Caso

*La programación orientada a objetos ayuda a mejorar la calidad del software y una empresa como **SI Andalucía**, que quiere competir con las grandes empresas del sector en su zona e incluso a nivel nacional, tiene que hacer software de calidad y en un tiempo competitivo. Cuando preparan a sus programadores, uno de los puntos más importantes es que sigan un método de trabajo que se integre perfectamente en el equipo de trabajo de la empresa de modo que cualquiera pueda continuarlo con las mínimas dificultades, si ocurre algún imprevisto y algún programador tiene que dejar el proyecto por cualquier causa. La verdad es que con Víctor les está costando bastante. No está muy habituado a trabajar en equipo y menos aún a seguir un método de trabajo organizado, en el que debe ajustarse a los procedimientos preestablecidos.*

*La esperanza de **Víctor** es su compañera **Carmen**, que tiene una paciencia infinita y le conoce muy bien. Siempre que **Carmen** le ha explicado algo **Víctor** ha terminado asimilándolo en tiempo record, por lo que ella se ha convertido en su guía en la empresa.*

*En la aplicación que están desarrollando actualmente, es necesario hacer la definición de una clase **Trabajador** para la gestión empresarial de unas oficinas. **Carmen** le explica las características comunes que tienen todos los trabajadores de la empresa y qué características particulares tienen los trabajadores Fijos frente a los Temporales. Le demuestra entonces la jerarquía de clases que se va a utilizar con las clases **TrabajadorFijo** y **TrabajadorTemporal** como subclases de la superclase **Trabajador**.*

- ***Común a todos los trabajadores**: todos tienen un NIF, un nombre, una fecha de nacimiento, una categoría profesional, una fecha de alta en la empresa, una edad expresada en años, una dirección de correo electrónico y un sueldo.*
- ***Sólo los fijos**: Tienen antigüedad en la empresa expresada en años. Además, sólo de este tipo de trabajadores me interesa recoger información acerca del número de hijos que tiene, pues esto influirá en su sueldo como complemento. El sueldo de un trabajador fijo se calcula como la suma del salario base, el complemento asociado a la categoría profesional que se tenga, 50 euros por hijo y 20 euros por año de antigüedad en la empresa.*
- ***Sólo los temporales**: Tienen una fecha de finalización de contrato. Por su parte, el sueldo de un trabajador temporal se calcula como la suma entre el salario base y el complemento asociado a la categoría profesional que se tenga.*

***Carmen** concluye diciéndole que esto es un claro ejemplo de Reutilización y Extensibilidad del código, y que esta reutilización no se queda en este proyecto, sino que se puede extender a cualquier proyecto que utilice la clase **Trabajador** como superclase de otras clases.*

Reutilización y extensibilidad: pilares básicos de la calidad del software (I)

En la unidad anterior iniciamos nuestro camino hacia la consecución de software de calidad utilizando como vehículo la programación orientada a objetos y, más concretamente, el lenguaje de programación Java. De entre todas las características que debe reunir un software para que sea considerado de calidad, se señalaron especialmente dos:

- **La extensibilidad** o capacidad de los sistemas software de ser adaptados fácilmente a los cambios de las especificaciones, produciendo programas a los que se les pueda añadir nueva funcionalidad o se les pueda realizar modificaciones de manera sencilla y poco traumática, lo cual reduce considerablemente los costes derivados del

mantenimiento.

- **La reutilización** o capacidad de los elementos del software de servir para la construcción de muchas aplicaciones diferentes, evitando así la reescritura del mismo código una y otra vez.

Una vez marcado el objetivo, vimos cómo **la programación orientada a objetos nos proporciona un conjunto de técnicas de elaboración de programas informáticos** para alcanzarlo; es decir, **para obtener software que sea extensible y reutilizable**. Así, vimos que:

- El principal problema de la programación estructurada o basada en procedimientos es que las unidades de programación no reflejan de manera fácil y efectiva a las entidades del mundo real, lo cual deriva en que estas unidades no sean particularmente reutilizables. Sin embargo, en la programación orientada a objetos, **una clase modela una entidad del mundo real, por lo que si ésta está diseñada apropiadamente, es muy probable que pueda ser reutilizada sin ninguna modificación en proyectos futuros**. De hecho, toda la potencia de la programación orientada a objetos proviene de la correspondencia directa entre la realidad modelada y el programa que la modela.
- Una clase es un módulo tanto a nivel semántico como a nivel sintáctico.
  - A nivel **semántico** porque se encierra en una misma entidad conceptual o semántica, la clase, todos los datos que describen a una entidad genérica del mundo real, junto con todas las operaciones que se pueden realizar sobre esos datos.
  - A nivel **sintáctico** porque una clase está encerrada en un único archivo o módulo a nivel de programa.
  - Por lo tanto, **la programación orientada a objetos sigue una metodología modular en el desarrollo de programas informáticos, lo cual favorece la extensibilidad** pues hace que la introducción de cambios sea más fácil al estar éstos localizados en módulos concretos.
  - Así mismo, también favorece **la reutilización** de ciertas partes del software, las que me proporcionen ciertos módulos del programa.
- Para poder utilizar una clase es suficiente con conocer su interfaz, es decir, qué servicios o funcionalidad ofrece la clase y cómo se utilizan dichos servicios, siendo totalmente innecesario conocer su implementación. **Esta propiedad de la programación orientada a objetos, conocida con el nombre de ocultación de la implementación, favorece la extensibilidad, pues permite variar la implementación de las clases sin afectar a sus clientes**, que seguirán interactuando con ellas de la misma manera a como lo hacían antes de las modificaciones al no haberse modificado la interfaz con la que trabajan.

Reutilización y extensibilidad: pilares básicos de la calidad del software (II)

Sin embargo, **las técnicas estudiadas hasta aquí no son suficientes**. Las clases proporcionan una buena técnica de descomposición modular y poseen muchas de las cualidades esperadas de los

componentes reutilizables: son módulos homogéneos y coherentes, y se puede separar claramente sus interfaces de sus implementaciones de acuerdo con el principio de ocultación de la información; pero **se necesita más para alcanzar en su totalidad los objetivos de reutilización y extensibilidad**:

- Para favorecer la reutilización **se necesitan técnicas para capturar las evidentes semejanzas que existen dentro de grupos de estructuras similares, para no tener que estar repitiendo lo mismo una y otra vez, pero respetando al mismo tiempo las muchas diferencias que caracterizan a cada una de ellas**. Imagina una granja con distintos tipos de animales. Es evidente que cada uno de ellos tiene atributos y comportamiento propios que lo diferencia del resto y que, por lo tanto, debe ser modelado en una clase distinta. Piensa, por ejemplo, en gallinas y en vacas. Poco tienen que ve las unas con las otras ¿verdad? Pero, ¿realmente no tienen nada en común? Si lo pensamos detenidamente, seguro que encontramos elementos comunes entre ellas, ya sean atributos o comportamiento, por el simple hecho de que ambos son animales de nuestra granja. ¿No sería estupendo no tener que repetir el código de estas características comunes en cada una de las clases que modela a cada tipo de animal? ¿No sería fantástico poder escribir una sola vez esta estructura común a todos los animales de mi granja y poder reutilizarla en cada una de las clases particulares que modela a cada tipo concreto de animal? Pues bien, **esto es posible en la programación orientada a objetos gracias a lo que se conoce como herencia**, que será algo que veremos en profundidad en este tema.
- Por su parte, para favorecer la extensibilidad **se necesitan técnicas que me permitan programar para esta generalidad común de la que acabamos de hablar, de tal manera que el mismo código funcione exactamente igual con cada uno de los distintos elementos particulares que comparten dicha generalidad**. Imagina que necesito realizar alguna acción sobre todos los animales de la granja, sean estos de la clase que sean, por ejemplo, alimentarlos. Es evidente que para alimentar a una vaca no se va a hacer exactamente lo mismo que para alimentar a una gallina. ¿No sería estupendo que el mismo código genérico del tipo "dar de comer a los animales" me valiese exactamente lo mismo para alimentar vacas que para alimentar gallinas? ¿No sería fantástico que yo pudiese ordenar de manera genérica que se alimentase a todos los animales de la granja y que cada animal concreto estableciese exactamente cómo debe ser alimentado? Pues bien, **esto es posible en la programación orientada a objetos gracias a lo que se conoce como polimorfismo y ligadura dinámica**, que será algo que también veremos en profundidad en este tema.

Todos los animales tienen cosas en común

En este tema veremos estas potentes herramientas que nos proporciona la programación orientada a objetos para alcanzar el objetivo de crear programas altamente reutilizables y extensibles: la herencia y el polimorfismo. Comencemos el camino.

Autoevaluación

PARA SABER MÁS.
Aquí encontrarás unas interesantes reflexiones, enfocadas a la programación orientada a objetos, sobre la reutilización del software, basándose en experiencias en la Industria, en las que se deduce que mejora la calidad con aspectos organizativos más que en los propiamente tecnológicos.
Reusabilidad y Desarrollo Orientado a Objetos **[Versión en caché]**

La herencia

__Víctor__ ha terminado entendiendo el concepto de herencia desde el principio, y tal y como se lo ha explicado __Carmen.__ No ve que tenga ninguna dificultad. Pero otra cosa es ponerlo en práctica en un proyecto y sin que nadie le

*prepare el terreno como viene haciendo su compañera. Intenta por su cuenta definir clases partiendo de una superclase como la clase **Trabajador**. **Carmen** le ha proporcionado las particularidades que tiene cada tipo de trabajador (el que tiene contrato fijo y el que es temporal) y le ha pedido que partiendo de la clase **Trabajador**, intente crear las clases TrabajadorFijo y TrabajadorTemporal, para ello debe elaborar el código que defina cada una de esas clases y los métodos que utilizan.*

"Tiene los mismos ojos del padre", "mira qué guapa que es, tiene la misma carita de la madre", "no puede negar que es un Fernández, todos tienen el mismo carácter". Seguro que frases como éstas las has escuchado una y mil veces. ¿Quién es la responsable de que los hijos tengamos los mismos rasgos e incluso el mismo comportamiento que sus progenitores? Sin duda alguna, la herencia genética.

