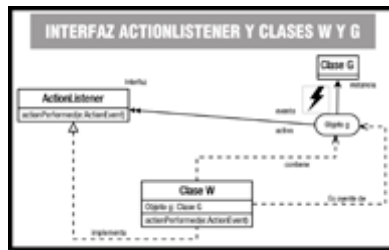
Si deseas profundizar algo más en el tema de los Event Listeners puedes echar un vistazo a los tutoriales de iniciación de Java en el sitio web de Oracle (en inglés) en el que se explican detalladamente ejemplos sencillos así como otros bastante más complejos:

[Introduction to Event Listeners.](#)

Vamos a ver un ejemplo lo más sencillo posible: se trata de desarrollar una pequeña aplicación de escritorio (utilizando la API de **Swing**) con una **ventana** que contenga un par de **botones** y que al pulsar alguno de esos **botones (evento)**, la aplicación sea capaz de detectar que se ha producido ese **evento** y por tanto realizar una determinada **acción** (por ejemplo generar un sonido o mostrar un determinado texto).

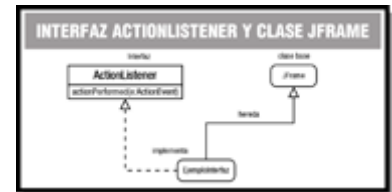
Para ello es necesario que la clase que vaya a gestionar el evento sea un "**oyente**" de ese evento, es decir, que sea capaz de enterarse de que se ha producido ese evento. Esa capacidad o habilidad la proporciona la API de Java a través de las interfaces de tipo "**oyente**". En concreto vamos a implementar la **interfaz ActionListener**.

La **interface ActionListener** contiene un único método: **actionPerformed**. Toda clase **W** que implemente la esta interfaz tendrá que definir ese método. Si incrustamos un determinado objeto gráfico **G** en una ventana **W** y asociamos a ese objeto gráfico la clase **W**, indicando que es su **oyente y gestor de eventos**, cuando se produzca algún evento sobre **G** se ejecutará el método **actionPerformed** implementado por **W**.



En nuestro ejemplo podríamos hacer lo siguiente:

- ✓ Definir una **ventana principal**, por ejemplo una clase basada en el tipo **marco** o **frame** (subclase de JFrame) que implemente la **interfaz** ActionListener.
- ✓ Definir uno o varios **botones** (objetos de la clase JButton) dentro de la ventana.
- ✓ Asociar como **oyente** a los objetos de tipo **botón** la propia **ventana principal (frame)**, pues va a ser capaz de gestionar **eventos** (implementa el método actionPerformed).
- ✓ Implementar el método actionPerformed como un **método de la ventana principal (frame)** para **reaccionar** de algún modo cuando se produzca algún **evento desencadenado por un botón** (pues se ha establecido a la ventana principal como **oyente** y **gestora de los eventos** del botón).



En la siguiente presentación puedes observar detalladamente el proceso completo para llevar a cabo este ejemplo de implementación de la interfaz **ActionListener**:

[Resumen textual alternativo](#)

Desde el siguiente enlace puedes descargar el ejemplo completo en el que se implementa la interfaz **ActionListener** en un pequeño programa que crea una ventana que gestiona dos botones: uno de ellos produce un pitido y el otro produce dos pitidos.

[Proyecto EjemploInterfazActionListener.](#) (19 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 tienen **un valor de retorno 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 ésta).

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.



### Autoevaluación

Dada una clase Java que implementa dos interfaces diferentes que contienen un método con el mismo nombre, indicar cuál de las siguientes afirmaciones es correcta.





Si los dos métodos tienen **un valor de retorno de un tipo diferente**, se producirá un **error de compilación**.

- ☐ Si los dos métodos tienen **un valor de retorno de un tipo diferente**, se **implementarán dos métodos**.
- ☐ Si los dos métodos son **exactamente iguales en identificador, parámetros y tipo devuelto**, se producirá un **error de compilación**.
- ☐ Si los dos métodos tienen **diferentes parámetros** se producirá un **error de compilación**.

## Ejercicio resuelto

Localiza en la API de Java algún ejemplo de clase que implemente varias interfaces diferentes (puedes consultar la documentación de referencia de la API de Java).

