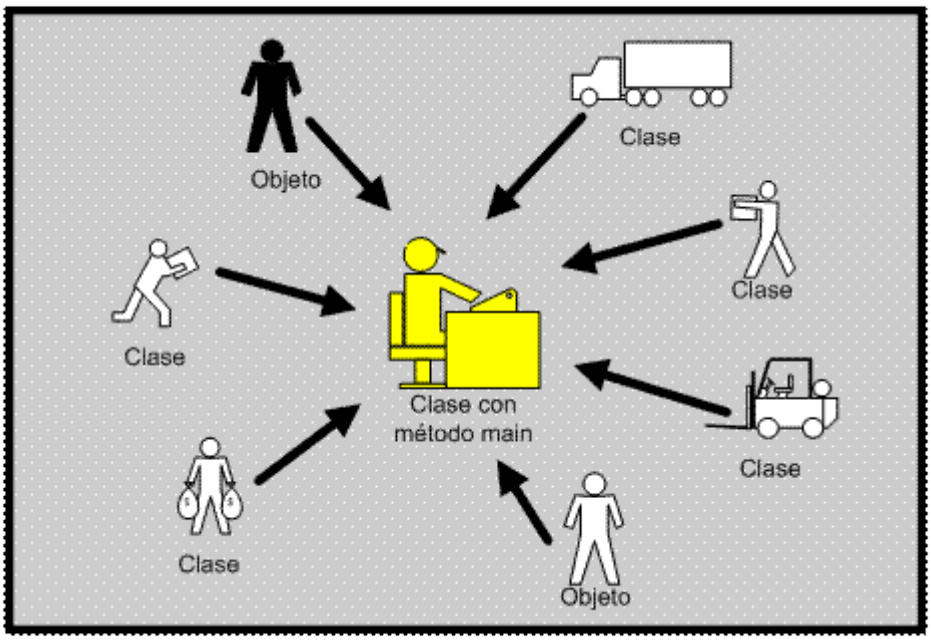
TEMA 5b: Herencia y Composición.

5.G. Relación entre clases.

5.H. Composición.

5.I. Herencia.



5.G. Relación entre clases.

1.1. Composición.

1.2. Herencia.

1.3. ¿Herencia o composición?

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:

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

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

La relación de **composición** es posible que ya la hayas tenido en cuenta si has definido clases que contenían (tenían como atributos) otros objetos en su interior, lo cual es bastante habitual. Por ejemplo, si 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.

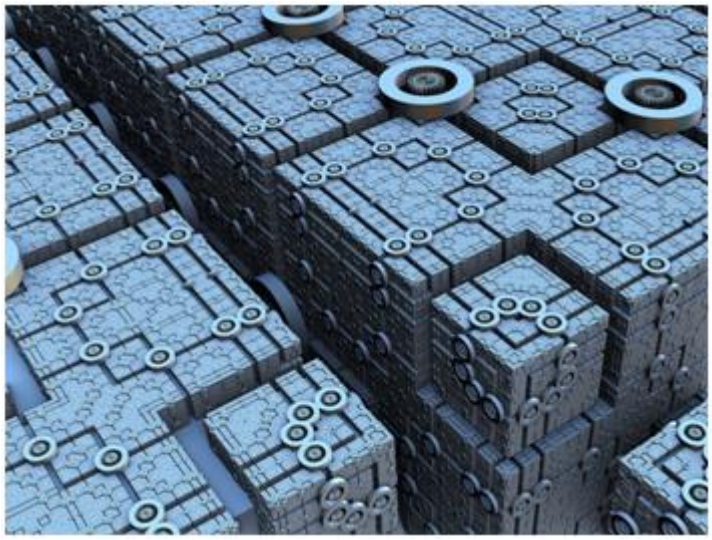
En el caso de la relación de **herencia** también la has visto ya, pues seguro que has utilizado unas clases que derivaban de otras, sobre todo, en el caso de los objetos que forman parte de las **interfaces gráficas**. Lo más probable es que hayas tenido que declarar clases que derivaban de algún componente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

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

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

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 **Rectangulo** podría contener en su interior dos objetos de la clase **Punto** para almacenar los vértices inferior izquierdo y superior derecho.
- Una clase **Empleado** podría contener en su interior un objeto de la clase **DNI** para almacenar su DNI/NIF, y otro objeto de la clase **CuentaBancaria** para guardar la cuenta en la que se realizan los ingresos en nómina.
- Una clase **JFrame** (**javax.Swing.JFrame**) de la **interfaz gráfica** contiene en su interior referencias a objetos de las clases **JRootPane**, **JMenuBar** o **JLayeredPane**, pues contiene Solmenús, paneles, etc.

Ejercicio resuelto

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

Solución:

No. Aunque claramente existe algún tipo de relación entre ambas, no parece que sea la de composición. No parece que se cumpla la expresión "**tiene un**": "Un loro tiene un ave". Se cumpliría más bien una expresión del tipo "**es un**": "Un loro es un ave". Algunos objetos que cumplirían la relación de composición podrían ser **Pico** o **Alas**, pues "un loro tiene un pico y dos alas", del mismo modo que "un ave tiene pico y dos alas". Este tipo de relación parece más de **herencia** (un loro es un tipo de ave).

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**).