En cierta medida, podríamos afirmar que, gracias a la **herencia** genética, los hijos reutilizamos el código genético de nuestros padres, lo cual nos evita tener que comenzar desde cero la construcción de nuestro ser. Esto no quiere decir, sin embargo, que los hijos seamos copias idénticas de nuestros padres, no somos clones, sino que tenemos características y personalidad propia que nos hace únicos y diferentes.

¿Es posible trasladar este mecanismo de reutilización biológica al mundo de la programación? Es decir, ¿es posible crear clases que sean hijas de otras clases y que, por lo tanto, puedan reutilizar el código, aunque a su vez puedan incorporar elementos nuevos que las haga únicas? **La respuesta es, evidentemente, que sí, y el mecanismo que nos lo permite**, como no podía ser de otra manera, **recibe el nombre de herencia**. Veamos cómo funciona.

Relaciones básicas entre clases: composición y herencia

Como ya sabemos, una **clase** modela una entidad del mundo real y encierra en su interior todo lo que tenga que ver con ella; es decir, todos los datos que la describen y todas las operaciones que se pueden realizar sobre esos datos. De esta manera, programando módulos con **alta cohesión y bajo acoplamiento**, pretendemos conseguir que una clase sea una entidad de programación lo más independiente posible y, consecuentemente, fácil de reutilizar en la resolución de otros problemas distintos a aquellos para los que fue programada. Sin embargo, una clase no es un ente aislado y sin relaciones con el exterior; más bien todo lo contrario, una clase es una entidad que tiene que colaborar con otras para dar solución a los problemas y que, por lo tanto, establece relaciones con otras clases.

Progresar en la reutilización o en la extensibilidad exige que se aprovechen las fuertes relaciones conceptuales que guardan las clases entre sí: una clase puede ser una extensión, una

especialización o una combinación de otras clases, y se necesita soporte por parte del lenguaje para registrar y utilizar estas relaciones. Básicamente, **las relaciones entre clases se reducen a dos tipos**:

- **La composición**, que no es más que un caso particular del concepto de clientela que vimos en la unidad anterior.
- **La herencia**, que es quizá una de las características principales de la programación orientada a objetos.

Aprendamos en qué consiste cada una de estas relaciones entre clases.

Lee detenidamente las siguientes frases que pueden surgir durante el proceso de análisis y diseño de una aplicación informática:

- "un coche tiene una radio",
- "un departamento tiene un jefe de departamento",
- "una cuenta bancaria tiene un titular de la cuenta",
- "un elemento sobre el cual se produce un evento determinado tiene una lista de otros elementos que desean que se les comunique que dicho evento se ha producido".

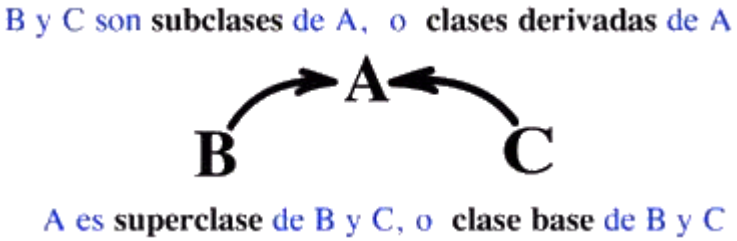
¿Qué tienen en común? Todas hablan de una circunstancia en la que una entidad tiene a otra como una de sus partes. **El caso más frecuente de relación entre clases es el que viene dado por esta situación "tiene un"**. De igual manera que un ordenador es el producto del ensamblaje de distintas piezas hardware, una clase puede estar formada por la composición de varias clases distintas, recibiendo este tipo de relación entre clases el nombre de composición. **La composición se da cuando una clase tiene alguna variable de instancia que es una referencia a un objeto de otra clase**. Además, en cierto modo, éste es un mecanismo de reutilización, pues para la construcción de una clase utilizamos como cliente a otra clase ya existente, la cual reutilizamos.

Lee ahora estas otras frases:

- "Un coche es un vehículo",
- "una cuenta corriente es una cuenta bancaria",
- "una gallina es un animal de granja",
- "un préstamo hipotecario es un préstamo".

En esta ocasión, todas ellas hablan de una situación en la que una entidad **es** un tipo especial de otra entidad. **Esta relación "es un" o "es un tipo de" es también bastante frecuente entre clases, donde una de ellas no es más que una especialización o una extensión de otra**. Es decir, una entidad es igual a una ya existente, con sus mismas características, pero además tiene otras particularidades que la diferencian de aquélla y la hacen tener entidad propia. ¡Oye! ¿No se parece esto a lo que sucede entre hijos y padres con la herencia biológica? La verdad es que alguna semejanza tiene. De hecho, **este tipo de relación entre clases representa lo que en el mundo de la programación orientada a objetos recibe el nombre de herencia**.

Se dice que una clase B hereda de una clase A cuando incorpora la estructura (atributos) y el comportamiento (métodos) de la clase A, pero puede incluir algunas adaptaciones, como añadir nuevos atributos o nuevos métodos, o cambiar la implementación de los existentes. Se dice entonces que la clase B es una versión especializada o extendida de la clase A. **A la clase de la que se hereda se le llama superclase, mientras que a la clase que hereda se le llama subclase**.



Autoevaluación

Características básicas de la herencia (I)

Como estamos viendo, **la herencia es un mecanismo potente de reutilización**, pues una clase absorbe las características de otra ya existente, la cual reutiliza. Pero que una clase herede de otra tiene más implicaciones de lo que en principio podría parecer y a lo largo de esta unidad veremos la gran potencia de reutilización que el mecanismo de la herencia nos ofrece. No obstante, antes de adentrarnos más en los entresijos de la herencia y en sus beneficios, hagamos una pausa para puntualizar algunas cuestiones que creemos importantes:

- **Una superclase puede tener muchas subclases**, tantas como sean necesarias en el modelado de nuestra aplicación.
- **Cuando una clase hereda de otra, hereda todos y cada uno de los miembros de la misma**; es decir, todas sus variables de instancia, todas sus variables de clase, todos sus métodos de clase y todos sus métodos de instancia. No se puede establecer una herencia parcial, no se puede decidir que se heredan ciertos miembros de una clase y que otros miembros no se heredan.
- Una vez creada, **cada subclase puede convertirse en superclase de futuras subclases**, con lo que se puede formar una cadena de herencia. Llamamos **superclase directa** a la superclase a partir de la cual la subclase hereda en forma explícita, estableciéndolo así en el propio código de definición de la clase. Por su parte, llamamos **superclase indirecta** a una clase de la que no se hereda directamente, sino a través de otra clase que sí hereda de ella directa o indirectamente. Evidentemente, **una subclase posee todas las características de todas las clases de las que hereda, ya sea esta herencia directa o indirecta**.
- En ciertas ocasiones nos podemos ver en la necesidad de **diseñar una clase que herede directamente de varias clases a la vez.** Volvamos al ejemplo de la granja para ilustrar esta necesidad. Imagina que tenemos una superclase "**`AnimalDeGranja`**" que recoge todas las características comunes a cualquier animal de la granja. Dicha superclase tiene dos subclases: "**`Cerdo`**" y "**`Perro`**", cada una de las cuales es una especialización de un animal de granja aportando las características particulares de ese tipo de animal concreto. Ésta es una jerarquía de herencia que nos da una visión de los cerdos y perros como animales de granja. Sin embargo, si lo miramos desde un punto de vista empresarial, podríamos considerar a los cerdos como productos de nuestra empresa agropecuaria, al igual que, desde este punto de vista, también sería un producto, por ejemplo, un tomate o una lechuga.

Si tenemos una clase llamada "**`Producto`**", que define las características comunes a todos los elementos que son productos que se venden en nuestra granja, entonces la clase "**`Cerdo`**", además de heredar de la clase "**`AnimalDeGranja`**" también debería heredar de esta clase "**`Producto`**".

Se llama herencia múltiple al hecho de que una clase herede de forma directa de dos o más clases.
Se llama herencia simple al hecho de que una clase herede de forma directa únicamente de una superclase.

Características básicas de la herencia (II)

- Al igual que el ordenamiento en forma de árbol de los miembros de una familia según las relaciones de parentesco entre ellos recibe el nombre de árbol genealógico, **al ordenamiento que define y refleja las relaciones de herencia entre clases recibe el nombre de jerarquía de clases o jerarquía de herencia** y también tiene una representación gráfica. Así, por ejemplo, la jerarquía de clases resultante en nuestro programa de gestión de los trabajadores de una empresa es la siguiente:

- Sabemos que cuando una clase hereda de otra absorbe y hace propias todas sus características: atributos y métodos, estableciéndose entre ellas una relación del tipo "es un". Este tipo de relación no viene más que a decir que **un objeto de la subclase es también un objeto de la superclase y, consecuentemente, puede ser tratado como tal**. Ejemplificando esta afirmación con nuestro programa de **gestión de trabajadores**, lo que estamos diciendo es que un "trabajador fijo" es un "trabajador" y, por esa razón, se va a poder hacer con él cualquier cosa que se pudiese hacer con un "trabajador"; algo totalmente lógico si tenemos en cuenta que una subclase tiene todos los atributos y métodos de su superclase. Esta afirmación, que puede parecer trivial, tiene muchas más implicaciones de las que a primera vista pudiera parecer, y va a dar pie a la aparición de otra potente herramienta de la programación orientada a objetos: **el polimorfismo,** mecanismo que redunda ostensiblemente en la capacidad de extensibilidad de nuestras aplicaciones. No obstante, aparquemos el polimorfismo de momento porque ya habrá tiempo de volver a él más adelante una vez que hayamos afianzado el concepto de herencia.
- Por su parte, la inversa de la afirmación anterior no es cierta. Es decir, **un objeto de la superclase no puede ser tratado como un objeto de la subclase**, pues no soporta las novedades o especializaciones que aporta ésta. Volviendo al ejemplo de nuestra aplicación de gestión, no podemos tratar a un objeto de la clase Trabajador como trataríamos a un objeto de la clase `TrabajadorFijo`, pues un "trabajador" no es un "trabajador fijo".
- Un problema que se puede producir en la herencia es que una subclase herede métodos o propiedades que no necesita o que no debe tener, lo cual tiene difícil solución.
- Otro problema que puede aparecer con la herencia es que una subclase requiera que un método heredado realice su tarea de una manera específica o distinta a como lo hace en la superclase. En estos casos **la subclase puede sobrescribir el método de la superclase con una implementación propia y apropiada.** Esta acción recibe en el ámbito de la programación orientada a objetos el nombre técnico de **redefinición de un método**. Esta posibilidad de redefinir un método en una subclase, unido al mecanismo del polimorfismo, abren al programador multitud de posibilidades que ya abordaremos en profundidad más adelante en esta unidad.