## 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 {  
  
    // Métodos y constantes de la interfaz compleja  
  
}
```



### Autoevaluación

En Java no está permitida la herencia múltiple ni para clases ni para interfaces. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

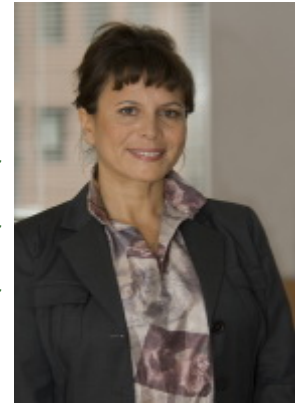
### Ejercicio resuelto

Localiza en la API de Java algún ejemplo de interfaz que herede de una o varias interfaces (puedes consultar la documentación de referencia de la API de Java).

## 6.- Polimorfismo.

### Caso práctico

**María** está desarrollando algunas clases que representan categorías de **animales** para el proyecto de la **Clínica Veterinaria**. En algunos casos declara objetos de un tipo de animal y en ciertas ocasiones de otros, según las necesidades que tenga en cada momento. La clase **Animal** es demasiado genérica como para poder utilizarla en determinados casos y necesita clases más específicas para poder trabajar pues tendrá que usar unas u otras versiones de sus métodos. **Juan** también está haciendo algo parecido y ambos intuyen que el código que están escribiendo podría ser mucho más sencillo y flexible si pudieran declarar inicialmente objetos de la clase **Animal** y más tarde, durante la ejecución de la aplicación, utilizar objetos de tipo **Animal**, pero de clases más especializadas (**subclases de Animal**) en función de lo que suceda en cada momento. Sería muy interesante poder hacer algo así. **Ada** lleva algunos minutos escuchándolos y decide intervenir:



- "Veo que habéis llegado a la conclusión de que necesitáis trabajar con objetos cuya clase aún no está clara en tiempo de compilación, ¿no?". - Les pregunta a ambos.

- "Así es. Pero eso no se puede hacer, el compilador no nos lo va a permitir". - Le responden casi al unísono.

- "Bueno, es lógico que el compilador tenga que saber a qué clase pertenece un objeto para poder analizar si se está accediendo a los miembros correctos y con la sintaxis apropiada, ¿no crees?". - Les vuelve a preguntar.

- "Totalmente de acuerdo.". - Le contesta **María**.

- "Pero si declaramos un objeto de una clase que sea **superclase** de otras, quizá podríamos más tarde intentar para ese objeto instanciar una **subclase** más específica. Al fin y al cabo, una clase de tipo **MascotaDoméstica** sigue siendo también **Animal**, pues ha heredado de ella, ¿no es así?".

- "¿Quieres decir que podríamos utilizar en el programa objetos de clases cuyos métodos llamados no sabemos exactamente cuáles van a ser porque dependerá de la **subclase** concreta que se instancie en tiempo de ejecución?". - Le responde **María** muy interesada.

- "Parece que ha llegado el momento de que empecéis a trabajar con el polimorfismo y la [ligadura dinámica](#)". - Les contesta satisfecha.



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 posible 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.



## Autoevaluación

¿Cuál de las siguientes características dirías que no es una de las que se suelen considerar como uno de los tres grandes pilares de la Programación Orientada a Objetos?

- ☐ Recursividad.
- ☐ Herencia.
- ☐ Polimorfismo.
- ☐ Encapsulación.

## 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 subclases**. 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 en durante la **ejecución** podrá ser de tipo **A** o de tipo **B** (algo desconocido en **tiempo de compilación**). 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**). 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 variable obj adquiere la forma de la subclase A.
obj = 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.
obj = 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).
```

...

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 **devolverContenidoString**) 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**.



## Autoevaluación

El polimorfismo ofrece la posibilidad de que toda referencia a un objeto de una clase A pueda tomar la forma de una referencia a un objeto de cualquier otra clase B. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