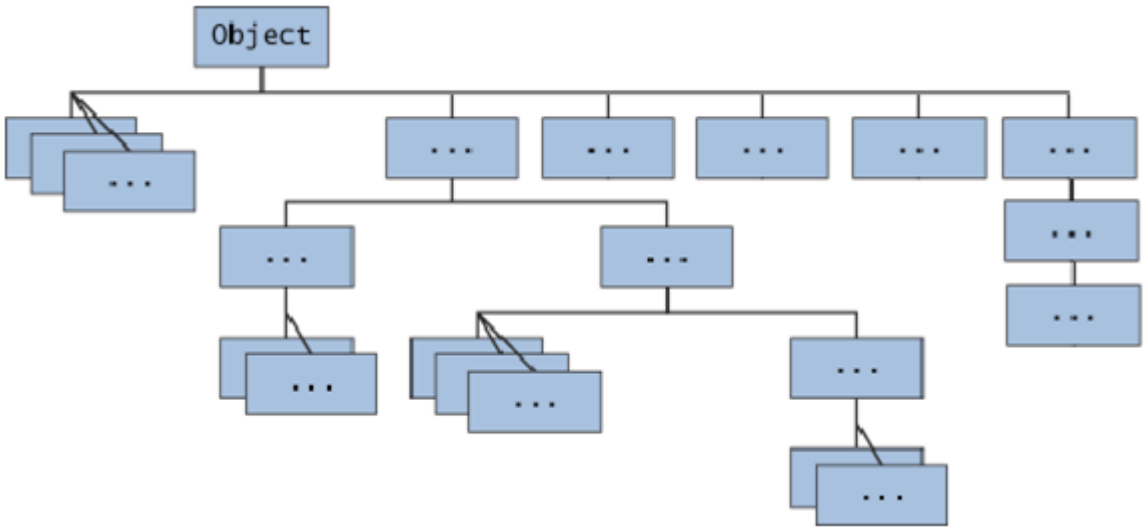
Recuperando algunos ejemplos de clases que ya has utilizado en otras unidades:

· 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 aquéllos 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?

Solución:

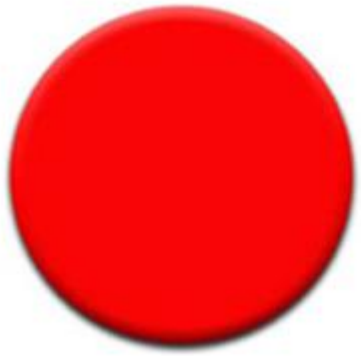
No es cierto. Aunque no indiques explícitamente ningún tipo de herencia, el compilador asumirá entonces de manera implícita que tu clase hereda de la clase **Object**, que define e implementa el comportamiento común a todas las clases.

1.3. ¿Herencia o composición?

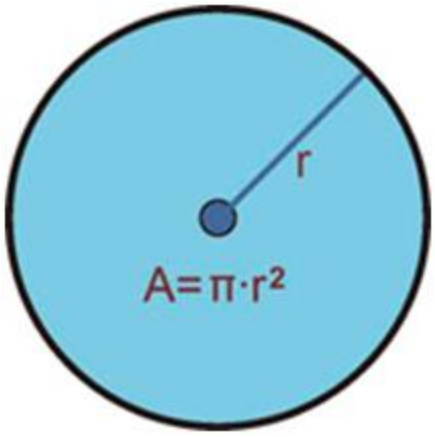
Cuando escribas tus propias clases, debes intentar tener claro en qué casos utilizar la **composición** y cuándo la **herencia**:

- **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** (**particularización**, **extensión** o **restricción**) de la clase base. Desde otro punto de vista se diría que la clase base es una **generalización** de las clases derivadas.



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



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

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

1. "**Un círculo es un punto** (su centro)", y por tanto heredará las coordenadas **x1** e **y1** que tiene todo punto. Además, tendrá otras características específicas como el **radio** o métodos como el cálculo de la **longitud** de su perímetro o 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?**".

5.H. Composición.

1. Composición.

¿Cómo 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?

1.1. Sintaxis de la composición.

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

```
class <nombreClase> {
     [modificadores] <NombreClase1> nombreAtributo1;
     [modificadores] <NombreClase2> nombreAtributo2;
…
}
```

En unidades anteriores has trabajado con la clase **Punto**, que definía las coordenadas de un punto en el plano, y con la clase **Rectangulo,** que definía una figura de tipo rectángulo también en el plano a partir de dos de sus **vértices** (**inferior izquierdo** y **superior derecho**). Tal y como hemos

formalizado ahora los tipos de relaciones entre clases, parece bastante claro que aquí tendrías un caso de **composición**: "**un rectángulo contiene puntos**". Por tanto, podrías ahora redefinir los atributos de la clase **Rectangulo** (cuatro **números reales**) como dos objetos de tipo **Punto**:

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

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

Ejercicio resuelto

Intenta rescribir los siguientes los métodos 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. **Método calcularSuperfice, 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.**

Solución:

En ambos casos la **interfaz** no se ve modificada en absoluto (desde fuera su funcionamiento es el mismo), pero internamente deberás tener en cuenta que ya no existen los atributos x1, y1, x2, y2, de tipo double, sino los atributos vertice1 y vertice2 de tipo Punto.

```
public double calcularSuperficie () {
    double area, base, altura; // Variables locales
    base= vertice2.obtenerX () - vertice1.obtenerX ();// Antes era
x2 - x1
    altura= vertice2.obtenerY () - vertice1.obtenerY ();// Antes
era y2 - y1
```

```
    area= base * altura;
    return area;
}

public double CalcularPerimetro () {
    double perimetro, base, altura;   // Variables locales
    base= vertice2.obtenerX () - vertice1.obtenerX ();  // Antes
era x2 - x1
    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes
era y2 - y1
    perimetro= 2*base + 2*altura;
    return perimetro;
}
```

1.2. 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 **Rectangulo** y observar sus nuevos métodos de tipo **get**.