Si para un programador inexperto puede ser difícil incluso **identificar las clases** a utilizar en su programa, tanto más difícil le suele resultar identificar las **jerarquías de herencia** que subyacen en el problema al que intenta dar solución. Sin embargo, identificar y utilizar estas jerarquías de clases es de vital importancia para obtener un software que sea altamente reutilizable y extensible. No hay recetas

mágicas para crear buenas jerarquías de clases; no obstante, para identificarlas los programadores expertos suelen utilizar dos estrategias:

- Tratar de detectar clases que, aunque sean distintas, compartan un comportamiento común. Esta estrategia recibe el nombre de **generalización o factorización**.
- Tratar de detectar clases que sean un caso especial de otras. Esta estrategia recibe el nombre de **especialización o abstracción**.

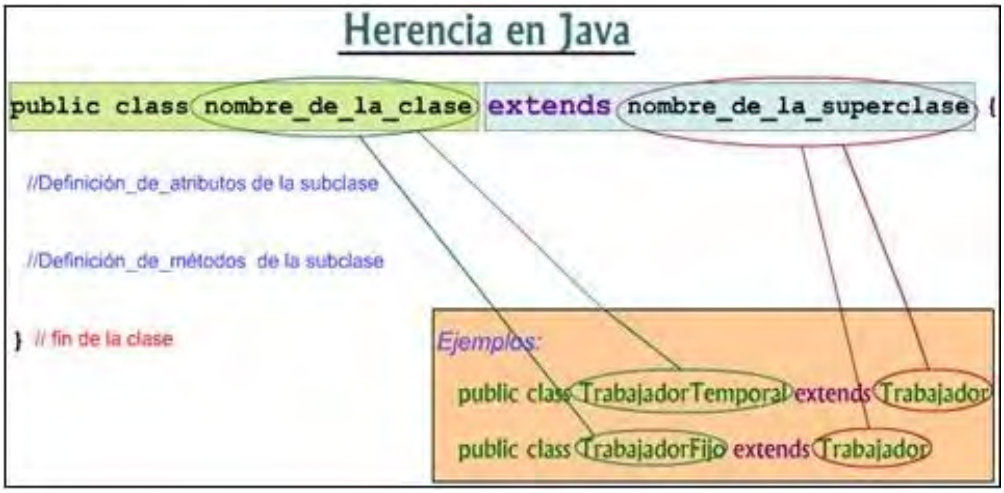
DEMO: Mira un ejemplo de la herencia múltiple

Autoevaluación

Sintaxis y particularidades de la herencia en Java.

Una vez que conocemos el **concepto de herencia** y las características básicas de este mecanismo o herramienta que nos proporciona la programación orientada a objetos, es hora de concretar cómo gestiona Java la herencia y qué sintaxis debemos emplear para explicitar que una clase hereda de otra.

Para establecer en Java que una clase hereda de otra clase ya existente, en la definición de la subclase utilizaremos el modificador `extends` en la cabecera de su definición, seguido del nombre de la clase que va a ser su superclase, de la siguiente manera:



En nuestro ejemplo de gestión de trabajadores, esto se traduciría en las siguientes cabeceras de definición de las clases TrabajadorFijo y TrabajadorTemporal:

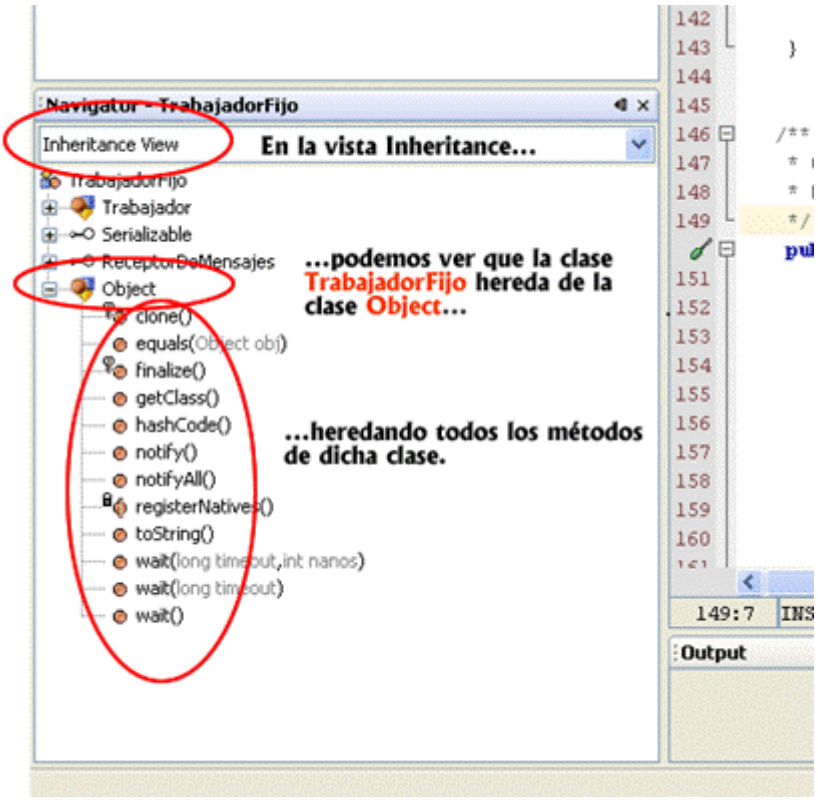
```
public class TrabajadorFijo extends Trabajador{

}

public class TrabajadorTemporal extends Trabajador{

}
```

A partir de ese instante las clases **`TrabajadorFijo`** y **`TrabajadorTemporal`** han heredado todos los atributos y métodos de la clase Trabajador, aparte de contar cada una con los métodos y atributos propios que se establezcan en su propia definición como clase.

Java no soporta herencia múltiple, por lo que una subclase sólo podrá heredar de una superclase y, consecuentemente, tras la palabra clave **`extends`** sólo podrá aparecer un único nombre de clase. **En Java una clase puede tener**

infinitas subclases directas pero, una clase tan sólo puede ser subclase directa de una única superclase. Por lo tanto, las jerarquías de clases que se pueden formar con Java tienen forma de árbol.

Todas las clases en Java, excepto la clase **`Object`**, extienden o heredan de una clase existente. **En Java, la jerarquía de clases empieza con la clase `Object`**, perteneciente al paquete **`java.lang`**, **a partir de la cual heredan todas las clases en Java, ya sea en forma directa o indirecta**. Si la declaración de una clase no especifica **`extends`** y el nombre de una clase a la derecha del nuevo nombre de la clase, esta nueva clase extiende implícitamente a la clase **`Object`**. Al ser todas las clases en Java subclases directas o indirectas de la clase **`Object`**, el lenguaje consigue que todas las clases tengan unas características comunes, pues absolutamente todas las clases que usemos van a contar con todos los atributos y los métodos que proporciona esta clase raíz de la jerarquía de clases que es la clase **`Object`**. Este hecho es el que permite, entre otras cosas, que funcione el recolector de basura en Java.



DEMO: Visualiza cómo acceder a la Inheritance view

A continuación tienes el código Java de las clases, **`TrabajadorFijo`** y **`TrabajadorTemporal`** que hemos presentado ya en la unidad y que iremos construyendo a lo largo de la misma. De momento, échale un vistazo al código de las tres clases, identificando qué es lo que añaden o modifican las subclases respecto a la superclase de la que heredan.

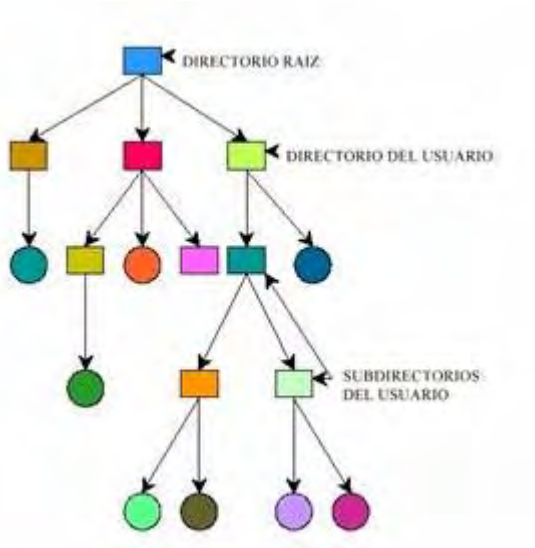
A partir de este momento, conforme vayamos exponiendo los contenidos de la unidad, iremos haciendo referencias a partes de este código. Una vez más, **te recomendamos que leas con atención el código del ejemplo, junto a los comentarios explicativos que en él se incluyen.**

Al igual que en unidades anteriores, te proporcionamos a continuación un enlace a la carpeta que contiene todo el proyecto, para que puedas seleccionar en el entorno NetBeans la opción abrir proyecto, y tener directamente cargadas todas las clases del ejemplo, listas para ejecutarlas sin más. Los ficheros que más interesa que estudies con atención son **`Trabajador.java`**, **`TrabajadorFijo.java`** y **`TrabajadorTemporal.java`**

Descarga el proyecto Trabajador (Carpeta completa)