## 6.2.- Ligadura dinámica.

La conexión que tiene lugar durante una llamada a un método suele ser llamada **ligadura**, **vinculación** o **enlace** (en inglés **binding**). Si esta **vinculación** se lleva a cabo durante el proceso de compilación, se le suele llamar **ligadura estática** (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** (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)
```

### Ejercicio resuelto

Imaginate una clase que represente a **instrumento musical** genérico (**Instrumento**) y dos subclases que representen tipos de instrumentos específicos (por ejemplo **Flauta** y **Piano**). Todas las clases tendrán un método **tocarNota**, que será específico para cada subclase.

Haz un pequeño programa de ejemplo en Java que utilice el **polimorfismo** (referencias a la **superclase** que se convierten en instancias específicas de **subclases**) y la **ligadura dinámica** (llamadas a un método que aún no están resueltas en **tiempo de compilación**) con estas clases que representan instrumentos musicales. Puedes implementar el método **tocarNota** mediante la escritura de un mensaje en pantalla.

## 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**).

### Ejercicio resuelto

Haz un pequeño programa en Java en el que se declare una variable de tipo **Persona**, se pidan algunos datos sobre esa persona (**nombre**, **apellidos** y si es **alumno** o si es **profesor**), y se muestren nuevamente esos datos en pantalla, teniendo en cuenta que esa variable no puede ser instanciada como un objeto de tipo **Persona** (es una **clase abstracta**) y que tendrás que instanciarla como **Alumno** o como **Profesor**. Recuerda que para poder recuperar sus datos necesitarás hacer uso de la **ligadura dinámica** y que tan solo deberías acceder a métodos que sean de la **superclase**.



## 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 utilizarse de 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 podrías 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 podrías 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 obj; // 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:
obj= new Alumno (nombre, apellidos, fecha, grupo, nota); // Polimorfismo con interfaces
...
// En otras circunstancias podría suceder esto:
obj= 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= obj.devolverContenidoString(); // Ligadura dinámica con interfaces
```



### Autoevaluación

El polimorfismo puede hacerse con referencias de superclases abstractas, superclases no abstractas o con interfaces. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐



## 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 (
```

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
System.out.printf ("obj.atrib2=%d\n", ((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
```



```
// Casting del tipo A al tipo B (puede dar problemas porque el objeto es realmente del 1

// Funciona (la clase A tiene atrib1)
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib1);

// ¡Error en ejecución! (la clase A no tiene atrib2). Producirá una ClassCastException.
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib2);
```



# Anexo I.- Elaboración de los constructores de la clase Rectangulo.

---

## ENUNCIADO

Intenta describir los constructores de la clase **Rectangulo** teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase **Punto**, en lugar de cuatro elementos de tipo **double**):

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros, **x1**, **y1**, **x2**, **y2**, que cree un rectángulo con los vértices (x1, y1) y (x2, y2).
3. Un constructor con dos parámetros, **punto1**, **punto2**, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros, **base** y **altura**, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor copia.

## POSIBLE SOLUCIÓN

Durante el proceso de creación de un objeto (**constructor**) de la **clase contenedora** (en este caso **Rectangulo**) hay que tener en cuenta también la creación (llamada a **constructores**) de aquellos objetos que son contenidos (en este caso objetos de la clase **Punto**).

En el caso del primer **constructor**, habrá que crear dos **puntos** con las coordenadas (0,0) y (1,1) y asignarlos a los atributos correspondientes (**vertice1** y **vertice2**):

```
public Rectangulo ()
{
    this.vertice1= new Punto (0,0);
    this.vertice2= new Punto (1,1);
}
```

Para el segundo **constructor** habrá que crear dos puntos con las coordenadas **x1**, **y1**, **x2**, **y2** que han sido pasadas como parámetros:

```
public Rectangulo (double x1, double y1, double x2, double y2)
{
    this.vertice1= new Punto (x1, y1);
    this.vertice2= new Punto (x2, y2);
}
```

En el caso del tercer **constructor** puedes utilizar directamente los dos puntos que se pasan como parámetros para construir los vértices del rectángulo:

Ahora bien, esto podría ocasionar un **efecto colateral** no deseado si esos objetos de tipo **Punto** son modificados en el futuro desde el código cliente del **constructor** (no sabes si esos puntos fueron creados especialmente para ser usados por el rectángulo o si pertenecen a otro objeto que podría modificarlos más tarde).

Por tanto, para este caso quizá fuera recomendable crear dos nuevos puntos a imagen y semejanza de los puntos que se han pasado como parámetros. Para ello tendrías dos opciones:

1. Llamar al **constructor** de la clase **Punto** con los valores de los atributos (x, y).
2. Llamar al **constructor copia** de la clase **Punto**, si es que se dispone de él.