Ejercicio resuelto

Dada la clase `Rectangulo`, 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`):

Solución:

Los métodos de obtención de vértices devolverán objetos de la clase **Punto**:

```
public Punto obtenerVertice1 ()
{
     return vertice1;
}

public Punto obtenerVertice2 ()
{
     return vertice2;
}
```

Esto funcionaría perfectamente, pero deberías tener cuidado con este tipo de métodos que devuelven directamente una referencia a un objeto atributo que probablemente has definido como privado. Estás de alguna manera haciendo público un atributo que fue declarado como privado.

Para evitar que esto suceda bastaría con crear un nuevo objeto que fuera una copia del atributo que se desea devolver (en este caso un objeto de la clase **Punto**).

Aquí tienes algunas posibilidades:

```
 public Punto obtenerVertice1 ()    // Creación de un nuevo punto extrayendo sus atributos
{
     double x, y;
     Punto p;
     x= vertice1.obtenerX();
     y= vertice1.obtenerY();
     p= new Punto (x,y);
     return p;
}

public Punto obtenerVertice1 ()  // Utilizando el constructor copia de Punto (si es que está definido)
{
     Punto p;
```

```
        p= new Punto (this.vertice1); // Uso del constructor copia
        return p;
}
```

De esta manera, se devuelve un punto totalmente nuevo que podrá ser manipulado sin ningún temor por parte del código cliente de la clase pues es una copia para él.

Para el método **obtenerVertice2** sería exactamente igual.

1.3. 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 **Rectangulo** que contiene en su interior dos objetos de la clase **Punto**. En los constructores del rectángulo habrá que incluir todo lo necesario para crear dos instancias de la clase **Punto** evitando las referencias a parámetros (haciendo copias).

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 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.**

Solución:

Esta es una 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**:

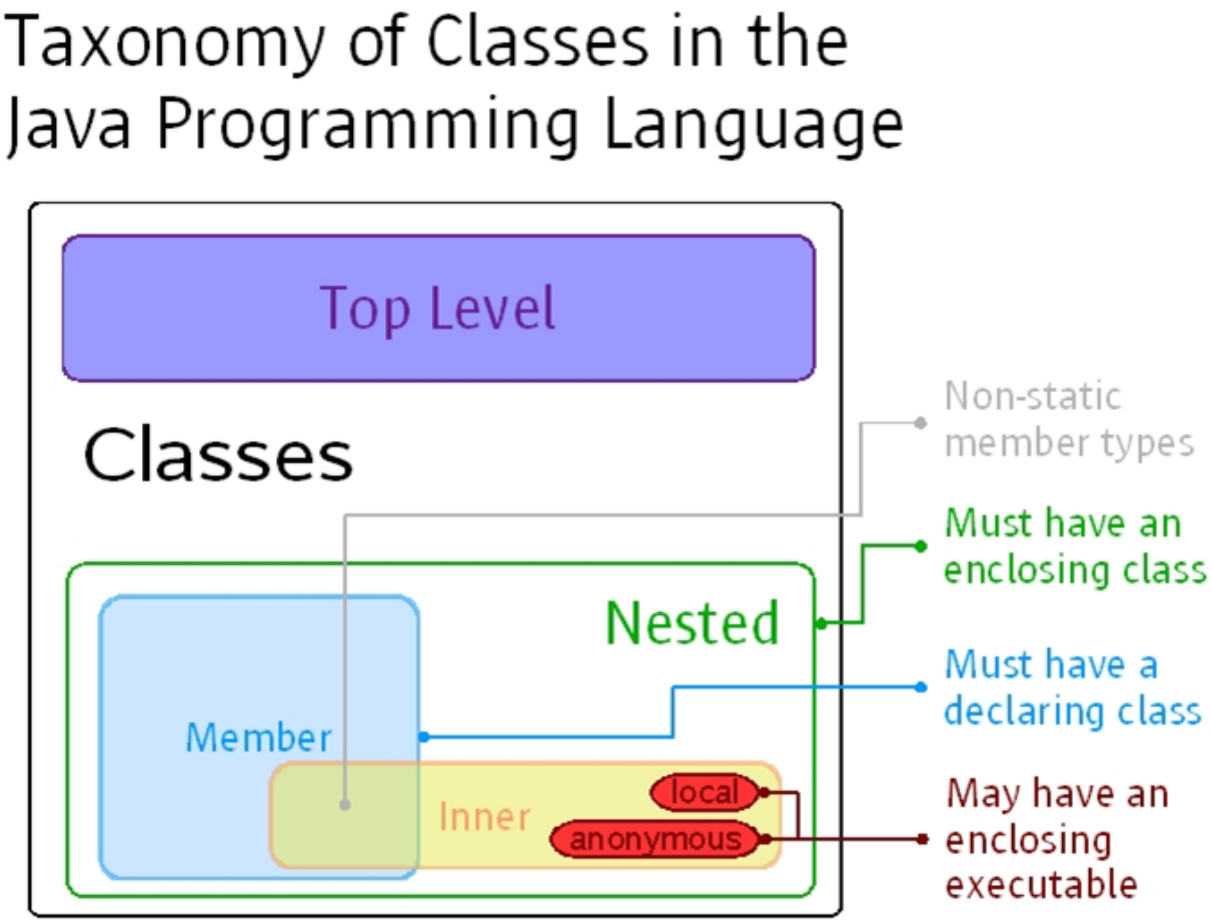
```
// Constructor copia
public Rectangulo (Rectangulo r) {
    this.vertice1= new Punto (r.obtenerVertice1() );
    this.vertice2= new Punto (r.obtenerVertice2() );
}
```

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.