Herencia y ocultación de información: control de acceso en Java

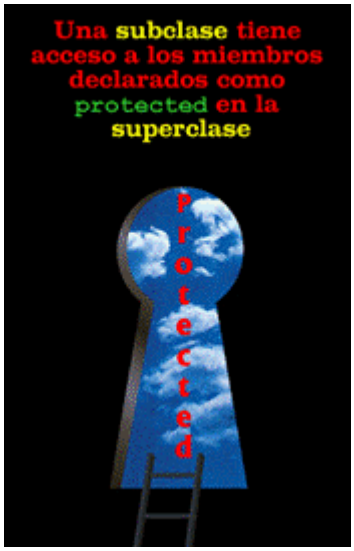
En el tema anterior hablamos sobre los modificadores de acceso en Java y cómo estos contribuían a la importante tarea de proporcionar mecanismos para la ocultación de información. Vimos que estos modificadores eran cuatro: **`public`**, **`package`**, **`private`** y **`protected`**; y vimos qué implicaciones en cuanto al acceso tenía el uso de cada uno de ellos. Una vez que ya conocemos la herencia, quizá sea el momento de recordar estas implicaciones enfocándolas a cómo afectan estos modificadores de acceso a las subclases. Veámoslo:



- **Cualquier miembro de la superclase cuyo acceso sea de tipo `public` podrá ser accedido desde la subclase**, pues recordemos que este modificador indica que al atributo o método que vaya precedido por él se podrá acceder desde cualquier otra clase incluidas, evidentemente, las subclases. Observa, por ejemplo, cómo en el método **`getAntigüedad()`** de la clase **`TrabajadorFijo`** se invoca sin problemas a los métodos públicos de la clase Trabajador **`getDiaAlta()`**, **`getMesAlta()`** y **`getAñoAlta()`**:

```
...
años = calendario.get(GregorianCalendar.YEAR) - this.getAñoAlta();
meses = calendario.get(GregorianCalendar.MONTH) + 1 - this.getMesAlta();
...
dias = calendario.get(GregorianCalendar.DAY_OF_MONTH) - this.getDiaAlta();
```

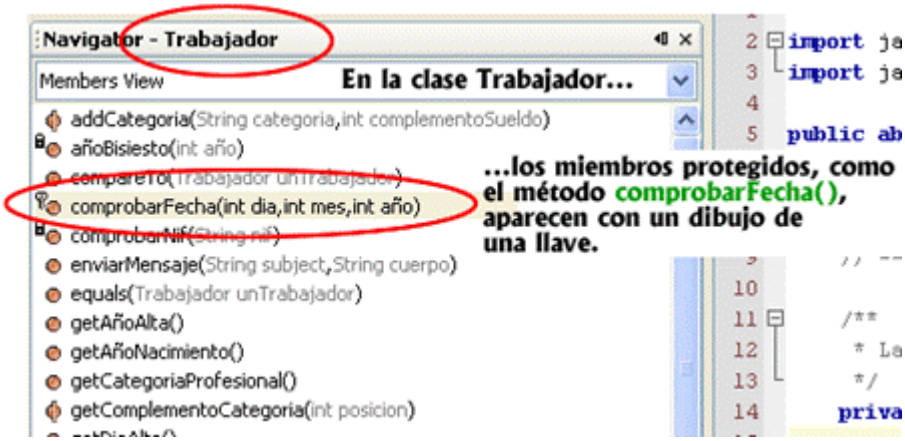
- **Los miembros de la superclase cuyo acceso sea de tipo `package` podrán ser accedidos desde la subclase siempre y cuando ésta pertenezca al mismo paquete**, pues recordemos que este modificador indica que el atributo o método que vaya precedido por él podrá ser accedido desde cualquier clase del mismo paquete, pero no desde una clase perteneciente a otro paquete. Por lo tanto, si una subclase no pertenece al mismo paquete que su superclase, entonces no podrá acceder a los miembros **`package`** de ésta.
- **Los miembros de la superclase cuyo acceso sea de tipo `private` no podrán ser accedidos desde la subclase bajo ningún concepto**, pues recordemos que este modificador indica que al atributo o método que vaya precedido por él sólo se podrá acceder desde la propia clase. Así, por **ejemplo**, desde las clases **`TrabajadorFijo`** o **`TrabajadorTemporal`** no podremos acceder directamente a ninguno de los atributos privados de **`Trabajador`** como son **`nif`**, **`nombre`**, **`diaNacimiento`**, **`comprobarNif()`**, **`añoBisiesto()`**, etc. Por lo tanto, al contrario de lo que ocurría en el código de la clase Trabajador, en el código de las subclases **`TrabajadorFijo`** o **`TrabajadorTemporal`** no podremos hacer referencia a estos miembros privados heredados, pues nos daría un error de sintaxis.



Tengamos en cuenta que si una subclase pudiese acceder a los datos privados de su superclase, las clases que heredasen de esa subclase también podrían acceder a ellos y, en definitiva, se propagaría el acceso a las partes declaradas como privadas, perdiéndose consecuentemente

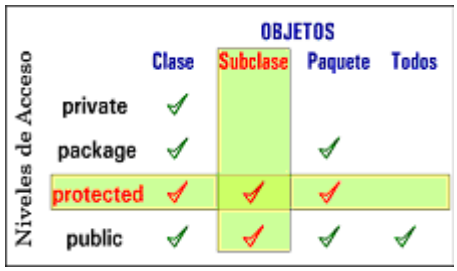
los beneficios del ocultamiento de la información. No obstante, sí que podremos acceder al valor de dichos atributos heredados utilizando los métodos públicos, también heredados, que los consultaban o modificaban. Así, por ejemplo, podemos establecer el nombre de un "trabajador fijo" haciendo uso del método público heredado **`setNombre()`**, y esto no viola el principio de ocultación de información. Es decir, **una subclase puede efectuar cambios de estado en los miembros `private` de la superclase, pero sólo a través de los métodos no privados que se proporcionen en la superclase y sean heredados por esa subclase**.

- Por su parte, **los miembros de la superclase cuyo acceso sea de tipo `protected` podrán ser accedidos desde todas sus subclases, pertenezcan éstas o no al mismo paquete**, pues indica que al atributo o método que vaya precedido por él sólo se podrá acceder desde la propia clase, desde sus subclases y desde las clases que pertenezcan a su mismo paquete. Así, por ejemplo, en la clase Trabajador contamos con los siguientes métodos protegidos: **`comprobarFecha()`** y **`posicionCategoria()`**. Observa por su parte cómo dichos métodos son utilizados respectivamente desde el método **`setFechaFinContrato()`** de su subclase **`TrabajadorTemporal`** y desde el método **`getSueldo()`** de su subclase **`TrabajadorTemporal`**.




```
public void setFechaFinContrato(int dia, int mes, int año){
  if (this.comprobarFecha(dia, mes, año)){
     diaFinContrato=dia;
     mesFinContrato=mes;
     añoFinContrato=año;
  }
  else{
     System.out.println("La fecha especificada no es correcta y no se ha asignado")
  }
}
public double getSueldo(){
  int posicion;
  posicion = this.posicionCategoria(getCategoriaProfesional());
  // Resto del código del método que no se muestra aquí
}
```

Estas **relaciones de visibilidad** quedan perfectamente reflejadas en la siguiente tabla, donde en las filas tenemos los distintos modificadores y marcamos con una equis en las columnas de las clases que tendrían acceso a un miembro con dicho modificador: la propia clase, las subclases, las clases del paquete y todas las clases, respectivamente.



O expresado de otra manera:

Todos los miembros públicos y protegidos de una superclase retienen su modificador de acceso original cuando se convierten en miembros de la subclase; es decir, los miembros `public` de la superclase se convierten en miembros `public` de la subclase y los miembros `protected` de la superclase se convierten en miembros `protected` de la subclase. Así, por ejemplo, en las clases `TrabajadorFijo` y `TrabajadorTemporal`, los métodos heredados de `Trabajador` `comprobarFecha()` y `posicionCategoria()` que estaban declarados con un acceso protegido, siguen manteniendo ese mismo acceso desde las subclases, y lo mismo sucede con los miembros públicos heredados como: `getNif()`, `setNif()`, `getNombre()`, `setNombre()`, etc., que mantienen su condición de públicos en las subclases.

DEMO: Descubre qué información puedes ver en la vista Members View

A la hora de decidir qué control de acceso establecer sobre un miembro de una superclase, **deberemos usar el modificador de acceso `protected` cuando una superclase deba proporcionar un servicio o método sólo a sus subclases y no a otros clientes**. Además, como ya comentamos en el tema anterior, los atributos deben ser declarados siempre como `private` para obtener las ventajas propias de la ocultación de la implementación. Por lo tanto, siempre **evitaremos incluir atributos protegidos en una superclase** y, en vez de ello, los declararemos como privados e incluiremos métodos no privados (públicos o protegidos) que accedan a ellos. Con esta estrategia de programación conseguiremos que la implementación en la superclase pueda cambiar sin afectar a las implementaciones de las subclases.

Autoevaluación

Constructores en las subclases

Todas las clases, ya sean éstas superclases o subclases, tienen sus propios constructores, cuya finalidad es la de inicializar los datos del objeto. **En el caso de las subclases, el constructor no sólo inicializa sus propios datos, sino que tiene que hacerlo también con los datos que proporciona la superclase**. Además, el orden en el que se tienen que inicializar estas variables es el mismo orden que impone la jerarquía: **primero se deben inicializar los valores de la superclase y luego los específicos de la subclase**.

¿Cómo se pueden inicializar los atributos heredados de la superclase desde la subclase, máxime cuando probablemente dichos atributos serán privados y, por lo tanto, inaccesibles desde la subclase? La respuesta es bien sencilla: **invocando desde el método constructor de la subclase al método constructor de la superclase**. Para ello, se debe utilizar la palabra reservada **`super`** que nos ofrece Java, mediante una sentencia del tipo siguiente, donde **`lista_parametros`** son los parámetros que necesita el constructor de la superclase:

```
super (lista_parametros);
```

Por lo tanto, **al crear la instancia del objeto de una subclase se genera una cadena de llamadas a los constructores, en donde cada constructor manipula sus propias variables de instancia**. Así:

- El **constructor de la subclase,** antes de realizar sus propias tareas, invoca al constructor de su superclase directa, ya sea en forma explícita, mediante la referencia **`super`**, o implícitamente, llamando al constructor predeterminado de la superclase, es decir, a su constructor sin argumentos.
- De manera similar, si la superclase es una clase derivada de otra clase, el constructor de la superclase tendrá que invocar al constructor de la siguiente clase que esté un nivel arriba en la jerarquía, y así sucesivamente.
- Como todas las clases derivan finalmente de la clase **`Object`**, el último constructor llamado en esta cadena siempre es el constructor de dicha clase.
- Por último, se ejecuta el cuerpo del constructor de la subclase instanciada.