Aquí tienes las dos posibles versiones:

**Constructor** que “extrae” los atributos de los parámetros y crea nuevos objetos:

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

Constructor que crea los nuevos objetos mediante el **constructor copia** de los parámetros:

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

En este segundo caso puedes observar la utilidad de los **constructores de copia** a la hora de tener que **clonar** objetos (algo muy habitual en las inicializaciones).

Para el caso del **constructor** que recibe como parámetros la base y la altura, habrá que crear sendos vértices con valores (0,0) y (0 + base, 0 + altura), o lo que es lo mismo: (0,0) y (base, altura).

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

Quedaría finalmente por implementar el **constructor copia**:

```
// Constructor copia
public Rectangulo (Rectangulo r) {
    this.vertice1= new Punto (r.obtenerVertice1() );
    this.vertice2= new Punto (r.obtenerVertice2() );
}
```

En este caso nuevamente volvemos a **clonar** los atributos **vertice1** y **vertice2** del objeto **r** que se ha pasado como parámetro para evitar tener que compartir esos atributos en los dos rectángulos.

## Anexo II.- Métodos para las clases heredadas Alumno y Profesor.

---

### ENUNCIADO

Dadas las clases **Alumno** y **Profesor** que has utilizado anteriormente, implementa métodos **get** y **set** en las clases **Alumno** y **Profesor** para trabajar con sus cinco atributos (tres heredados más dos específicos).

### POSIBLE SOLUCIÓN

#### 1. Clase Alumno.

Se trata de heredar de la clase **Persona** y por tanto utilizar con normalidad sus atributos heredados como si pertenecieran a la propia clase (de hecho se puede considerar que le pertenecen, dado que los ha heredado).

```
public class Alumno extends Persona {
    protected String grupo;
    protected double notaMedia;

    // Método getNombre
    public String getNombre (){
        return nombre;
    }

    // Método getApellidos
    public String getApellidos (){
        return apellidos;
    }

    // Método getFechaNacim
    public GregorianCalendar getFechaNacim (){
        return this.fechaNacim;
    }

    // Método getGrupo
    public String getGrupo (){
        return grupo;
    }

    // Método getNotaMedia
    public double getNotaMedia (){
        return notaMedia;
    }

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

    // Método setApellidos
    public void setApellidos (String apellidos){
        this.apellidos= apellidos;
    }
}
```

```

    }

    // Método setFechaNacim
    public void setFechaNacim (GregorianCalendar fechaNacim){
        this.fechaNacim= fechaNacim;
    }

    // Método setGrupo
    public void setGrupo (String grupo){
        this.grupo= grupo;
    }

    // Método setNotaMedia
    public void setNotaMedia (double notaMedia){
        this.notaMedia= notaMedia;
    }
}

```

Si te fijas, puedes utilizar sin problema la referencia **this** a la propia clase con esos atributos heredados, pues pertenecen a la clase: **this.nombre**, **this.apellidos**, etc.

## 2. Clase Profesor.

Seguimos exactamente el mismo procedimiento que con la clase **Alumno**.

```

public class Profesor extends Profesor {
    String especialidad;
    double salario;

    // Método getNombre
    public String getNombre (){
        return nombre;
    }

    // Método getApellidos
    public String getApellidos (){
        return apellidos;
    }

    // Método getFechaNacim
    public GregorianCalendar getFechaNacim (){
        return this.fechaNacim;
    }

    // Método getEspecialidad
    public String getEspecialidad (){
        return especialidad;
    }

    // Método getSalario
    public double getSalario (){

```



```

        return salario;
    }

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

    // Método setApellidos
    public void setApellidos (String apellidos){
        this.apellidos= apellidos;
    }

    // Método setFechaNacim
    public void setFechaNacim (GregorianCalendar fechaNacim){
        this.fechaNacim= fechaNacim;
    }

    // Método setSalario
    public void setSalario (double salario){
        this.salario= salario;
    }

    // Método setESpecialidad
    public void setESpecialidad (String especialidad){
        this.especialidad= especialidad;
    }
}

```

Una conclusión que puedes extraer de este código es que has tenido que escribir los métodos **get** y **set** para los tres atributos heredados, pero ¿no habría sido posible definir esos seis métodos en la clase base y así estas dos clases derivadas hubieran también heredado esos métodos? La respuesta es afirmativa y de hecho es como lo vas a hacer a partir de ahora. De esa manera te habrías evitado tener que escribir seis métodos en la clase **Alumno** y otros seis en la clase **Profesor**. Así que recuerda: **se pueden heredar tanto los atributos como los métodos**.