1.4. 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
…
        }
}
```



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:

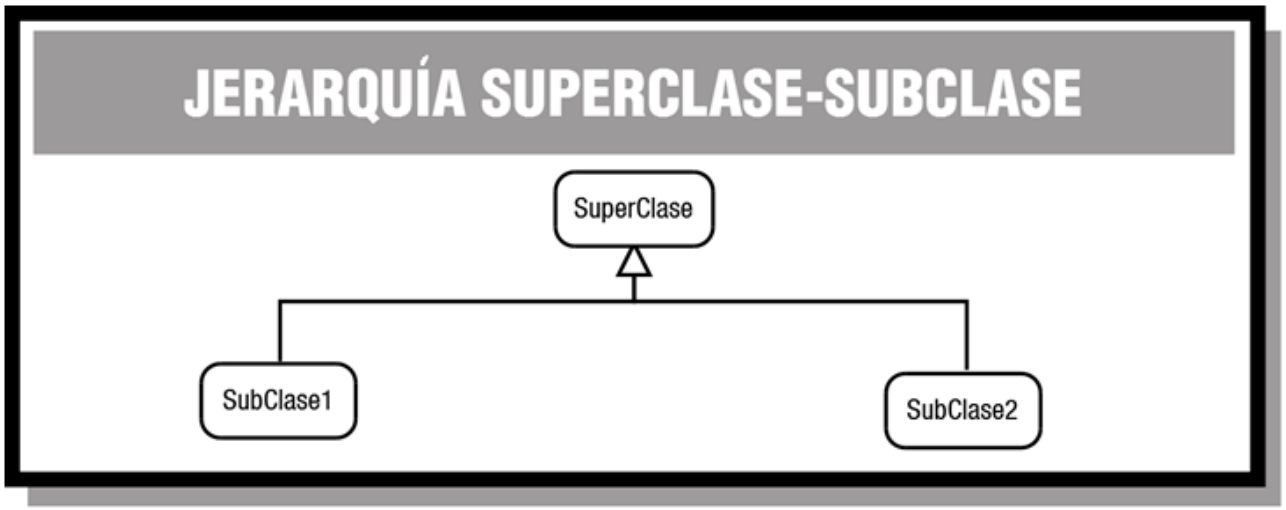
- **Agrupar** clases que sólo tiene sentido que existan en el entorno de la clase en la que han sido definidas, de manera que se oculta su existencia al resto del código.
- Incrementar el nivel de **encapsulación** y **ocultamiento**.
- Proporcionar un **código fuente más legible y fácil de mantener**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

5.I. Herencia.

1. **Herencia.**
    - **1.1. Sintaxis de la herencia.**
    - **1.2. Acceso a miembros heredados.**
    - **1.3. Utilización de miembros heredados (I). Atributos.**
    - **1.4. Utilización de miembros heredados (II). Métodos.**
    - **1.5. Redefinición de métodos heredados.**
    - **1.6. Ampliación de métodos heredados.**
    - **1.7. Constructores y herencia.**
    - **1.8. Creación y utilización de clases derivadas.**
    - **1.9. La clase Object en Java.**
    - **1.10. Herencia múltiple.**
    - **1.11. Clases y métodos finales.**

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**).

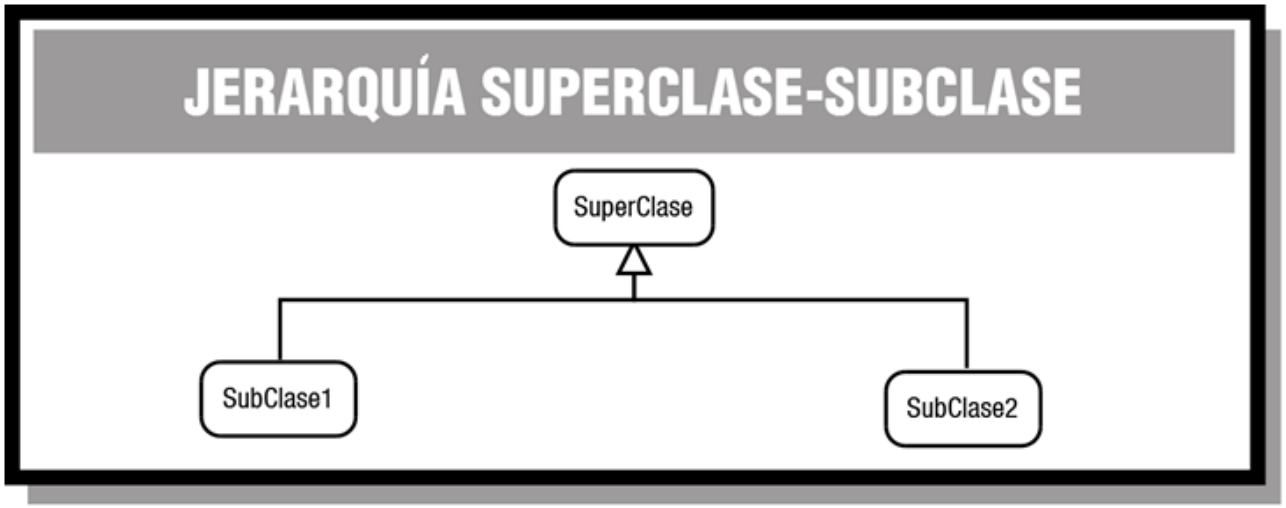
Una clase derivada extiende la funcionalidad de la clase base sin tener que volver a escribir el código de la clase base.

Autoevaluación

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

1. Herencia.

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

Respuesta: No es del todo cierto. Se heredan todos los miembros, pero la clase derivada no tendrá un acceso directo a aquellos miembros de la clase base que sean privados.

1.1. Sintaxis de la herencia.

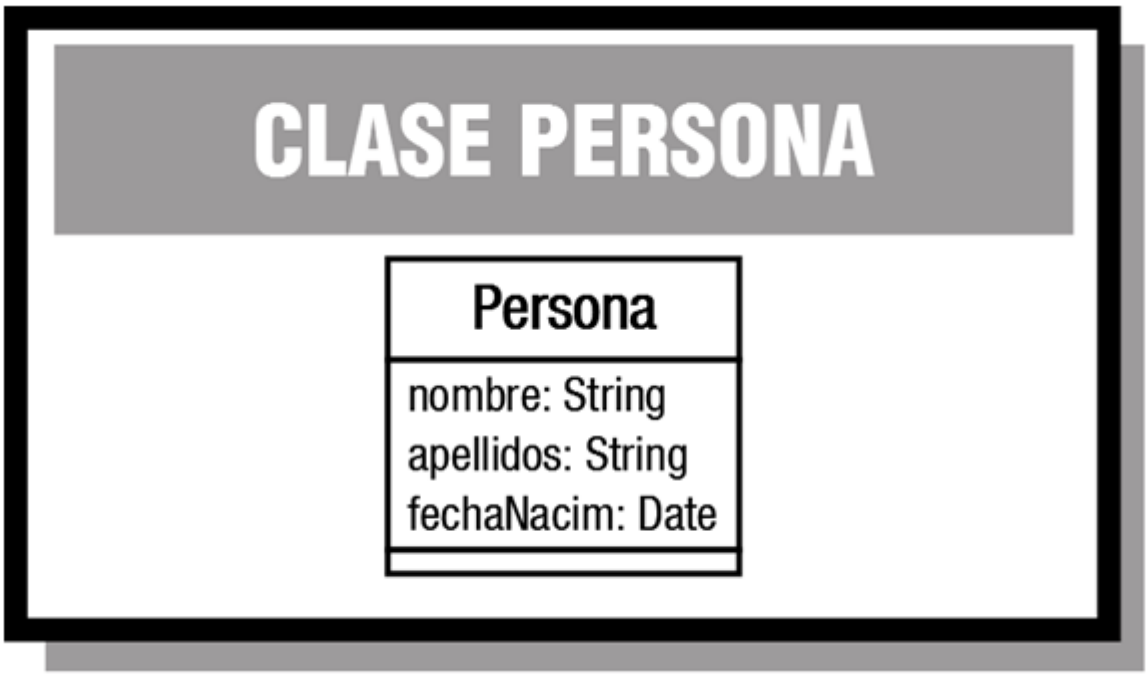
En Java la **herencia** se indica mediante la palabra reservada **extends**:

```
[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**:

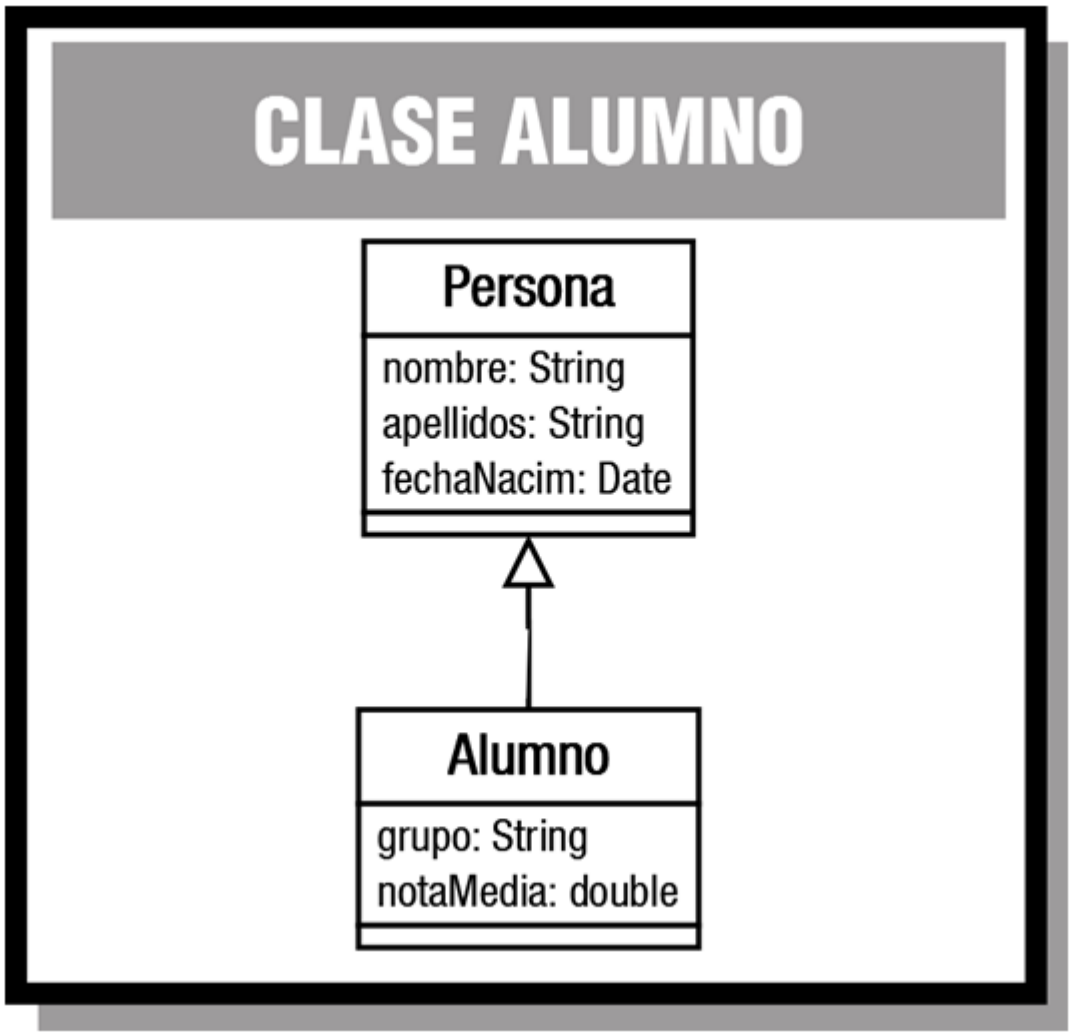
```
public class Persona {
    String nombre;
    String apellidos;
    GregorianCalendar fechaNacim;
    …
}
```