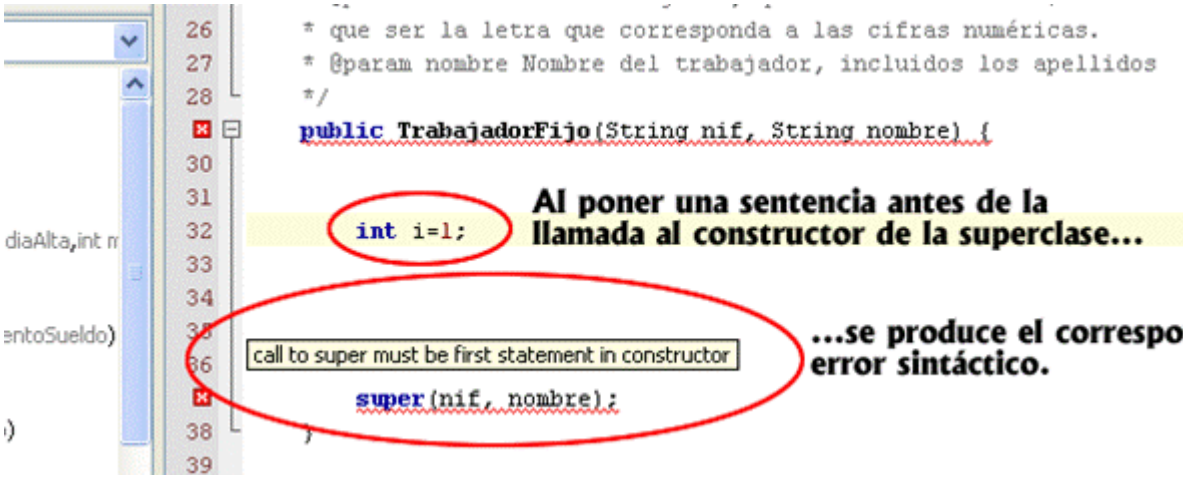
Observa cómo en la subclase **`TrabajadorTemporal`** se hace uso de la llamada **`super`** para invocar al constructor de la superclase antes de realizar la inicialización de las variables propias de la subclase, como son las variables de la fecha de fin del contrato:

```
public TrabajadorTemporal(String nif, String nombre, int diaAlta, int mesAlta, int a
                          int diaFinContrato, int mesFinContrato, int añoFinCon
        super(nif, nombre, diaAlta, mesAlta, añoAlta);
        if(Trabajador.getObjetoCreado()){
            if (this.comprobarFecha(diaFinContrato, mesFinContrato, añoFinContrato)
                this.diaFinContrato=diaFinContrato;
                this.mesFinContrato=mesFinContrato;
                this.añoFinContrato=añoFinContrato;
```

```
            }
            else{
               // Por defecto si la fecha introducida no es correcta se la asigna el valo
                this.diaFinContrato=0;
                this.mesFinContrato=0;
                this.añoFinContrato=0;
            }
        }
    }
```

Además, debemos tener siempre presente lo siguiente:

1. **La sentencia de llamada al constructor de la superclase debe aparecer siempre en primer lugar en el código de implementación del constructor de la subclase**. En caso contrario, se producirá un error en tiempo de compilación, pues constituye un error de sintaxis. Para probar in situ esta afirmación puedes añadir alguna sentencia antes de la sentencia **`super`** que encontrarás en cualquiera de los constructores de las subclases **`TrabajadorFijo`** y **`TrabajadorTemporal`**, podrás comprobar que se produce un error de compilación.



2. Sabemos que **una clase puede tener definidos varios constructores distintos**. Entonces, cuando hacemos una llamada al constructor de la superclase a través de la palabra clave **`super`**, ¿a cuál de los constructores de la superclase estaremos invocando? Como sabemos, cada uno de los constructores de una clase debe tener una lista de parámetros distinta. Por lo tanto, **según el número y tipo de los parámetros el sistema será capaz de localizar el constructor de la superclase al que se está haciendo referencia**.
3. **Si no se utiliza en el constructor de la subclase una llamada explícita a uno de los constructores de la superclase, entonces se llamará automáticamente y de manera implícita al constructor por defecto de la superclase**. Como sabemos por lo que aprendimos en la unidad anterior, el constructor por defecto es un constructor sin parámetros que el compilador crea de manera automática **únicamente** en el caso de que no se hubiera definido en la clase ningún otro constructor. Por lo tanto, debemos de llevar cuidado, porque si la superclase tiene definidos métodos constructores y, consecuentemente, no tiene ningún constructor por defecto, estaremos obligados a hacer una llamada explicita con **`super`** a uno de los constructores de dicha superclase; en caso contrario, se producirá un error en tiempo de compilación, pues la llamada implícita al constructor por defecto no encuentra dicho constructor por defecto.

Redefinición de métodos en las subclases

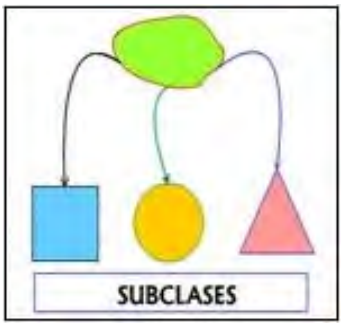
Como bien sabemos, en nuestra aplicación de gestión de trabajadores tenemos definidos dos tipos de trabajadores, los fijos y los temporales, los cuales heredan de la clase **`Trabajador`**, absorbiendo las características comunes que los define como trabajadores de nuestra empresa.

Una de estas características comunes es la tener un sueldo mensual, el cual puedo consultar haciendo uso del método **`getSueldo()`** de la clase **`Trabajador`**. Dicho método, contiene en su implementación la fórmula que establece cómo se calcula el sueldo de un empleado, lo cual depende de diversos factores. Los distintos empleados de la empresa tendrán sueldos distintos según dicte el resultado de aplicar dicha fórmula. Este método, evidentemente, es heredado por las subclases y puede ser utilizado para averiguar el sueldo de un "trabajador fijo" y el sueldo de un "trabajador temporal".

Pero, **¿qué sucede si la fórmula a aplicar para obtener lo que debe cobrar un empleado con contrato temporal debe ser distinta a la fórmula a aplicar a los trabajadores con contrato fijo?** Por ejemplo, imagina que:

1. El sueldo de los trabajadores con contrato fijo deba calcularse como la suma del salario base, el complemento asociado a la categoría profesional que se tenga, 50 euros por hijo y 20 euros por año de antigüedad en la empresa.
2. Mientras que el sueldo de los trabajadores con contrato temporal deba calcularse como la suma entre el salario base y el complemento asociado a la categoría profesional que se tenga, simplemente.

Si el método **`getSueldo()`** viene heredado de la superclase y, por tanto, tenemos una misma implementación heredada por ambas subclases, ¿cómo puedo dar cabida a esta divergencia entre las fórmulas a aplicar según el tipo de trabajador? La solución a nuestro problema nos la proporciona la posibilidad que nos ofrece la programación orientada a objetos de redefinir en las subclases los métodos heredados de la superclase. **Llamamos redefinición de un método a la posibilidad de cambiar en una subclase la implementación de un método heredado de la superclase**.



No obstante, el caso del método **`getSueldo()`** es un caso especial de redefinición de métodos que veremos más adelante cuando hablemos de las clases abstractas. Por lo tanto, vamos a dejar aparcado dicho ejemplo y vamos a centrar el resto de la exposición de lo que es la redefinición en el ejemplo del método **`toString()`**, método heredado de la clase **`Object`** y cuya función en nuestras clases va a ser la de generar una cadena de texto con la información más importante que caracteriza al trabajador en cuestión.

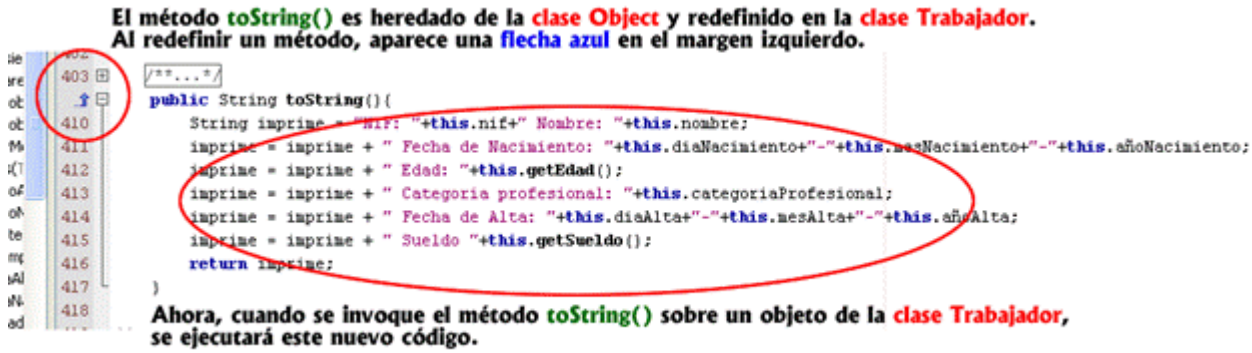
El método **`toString()`** es heredado de la clase **`Object`** y la implementación que nos proporciona la superclase no nos sirve, por lo que necesitamos redefinirlo en la clase **`Trabajador`** para darle la funcionalidad que necesitamos, en este caso para que elabore una cadena de texto con información como el NIF, el nombre, la fecha de nacimiento, la edad, la categoría profesional, la fecha de alta en la empresa y el sueldo del trabajador. **Para redefinir un método sólo tenemos que poner en el código de la clase un método con el mismo nombre y signatura que el método a redefinir, junto a su nueva implementación**. Así, para redefinir el método **`toString()`**, pondríamos lo siguiente:

```
public String toString(){
```

```
        String imprime = "NIF: " + this.nif + " Nombre: " + this.nombre;
        imprime = imprime + " Fecha de Nacimiento: " + this.diaNacimiento + "-" +
                  this.mesNacimiento + "-" + this.añoNacimiento;
        imprime = imprime + " Edad: " + this.getEdad();
        imprime = imprime + " Categoria profesional: " + this.categoriaProfesional;
        imprime = imprime + " Fecha de Alta: " + this.diaAlta + "-" + this.mesAlta + "
                  this.añoAlta;
        imprime = imprime + " Sueldo " + this.getSueldo();
        return imprime;
    }
```

DEMO: Mira lo que ocurre con el método redefinido

A partir de este momento el método **`toString()`** ha quedado redefinido y ahora, cada vez que sea invocado sobre un objeto de la clase **`Trabajador`** este nuevo código será el que se ejecute.