Aquí tienes un ejemplo de cómo podrías haber definido la clase **Persona** para que luego se hubieran podido heredar de ella sus métodos (y no sólo sus atributos):

```

public class Persona {
    protected String nombre;
    protected String apellidos;
    protected GregorianCalendar fechaNacim;

    // Método getNombre
    public String getNombre (){
        return nombre;
    }

    // Método getApellidos
    public String getApellidos (){
        return apellidos;
    }
}

```

```
// Método getFechaNacim
public GregorianCalendar getFechaNacim (){
    return this.fechaNacim;
}

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

// Método setApellidos
public void setApellidos (String apellidos){
    this.apellidos= apellidos;
}

// Método setFechaNacim
public void setFechaNacim (GregorianCalendar fechaNacim){
    this.fechaNacim= fechaNacim;
}

}
```

## Anexo III.- Métodos para los atributos de las clases Alumno y Profesor.

---

### ENUNCIADO

Dadas las clases **Persona**, **Alumno** y **Profesor** que has utilizado anteriormente, implementa métodos **get** y **set** en la clase **Persona** para trabajar con sus tres atributos y en las clases **Alumno** y **Profesor** para trabajar con sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para **Persona** van a ser heredados en **Alumno** y en **Profesor**.

### POSIBLE SOLUCIÓN

#### 1. Clase Persona.

```
public class Persona {
    protected String nombre;
    protected String apellidos;
    protected GregorianCalendar fechaNacim;

    // Método getNombre
    public String getNombre (){
        return nombre;
    }

    // Método getApellidos
    public String getApellidos (){
        return apellidos;
    }

    // Método getFechaNacim
    public GregorianCalendar getFechaNacim (){
        return this.fechaNacim;
    }

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

    // Método setApellidos
    public void setApellidos (String apellidos){
        this.apellidos= apellidos;
    }

    // Método setFechaNacim
    public void setFechaNacim (GregorianCalendar fechaNacim){
        this.fechaNacim= fechaNacim;
    }
}
```

#### 2. Clase Alumno.

Al heredar de la clase **Persona** tan solo es necesario escribir métodos para los nuevos atributos (**métodos especializados** de acceso a los **atributos especializados**), pues los **métodos genéricos** (de acceso a los **atributos genéricos**) ya forman parte de la clase al haberlos heredado.

```
public class Alumno extends Persona {
    protected String grupo;
    protected double notaMedia;

    // Método getGrupo
    public String getGrupo (){
        return grupo;
    }

    // Método getNotaMedia
    public double getNotaMedia (){
        return notaMedia;
    }

    // Método setGrupo
    public void setGrupo (String grupo){
        this.grupo= grupo;
    }

    // Método setNotaMedia
    public void setNotaMedia (double notaMedia){
        this.notaMedia= notaMedia;
    }

}
```

Aquí tienes una demostración práctica de cómo la herencia permite una reutilización eficiente del código, evitando tener que repetir atributos y métodos. Sólo has tenido que escribir cuatro métodos en lugar de diez.

### 3. Clase Profesor.

Seguimos exactamente el mismo procedimiento que con la clase **Alumno**.

```
public class Profesor extends Profesor {
    String especialidad;
    double salario;

    // Método getEspecialidad
    public String getEspecialidad (){
        return especialidad;
    }

    // Método getSalario
    public double getSalario (){
        return salario;
    }
}
```

```
        // Método setSalario
        public void setSalario (double salario){
            this.salario= salario;
        }

        // Método setEspecialidad
        public void setEspecialidad (String especialidad){
            this.especialidad= especialidad;
        }
    }
```

## Anexo IV.- Contextos del modificador final.

### Distintos contextos en los que puede aparecer el modificador final

Lugar	Función
Como modificador de clase.	La clase no puede tener subclases.
Como modificador de atributo.	El atributo no podrá ser modificado una vez que tome un valor. Sirve para definir constantes.
Como modificador al declarar un método	El método no podrá ser redefinido en una clase derivada.
Como modificador al declarar una variable referencia.	Una vez que la variable tome un valor referencia (un objeto), no se podrá cambiar. La variable siempre apuntará al mismo objeto, lo cual no quiere decir que ese objeto no pueda ser modificado internamente a través de sus métodos. Pero la variable no podrá apuntar a otro objeto diferente.
Como modificador en un parámetro de un método.	El valor del parámetro (ya sea un tipo primitivo o una referencia) no podrá modificarse dentro del código del método.