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).

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?

Solución:

Está claro que un **Profesor** es otra especialización de **Persona**, al igual que lo era **Alumno**, así que podrías crear otra clase derivada de **Persona** y así aprovechar los atributos genéricos (**nombre**, **apellidos**, **fecha de nacimiento**) que posee todo objeto de tipo **Persona**. Tan solo faltaría añadirle sus atributos específicos (**salario** y **especialidad**):

```
public class Profesor extends Persona {

	String especialidad;

	double salario;

	…

}
```

1.2. 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:

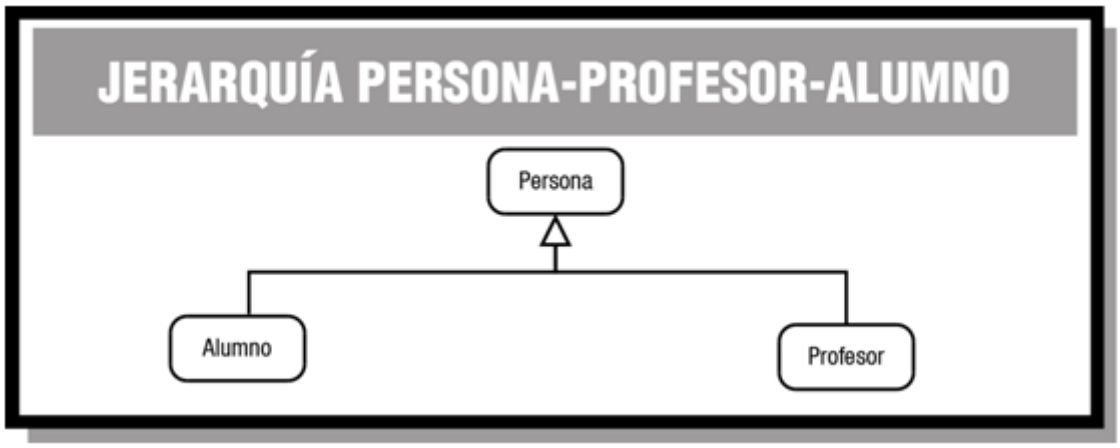
Cuadro de niveles accesibilidad a los atributos de una clase				
	Misma clase	**Subclase**	**Mismo paquete**	**Otro paquete**
Sin modificador (paquete)	X		X	X
public	X	X	X	X
Private	X			
Protected	X	X	X	

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

Solución:

1. **Clase Alumno.**

Se trata simplemente de añadir el modificador de acceso **protected** a los nuevos atributos que añade la clase.

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

2. **Clase Profesor.**

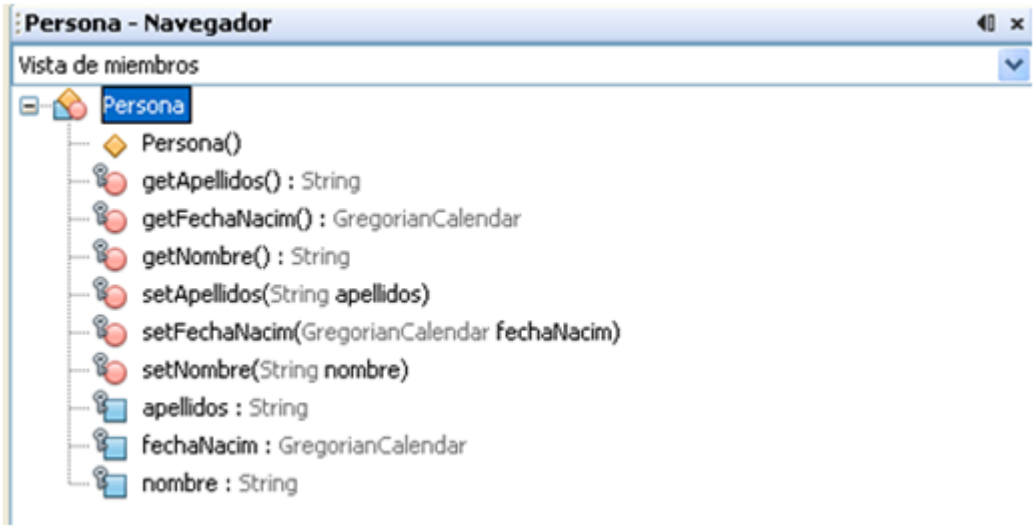
Exactamente igual que en la clase **Alumno**.

```
public class Profesor extends Persona {
    protected String especialidad;
    protected double salario;
    …
}
```

1.3. 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).