Cuando construimos las subclases de la clase **`Trabajador`**, **`TrabajadorFijo`** y **`TrabajadorTemporal`**, éstas heredan el método **`toString()`** con la misma implementación que se le dio a éste en la clase Trabajador, pues **las subclases heredan los métodos redefinidos en su superclase con la implementación que en ella se les da a los mismos**. Sin embargo, esta implementación heredada de **`Trabajador`** para el método **`toString()`** no es válida en las subclases pues, por ejemplo, para los empleados con contrato fijo también nos interesa que se incluya en esa cadena de información la antigüedad y el número de hijos del trabajador. Por su parte, para los empleados con contrato temporal nos interesa mostrar también información acerca de la fecha de fin de contrato. Por lo tanto, es preciso volver a redefinir nuevamente el método **`toString()`** en cada una de las subclases **`TrabajadorFijo`** y **`TrabajadorTemporal`**.

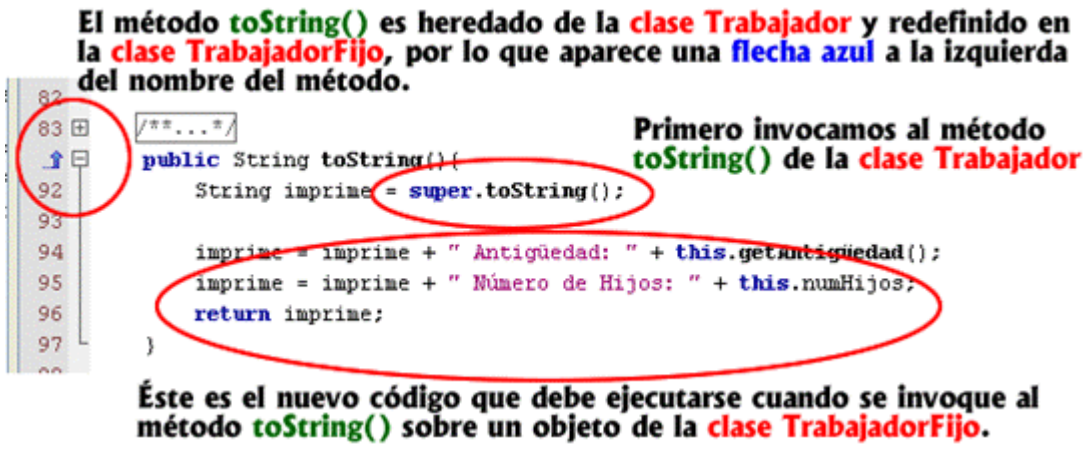

Correos

¿Qué es lo que tiene que hacer el método **`toString()`** en las subclases de la clase **`Trabajador`**? Si lo pensamos, exactamente lo mismo que en la clase **`Trabajador`**, más algunas cosas añadidas, en este caso añadir a la cadena la información particular de la subclase correspondiente. La pregunta que se plantea entonces es la siguiente: ¿tenemos que repetir este código otra vez al redefinir el método o podemos reutilizar el código de la superclase? Como puedes imaginarte, la respuesta es que sí, que podemos reutilizarlo. Veamos cómo.

Cuando desde una subclase se quiere invocar a un método redefinido de la

superclase, se debe emplear la palabra clave `super` seguida de un punto y el nombre del método de la superclase al que se quiere invocar. Así, por ejemplo, para invocar al método **`toString()`** de la superclase desde el método **`toString()`** de la subclase **`TrabajadorFijo`** haremos lo siguiente:

```
public String toString(){
        String imprime = super.toString();
        imprime = imprime + " Antigüedad: " + this.getAntigüedad();
        imprime = imprime + " Número de Hijos: " + this.numHijos;
        return imprime;
}
```



Por su parte, el método **`toString()`** de la clase **`TrabajadorTemporal`** quedaría como sigue:

```
public String toString(){
        String imprime = super.toString();
        imprime = imprime + " Fecha de Fin de Contrato: "+ this.diaFinContrato + "-
                  this.mesFinContrato + "-" + this.añoFinContrato;
        return imprime;}
```

Como puedes ver, la posibilidad de redefinir métodos en las subclases otorga una **flexibilidad** al mecanismo de herencia que lo hace, si cabe, más potente.

Hay una serie de cuestiones que debemos tener muy presentes a la hora de redefinir un método heredado para no cometer errores de programación:

1. **Para redefinir un método de la superclase en una subclase, éste tiene que tener exactamente los mismos parámetros que en la superclase**; es decir, debe tener el mismo número y del mismo tipo. Si en una subclase se define un método con el mismo nombre que uno que ya proporciona la superclase pero con distintos parámetros, entonces lo que se está haciendo es sobrecargar el método en la subclase, y no redefiniendo el de la superclase.
2. Además, **cuando se redefine un método en una subclase, no se puede cambiar su modificador de acceso a uno más restrictivo que el que tenía en la superclase, aunque sí a uno menos restrictivo**. Así, por ejemplo, si un método que en la superclase está definido como protegido quisiésemos convertirlo en un método público en la redefinición, podríamos hacerlo, pues

el modificador **`public`** es menos restrictivo que **`protected`**. Sin embargo, si quisiésemos convertirlo en un método privado, nos daría un error en tiempo de compilación, al estar intentando hacerlo más restrictivo en su acceso. Evidentemente, dejarlo con el mismo modificador de acceso, **`protected`**, está permitido y, de hecho, eso es lo que hace el lenguaje por defecto: mantener el nivel de acceso para los miembros públicos y protegidos. En el cuadro siguiente tienes un resumen ilustrativo de qué cambios en el modificador de acceso de un miembro están permitidos al redefinirlo y cuáles no:

SUPERCLASE \ SUBCLASE	private	package (por defecto)	protected	public
private	PERMITIDO	PERMITIDO	PERMITIDO	PERMITIDO
package	PROHIBIDO	PERMITIDO	PERMITIDO	PERMITIDO
protected	PROHIBIDO	PROHIBIDO	PERMITIDO	PERMITIDO
public	PROHIBIDO	PROHIBIDO	PROHIBIDO	PERMITIDO

La palabra clave "final"

*Durante la preparación de una aplicación es habitual ayudarse de materiales como libros de referencia o enlaces en Internet, sobre todo cuando intentamos buscar ejemplos de uso de determinados componentes o clases. Éste es el caso de **Víctor** que se ha aventurado a buscar en la red alguna definición de clases similar a lo que **Carmen** le ha pedido que haga con las subclases de la clase **Trabajador**. Pero esto suele traer otros problemas y es la aparición de conceptos que no conoce y que resulta complicado entender. Por ejemplo el concepto de "constantes blancas" o métodos de tipo "final", e incluso clases con el modificador "final". Incluso la ayuda que ha encontrado sobre esto no le aclara nada. **Carmen** le tranquiliza y le explica que es un término muy usado y le explica en qué consiste su utilidad.*

La palabra clave **`final`** tiene en Java distintos significados, según a qué elemento sea aplicada. Ya fue mencionada en el tema anterior, pero es ahora, una vez que conocemos en profundidad lo que es la herencia, el momento de detenernos para realizar un estudio minucioso de su uso en sus diferentes contextos:

1. **Una variable de instancia puede ir precedida en su declaración del modificador `final`**, queriendo esto decir que es una variable que debe ser inicializada en el método de creación del objeto y, además, que no podrá ser modificada en ningún otro método. Imagina una clase **`Ordenador`** que me sirviese para modelar los ordenadores de un aula de informática. Cada ordenador viene con un número de serie, que es único (no habrá dos ordenadores con el mismo número de serie) y que no cambia durante todo la existencia del ordenador. Esta situación es ideal para que usemos en nuestro programa una variable de instancia final, donde cada objeto **`Ordenador`** puede darle valor a la

variable en el momento de su creación y mantenerlo inalterado durante todo el tiempo de vida del objeto. Este tipo de variables actúan, en cierto modo, como **constantes**, sólo que no son constantes con el mismo valor para todos los objetos, sino constantes con un valor particular para cada objeto. Es por esta similitud por la que este tipo de variables reciben también el nombre de **constantes blancas**. La palabra clave `final` deberá aparecer después del modificador de acceso del atributo, y antes del tipo del mismo.

```
[private | protected | public] final tipo_atributo nombre_atributo;
```

2. **Un método de una clase también puede ir acompañado del modificador** `final`, queriendo esto decir que el método no puede ser redefinido en las subclases. En este caso, la palabra clave `final` también aparece detrás del modificador de acceso del método y antes del tipo del valor de retorno del mismo.
```
[private | protected | public] final tipo_retorno nombre_método(parámetros) excepciones;
```
Cualquier intento de redefinir un método **final** en una subclase producirá un error de compilación.
3. **Un parámetro de un método puede ir acompañado del modificador** `final`, queriendo esto decir que no se puede modificar su valor dentro del método. En caso de que el parámetro sea una referencia a un objeto, dicha referencia no puede variar, es decir, no se puede hacer que apunte a otro objeto distinto al que apunta cuando se entra al método. En este caso la palabra clave **final** aparece antes del tipo del parámetro.
```
modificadores tipo_retorno nombre_método(final tipoParámetro nombreParámetro, ...) excepciones;
```
4. Por último, **también una clase puede ser definida como** `final`. En este caso, el modificador impide que la clase en cuestión pueda ser superclase de ninguna otra, pues cualquier intento de que otra clase la extienda o herede de ella producirá un error en tiempo de compilación. Al igual que en las anteriores ocasiones, el modificador aparecerá inmediatamente después del modificador de acceso, y antes de la palabra reservada `class`.
```
[public] final class Nombre_de_la_clase [extends superclase] implements lista_de_interfaces
```

Evidentemente, todos los métodos de una clase `final` son implícitamente `final`, pues al no poder haber subclases, éstas no podrán redefinir los métodos.