Veamos un ejemplo de cada posibilidad:

#### 1. Modificador de una clase.

```
public final class ClaseSinDescendencia {    // Clase "no heredable"
    ...
}
```

#### 2. Modificador de un atributo.

```
public class ClaseEjemplo {
    // Valor constante conocido en tiempo de compilación
    final double PI= 3.14159265;

    // Valor constante conocido solamente en tiempo de ejecución
    final int SEMILLA= (int) Math.random()*10+1;
    ...
}
```

#### 3. Modificador de un método.

```
public final metodoNoRedefinible (int parametro1) {    // Método "no redefinible"
```

```
    ...  
}
```

#### 4. Modificador en una variable referencia.

```
// Referencia constante: siempre se apuntará al mismo objeto Alumno recién creado,  
// aunque este objeto pueda sufrir modificaciones.  
final Alumno PRIMER_ALUMNO= new Alumno ("Pepe", "Torres", 9.55); // Ref. constan  
  
// Si la variable no es una referencia (tipo primitivo), sería una constante más  
// (como un atributo constante).  
final int NUMERO_DIEZ= 10; // Valor constante (dentro del ámbito de vida de la
```

#### 5. Modificador en un parámetro de un método.

```
void metodoConParametrosFijos (final int par1, final int par2) {  
    // Los parámetros "par1" y "par2" no podrán sufrir modificaciones aquí dentro  
    ...  
}
```



## Anexo.- Licencias de recursos.

### Licencias de recursos utilizados en la Unidad

Recurso (1)	Datos del recurso (1)	Recurso (2)
	Autoría: s_volenszki. Licencia: CC BY-NC. Procedencia: <a href="http://www.flickr.com/photos/s_volenszki/2218589271/">http://www.flickr.com/photos/s_volenszki/2218589271/</a>	
	Autoría: Oracle. Licencia: Copyright (cita). Procedencia: <a href="http://docs.oracle.com/javase/tutorial/figures/java/classes-object.gif">http://docs.oracle.com/javase/tutorial/figures/java/classes-object.gif</a>	
	Autoría: aplumb. Licencia: CC BY-SA 2.0. Procedencia: <a href="http://www.flickr.com/photos/aplumb/3890010497/">http://www.flickr.com/photos/aplumb/3890010497/</a>	
	Autoría: Oracle. Joseph D. Darcy. Licencia: Copyright (cita). Procedencia: <a href="http://blogs.oracle.com/darcy/entry/nested_inner_member_and_top">http://blogs.oracle.com/darcy/entry/nested_inner_member_and_top</a>	
	Autoría: Diosdado Sánchez Hernández. Licencia: GNU GPL. Procedencia: Captura de pantalla del programa NetBeans, propiedad de Oracle, bajo licencia GNU GPL v2.	
	Autoría: ethorson. Licencia: CC BY-SA. Procedencia: <a href="http://www.flickr.com/photos/ethorson/122310358/in/photostream/">http://www.flickr.com/photos/ethorson/122310358/in/photostream/</a>	
	Autoría: Image Editor. Licencia: CC BY 2.0. Procedencia: <a href="http://www.flickr.com/photos/11304375@N07/3197825319/">http://www.flickr.com/photos/11304375@N07/3197825319/</a>	
	Autoría: Dalekwidow. Licencia: CC BY 2.0. Procedencia: <a href="http://www.flickr.com/photos/dalekwidow/4297404633/">http://www.flickr.com/photos/dalekwidow/4297404633/</a>	
	Autoría: Alessandro Pinna. Licencia: CC BY-NC-SA 2.0.	





Procedencia:  
<http://www.flickr.com/photos/alessandropinna/1423128740/>



Autoría: pobre.ch.  
Licencia: CC BY 2.0.  
Procedencia: <http://www.flickr.com/photos/npobre/2601582256/>