Solución:

Una posible solución sería:

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 Persona {
        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
```

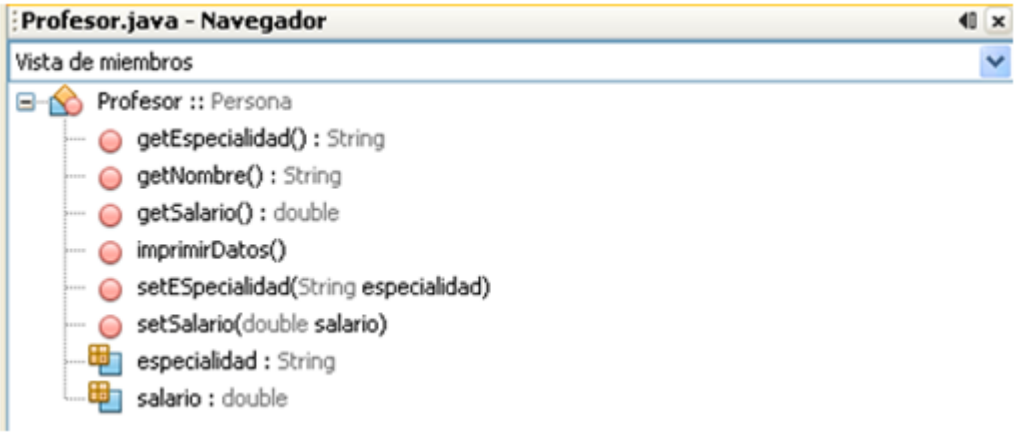
```
        public void setFechaNacim (GregorianCalendar fechaNacim){
            this.fechaNacim= fechaNacim;
        }
}
```

1.4. 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`**).
- Otros cuatro más por ser **`Alumno (getGrupo, setGrupo, getNotaMedia, setNotaMedia`**).

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



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.

Solución:

Una 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 Persona {
        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;
        }
}
```

1.5. 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 rescribir 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 ()
```

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.

Solución:

1. **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. Esos son los métodos que se implementaron en el ejercicio anterior (`getGrupo`, `setGrupo`, etc.).

Ahora bien, hay que escribir otro método más, pues tienes que redefinir el método `getNombre` para que tenga un comportamiento un poco diferente al `getNombre` que se hereda de la clase base **Persona**:

```
// Método getNombre
@Override
public String getNombre (){
            return "Alumno: " + this.nombre;
}
```

En este caso podría decirse que se "renuncia" al método heredado para redefinirlo con un comportamiento más especializado y acorde con la clase derivada.

2. **Clase `Profesor`.**

Seguimos exactamente el mismo procedimiento que con la clase `Alumno` (redefinición del método `getNombre`).

```
// Método getNombre
@Override
public String getNombre (){
```

```
            return "Profesor: " + this.nombre;
}
```

1.6. 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.

Solución:

1. Método **mostrar** de la clase **`Persona`.**

```
public void mostrar () {
    SimpleDateFormat formatoFecha = new
SimpleDateFormat("dd/MM/yyyy");
    String Stringfecha=
formatoFecha.format(this.fechaNacim.getTime());

    System.out.printf ("Nombre: %s\n", this.nombre);
    System.out.printf ("Apellidos: %s\n", this.apellidos);
    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);
}
```

2. Método **mostrar** de la clase **`Profesor`.**

Llamamos al método **`mostrar`** de su **clase padre** (**`Persona`**) y luego añadimos la **funcionalidad específica** para la **subclase** **`Profesor`**:

```
public void mostrar () {
                    super.mostrar ();      // Llamada al método
"mostrar" de la superclase
```

```
// A continuación mostramos la información "especializada" de esta
subclase

    System.out.printf ("Especialidad: %s\n", this.especialidad);

    System.out.printf ("Salario: %7.2f euros\n",
this.salario);

}
```

3. Método **mostrar** de la clase **Alumno.**

Llamamos al método **mostrar** de su **clase padre** (**Persona**) y luego añadimos la **funcionalidad específica** para la **subclase Alumno**:

```
public void mostrar () {

                super.mostrar ();


// 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);

}
```

1.7. 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 (p**rimero 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.nombe= 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 grupo, double notaMedia) {
    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.

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.

Solución:

```
public Profesor (String nombre, String apellidos, GregorianCalendar
fechaNacim, String especialidad, double salario) {
    super (nombre, apellidos, fechaNacim);
    this.especialidad= especialidad;
    this.salario= salario;
}
```

1.8. 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).

1.9. La clase Object en Java.

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

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.

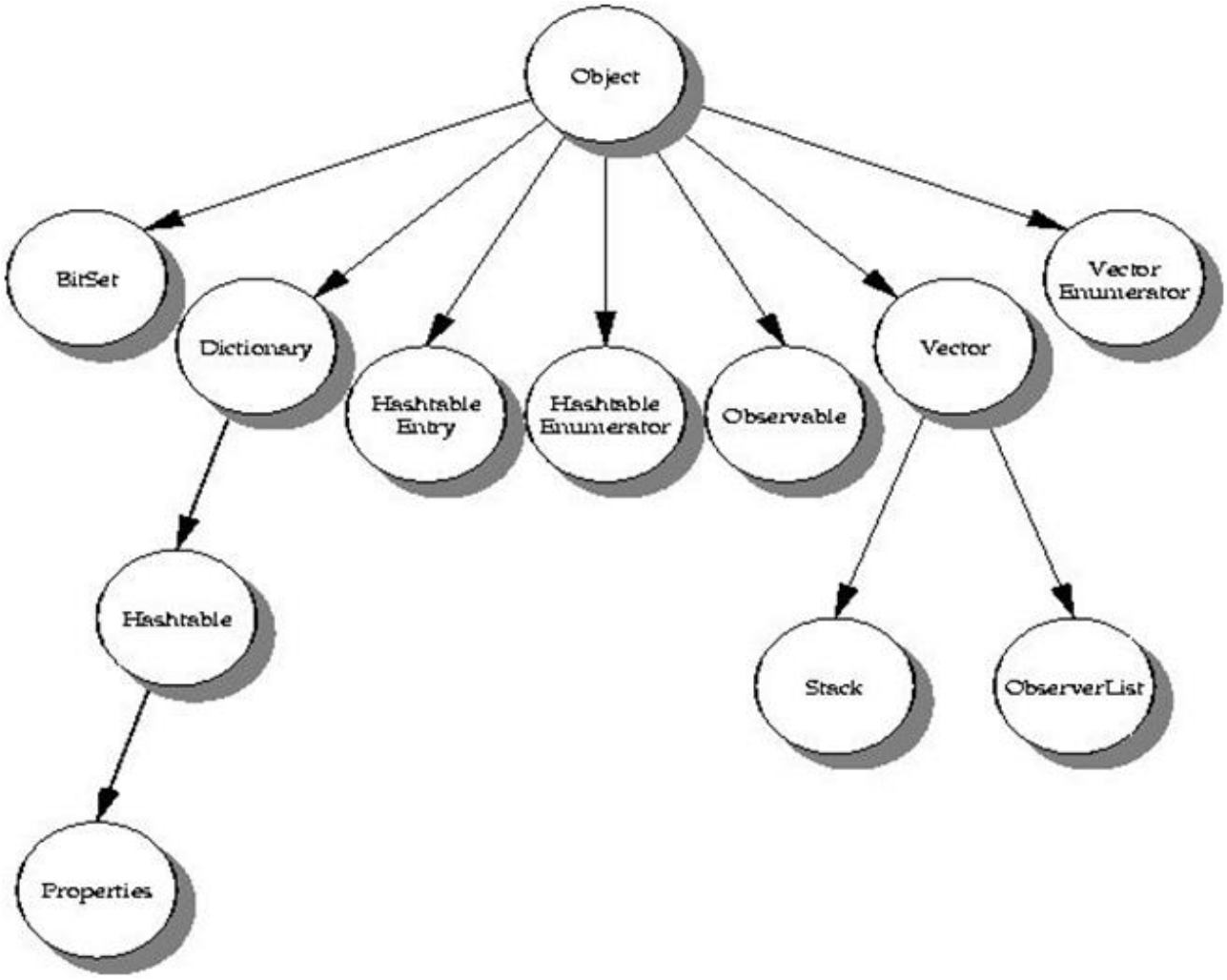


Imagen extraída de curso Programación del MECD.

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**.

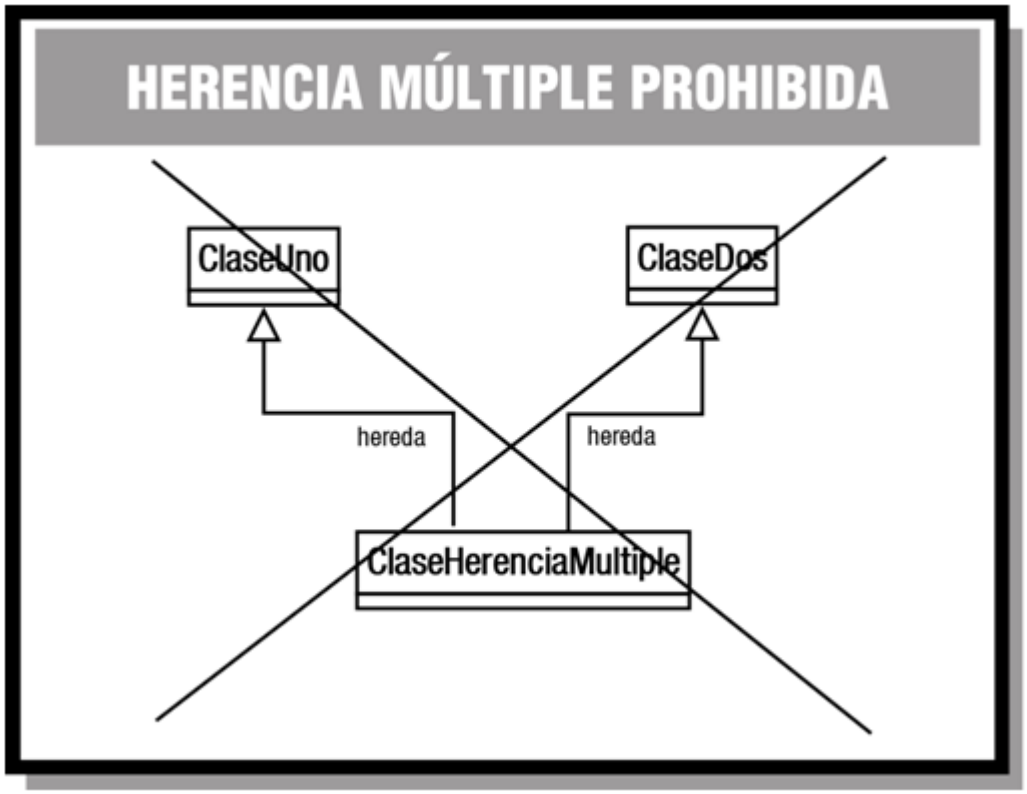
1.10. Herencia múltiple.

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.

1.11. 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**.