El polimorfismo

__Víctor__ se encuentra ante un problema con la definición de las subclases __TrabajadorFijo__ y __TrabajadorTemoporal__. Básicamente ha decidido definir dos clases y cuando crea un nuevo trabajador lo define de una u otra clase. Pero __Carmen__ le explica que está cometiendo un error, ya que no está aprovechando las posibilidades de la herencia con la clase __Trabajador__. Le explica que es necesario hacer uso del polimorfismo, de modo que cuando crea una instancia de __Trabajador__ adopta las características de esta clase, para después heredar además las de la subclase correspondiente. Es como si el trabajador fijo tuviera un aspecto diferente al temporal (por ejemplo un uniforme) que adquiere con la subclase.

Según establece el diccionario de la Real Academia de la Lengua Española, **polimorfismo es la propiedad de ciertos cuerpos de cambiar de forma sin variar su naturaleza**. Dichos cuerpos reciben el nombre de polimorfos o polimórficos, y pueden con frecuencia pasar de una a otra de sus formas.

Seguro que ahora mismo te asaltan multitud de preguntas: ¿qué tiene que ver todo esto con la programación orientada a objetos? ¿Existe la posibilidad de tener elementos polimórficos en nuestros programas? ¿Para qué me puede servir en un programa contar con un elemento que pueda cambiar de forma? Éstas y muchas otras son las preguntas a las que trataremos de dar respuesta a lo largo de este apartado del tema.

Características básicas del polimorfismo

Anteriormente en este tema abordamos la **herencia**, mecanismo que permite crear jerarquías de clases de tal manera que:

1. Una clase superior a otras en la jerarquía captura y recoge las características y comportamiento común a sus clases descendientes.
2. Consecuentemente, se evita tener que repetir en las clases descendientes lo ya establecido en la clase de la cual proceden, pudiéndose centrar éstas en la programación de sus hechos diferenciales.

Debemos recordar que **mediante el mecanismo de herencia se establece entre las clases implicadas una relación del tipo "es un"**, queriendo esto decir que:

1. Un objeto de la subclase es también un objeto de la superclase y, consecuentemente, puede ser tratado como tal.
2. Aunque, un objeto de la superclase no puede ser tratado como un objeto de la subclase, pues no soporta las novedades o especializaciones que aporta ésta.

Detengámonos un instante en la primera de las anteriores afirmaciones: "**Un objeto de la subclase es también un objeto de la superclase**". Es decir, trasladado a nuestro ejemplo de gestión de trabajadores, estamos afirmando que un "trabajador fijo" es un "trabajador" y que un "trabajador temporal" es un "trabajador". Esto quiere decir que un objeto que sea del tipo `TrabajadorFijo` podrá ser tratado como si fuese un objeto del tipo `Trabajador`, ya que tiene absolutamente todas las características y comportamiento de éstos. Eso mismo sucede con un objeto del tipo `TrabajadorTemporal`.

Hasta aquí nada que no supiéramos ya. Pero vayamos un poco más lejos en este razonamiento mirando ahora desde el otro lado. Si quisiese trabajar con objetos del tipo `Trabajador`, haciendo uso exclusivamente de las características y comportamiento de los objetos de este tipo, ¿importaría que éstos fuesen realmente objetos del tipo `TrabajadorFijo` o `TrabajadorTemporal`? Si lo pensamos detenidamente, realmente no, pues los objetos de estos tipos también dan soporte a esas características y comportamiento. Lo único que necesitaríamos es que el lenguaje de programación nos proporcionase algún mecanismo que nos permitiera gestionar todos los objetos Trabajador de la misma manera, independientemente de que éstos fuesen realmente objetos `TrabajadorFijo` u objetos `TrabajadorTemporal`; es decir, independientemente de la forma específica que adopte el "trabajador". ¡Oye! Lo que le estamos pidiendo al lenguaje de programación, ¿no es que soporte el polimorfismo? Pues sí, efectivamente; de hecho, ése es exactamente **el nombre que recibe el mecanismo de la programación orientada a objetos que permite actuaciones como la descrita: polimorfismo**.

Podemos definir el **polimorfismo** como la **posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una subclase heredada de la anterior**, lo cual nos va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía de clases, como si todos fueran objetos de sus superclases.

Observa el código de la clase `GestionTrabajadores` de nuestro programa de gestión, la cual supone el módulo principal de nuestra aplicación y la que se va a encargar de

manipular objetos de tipo **Trabajador** o de sus subclases. Observa el código que aparece en el método **añadir()**, método que se encarga de crear nuevos objetos trabajador para incluirlos en la lista de trabajadores de la empresa:

- Fíjate cómo se declara una variable llamada **unTrabajador**, que es una variable de clase **Trabajador**. Dicha variable contiene inicialmente una referencia vacía, pues en un principio no está ligada a ningún objeto.
  ```
  Trabajador unTrabajador = null;
  ```
- Observa cómo se crea un objeto de la clase **TrabajadorFijo** si el usuario del programa decide que el trabajador al que va a dar de alta en la empresa es un trabajador con contrato fijo. Sin embargo, observa que dicho objeto creado es asignado a la variable anterior **unTrabajador**, que era una variable de tipo **Trabajador**.
  ```
  unTrabajador = new TrabajadorFijo(nif, nombre, dia, mes, año, numHijos);
  ```

De esta forma, una variable que es de clase **Trabajador** referencia realmente a un objeto de la clase **TrabajadorFijo**. Esto es un claro ejemplo de lo que es el polimorfismo, **donde una variable cuyo tipo es el de una superclase puede hacer referencia a objetos que son de la subclase**.

- Por otra parte, observa cómo se crea un objeto de la clase **TrabajadorTemporal** si el usuario del programa decide que el trabajador al que va a dar de alta en la empresa es un trabajador con contrato temporal. Y lo que es más importante, observa que dicho objeto creado es asignado también a la variable **unTrabajador**, que es del tipo **Trabajador**.
  ```
  unTrabajador = new TrabajadorTemporal(nif, nombre, dia, mes, año,
  diaFinContrato, mesFinContrato, añoFinContrato);
  ```

Por lo tanto, observamos cómo la variable **unTrabajador** adopta una forma u otra según el tipo de trabajador que se quiera crear, lo cual es la esencia misma del polimorfismo: **una variable tiene un tipo definido en tiempo de compilación, pero luego, en tiempo de ejecución puede estar referenciando a objetos de tipos distintos, con la única condición de que estos tipos sean subclases directas o indirectas del tipo definido en tiempo de compilación para la variable**.

- Una vez creado el objeto del tipo de trabajador que corresponda, dicho trabajador es insertado en la lista que contiene a todos los trabajadores de la empresa:
  ```
  lista.insertarNodo(new Nodo(unTrabajador));
  ```

Detengámonos un instante a reflexionar. Si cada vez que creo un trabajador en mi aplicación, inserto éste en la variable de nombre **lista**, ¿qué tipo de objetos contiene esta lista de trabajadores de la empresa? Evidentemente dentro de la lista habrá tanto objetos **TrabajadorFijo** como objetos **TrabajadorTemporal**, según el tipo de trabajador que se haya creado en cada instante. Es decir, **contamos con una estructura de datos que contiene objetos de distintas clases y se dice entonces que se trata de una estructura de datos polimórfica o un contenedor polimórfico**. Y lo que es más importante, **¡no he tenido que hacer absolutamente ningún cambio en la definición de esta lista, es decir, en la clase Nodo, que al estar definida para almacenar objetos de la superclase, está automáticamente preparada para acoger objetos de cualquiera de sus subclases!**

- Vayamos ahora un poco más adelante en el código, al lugar donde se hace uso de la variable **unTrabajador** una vez que ésta ya hace referencia a un objeto. Por ejemplo, observa sentencias como las siguientes:

  ```
  unTrabajador.setFechaNacimiento(nuevoDiaNacimiento, nuevoMesNacimiento, nuevoAñoNac

  ...

  unTrabajador.setCategoriaProfesional(nuevaCategoria);
  ```

Observa cómo **las sentencias son exactamente las mismas, independientemente de si la variable unTrabajador está referenciando a un objeto de la clase TrabajadorFijo o a un objeto de la clase TrabajadorTemporal**. Evidentemente, esto es así, porque **estamos haciendo uso de características y comportamiento propios de la superclase y, por tanto, comunes y soportados por todas las subclases**.

Esta posibilidad de **abstracción** que nos abre el polimorfismo, al permitirnos poder **programar hacia la generalidad en vez de programar para casos particulares y concretos**, es lo que otorga a la programación orientada a objetos toda su potencia como **filosofía de programación extensible**. La experiencia en la creación de sistemas de software nos indica que cantidades considerables de código tratan con casos especiales y concretos, y esa programación orientada al detalle puede oscurecer el panorama general y hacer programas que sean difíciles de reutilizar y de extender. Con la programación orientada a objetos, los programadores deben enfocar su análisis y su programación a los elementos comunes entre los objetos del sistema en vez de enfocarse a los casos especiales. Deben tratar de **realizar un procedimiento de abstracción y programar para la generalidad, siempre que puedan hacerlo, de tal manera que sus programas funcionen igualmente y sin modificaciones para todos los casos particulares cuando se trabaja sobre los aspectos que son comunes a todos ellos**. Sólo tendrán que programar de forma específica cuando realmente es imprescindible, cuando se trabaja sobre aspectos particulares de una clase específica.

Identificación de tipos en tiempo de ejecución y conversión descendente

Acabamos de ver cómo el **polimorfismo** se aprovecha de la propiedad de que un objeto de la subclase sea también un objeto de la superclase para abrirnos de par en par las puertas de la extensibilidad. Pero si recuerdas, la afirmación anterior: "un objeto de la subclase es también un objeto de la superclase y, consecuentemente, puede ser tratado como tal", lleva aparejada otra no menos importante: "**un objeto de la superclase no puede ser tratado como un objeto de la subclase, pues no soporta las novedades o especializaciones que aporta ésta**".

Es evidente que un objeto de la clase Trabajador no puede ser tratado como un objeto de la clase **TrabajadorFijo**, o como un objeto de la clase `TrabajadorTemporal`, pues no tiene las características propias de estos tipos específicos de trabajadores. Así, por ejemplo, sería un error tratar de invocar el método `getNumHijos()` sobre un objeto `Trabajador`, pues ésta es una característica exclusiva de los trabajadores con contrato fijo. De hecho, **invocar un método exclusivo de una subclase sobre una variable del tipo de la superclase produce un error en tiempo de compilación**.

Podríamos entonces plantearnos la siguiente pregunta: **¿qué sucede en el caso de que realmente, en tiempo de ejecución, esa variable del tipo de la superclase contuviera una referencia a un objeto de la subclase, tal y como le permite el polimorfismo?**

- Es decir, ¿qué sucede si una variable del tipo `Trabajador` contuviese una referencia a un objeto de la clase `TrabajadorFijo`?
- En ese caso, ¿se podría invocar el método `getNumHijos()` sobre dicho objeto? En principio, parece lógico pensar que sí.

Sin embargo, el tipo del objeto al que referencia realmente una variable polimórfica sólo se conoce en tiempo de ejecución, por lo que, **¿quién puede asegurar que cuando se intente invocar al método en cuestión, la variable no va a referenciar a un objeto de otro tipo distinto al esperado, produciéndose consecuentemente un error en la ejecución?** Con la aparición del polimorfismo el compilador pierde todo el control sobre el tipo real que contiene una variable y tendría que ser el programador el que controlase en su código que no se pudiese producir este tipo de errores. Delegar este tipo de responsabilidades en el programador supone correr un gran riesgo, por lo que en Java, como la mayoría de los demás lenguajes de programación orientada a objetos, ha optado por impedirlo. Por lo tanto, **si una variable ha sido declarada con el tipo de la superclase, sólo se podrán invocar métodos propios de la superclase sobre el objeto al que hace referencia, aunque en tiempo de ejecución dicha variable esté haciendo referencia a un objeto de una de sus subclases**.

Vayamos ahora a nuestro programa de gestión de trabajadores. Una de las

opciones que éste proporciona al usuario es la de poder modificar los datos de un trabajador una vez que éste ya está dado de alta en la empresa. El código del programa que se hace cargo de esta función está recogido en el método **modificarTrabajador()** de la clase **GestionTrabajadores**. Para poder modificar los datos de un trabajador:

1. Lo primero que debe hacerse es localizar al objeto que lo representa entre todos los trabajadores que hay en la empresa. Dicha búsqueda se realizará por **NIF**.
2. Todos los trabajadores están almacenados en el programa en la variable **lista**.
3. Dicha variable **lista** es polimórfica, estando definida como un contenedor de objetos **Trabajador** y, por lo tanto, puede contener realmente tanto referencias a objetos **TrabajadorFijo** como referencias a objetos **TrabajadorTemporal**.
4. Una vez localizado el objeto trabajador buscado, invocaremos sobre él los métodos que sean necesarios para poder cambiar la información que deseamos cambiar del mismo.

Como hemos dicho, la variable **lista** está definida como un contenedor de objetos **Trabajador**, por lo que para recorrerla y trabajar con ella, tenemos que hacerlo en base a objetos de dicha clase. Según la premisa con la que comenzamos este apartado, esto quiere decir que solamente podremos invocar los métodos propios de la clase **Trabajador** y, consecuentemente, sólo podremos cambiar la información que es común a todos los tipos de trabajadores. Entonces, si el trabajador es uno con contrato fijo, ¿cómo puedo hacer para cambiar la información específica de este tipo de trabajador como es, por ejemplo, el número de hijos? Veámoslo:

- Primeramente, debo de disponer de algún mecanismo para averiguar de qué tipo de trabajador se trata: si de un "trabajador fijo" o si de un "trabajador temporal". **Java me ofrece un operador lógico, llamado instanceof, que me permite saber si un objeto dado es de una clase determinada**; es decir, que me permite identificar el tipo de un objeto en tiempo de ejecución. El operador irá antecedido por una variable que referencie a un objeto, e irá sucedido por el nombre de una clase.

```
variable instanceof NombreClase
```

Entonces, **se devolverá el valor verdadero si y sólo si el objeto referenciado por la variable es compatible con la clase evaluada; es decir, si es de la clase NombreClase o de alguna de sus subclases**. En caso contrario, la comprobación devolverá falso.

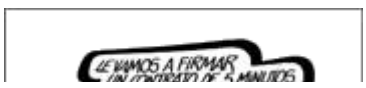
Así, por ejemplo, en el método **modificarTrabajador()**, cuando queremos comprobar si el objeto Trabajador localizado en la lista y referenciado por la variable **personaAModificar.dato** es del tipo **TrabajadorFijo**, hacemos lo siguiente:

```
if (personaAModificar.dato instanceof TrabajadorFijo){

}
```

- Una vez que sabemos la clase del objeto con el que estamos tratando, si es un "trabajador temporal" o un "trabajador fijo", entonces podemos proceder a cambiar la información particular de ese tipo de trabajador. Sin embargo, esto no puedo hacerlo directamente sobre la variable de tipo Trabajador, pues, como hemos comentado, sobre ella sólo puedo invocar métodos propios de esa clase, aunque realmente haga referencia a un objeto de una de sus subclases. Podría pensar en asignar el objeto directamente sobre una variable de trabajador concreto con el que tengo que trabajar. Haciendo ésto estaríamos hablando ya de un "trabajador fijo":

```
TrabajadorFijo unTrabajadorFijo = personaAModificar.dato;        // ESTO NO ES CORREC
```

Sin embargo, esto produciría un error en tiempo de compilación, pues los tipos de ambas variables en la asignación, la de origen y la de destino, no es el mismo, aún cuando el tipo del objeto al que referencia



la variable de origen sí que es el mismo que el tipo de la variable de destino. **Para salvar este escollo en Java, sólo tengo que anteponer a la variable origen, y entre paréntesis, el nombre de la clase de la variable destino**.

```
TrabajadorFijo unTrabajadorFijo = (TrabajadorFijo) personaAModificar.dato;
```

Este tipo de operación recibe el nombre de **casting o de conversión de tipos y**, evidentemente, **sólo puede realizarse con éxito si la variable origen está referenciando en tiempo de ejecución a un objeto de la clase de la variable destino**, pues, de otro modo, se produciría un error definido por la excepción **`ClassCastException`**.

- Una vez que ya disponemos de una variable del tipo de la subclase haciendo referencia al objeto, podemos invocar a los métodos propios de la subclase para cambiar la información particular de ese tipo de trabajador que el objeto representa. Así, por ejemplo, en el caso del "trabajador fijo" podría ejecutar una sentencia como la siguiente:

```
unTrabajadorFijo.setNumHijos(nuevoNumHijos);
```

Observa ahora, en su conjunto, cómo queda el código completo para el tratamiento particular de los trabajadores de la empresa:

```
if (personaAModificar.dato instanceof TrabajadorFijo){
   TrabajadorFijo unTrabajadorFijo = (TrabajadorFijo) personaAModificar.dato;
   int nuevoNumHijos=ES.leeNº("Introduce el número de hijos del empleado: ", 1, 20)
   unTrabajadorFijo.setNumHijos(nuevoNumHijos);
}
else if (personaAModificar.dato instanceof TrabajadorTemporal){
   TrabajadorTemporal unTrabajadorTemporal = (TrabajadorTemporal) personaAModificar
   int nuevoDiaFinContrato=ES.leeNº("Introduce el día de la fecha de fin del contra
                                          1, 31);
   int nuevoMesFinContrato=ES.leeNº("Introduce el mes de la fecha de fin del contra
                                          1, 12);
   int nuevoAñoFinContrato=ES.leeNº("Introduce el año de la fecha de fin del contra
                                          1900,2100);
   unTrabajadorTemporal.setFechaFinContrato(nuevoDiaFinContrato, nuevoMesFinContrat
                                            nuevoAñoFinContrato);
}
```

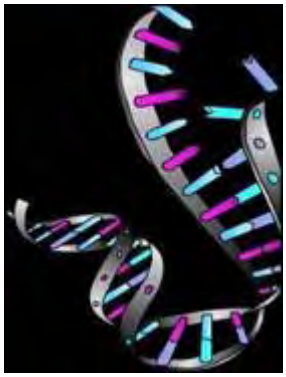
La ligadura dinámica

Anteriormente en este tema vimos que en una subclase se puede redefinir un método, de tal manera que se le da en ésta una implementación distinta a aquélla que tenía en la superclase. En el **ejemplo** de nuestra aplicación de gestión de trabajadores, esto nos permitía poder disponer de fórmulas distintas para calcular el sueldo de los trabajadores con contrato fijo con respecto a los trabajadores con contrato temporal. Para ello, lo único que había

que hacer era redefinir el método **getSueldo()** heredado de la superclase, proporcionándole al mismo implementaciones distintas en las subclases **TrabajadorTemporal** y **TrabajadorFijo**. Del mismo modo, también redefinimos el método **toString()** en las subclases para que en cada una de ellas el método muestre la información particular que define a la subclase.

Está claro que:

- Si **unTrabajador** es una variable de tipo **Trabajador**, que hace referencia a un objeto de la clase **Trabajador**, y se invoca sobre ella, por ejemplo, el método **toString()**, se ejecutará el código que para dicho método hay en la clase **Trabajador**.
- Por su parte, si **unTrabajadorFijo** es una variable de tipo **TrabajadorFijo**, que hace referencia a un objeto de la clase **TrabajadorFijo**, y se invoca sobre ella el método **toString()**, se ejecutará el código que para dicho método hay en la clase **TrabajadorFijo**.
- Por último, si **unTrabajadorTemporal** es una variable de tipo **TrabajadorTemporal**, que hace referencia a un objeto de la clase **TrabajadorTemporal**, y se invoca sobre ella el método **toString()**, se ejecutará el código que para dicho método hay en la clase **TrabajadorTemporal**.



Por otra parte, y también en este tema, hemos visto lo que es el **polimorfismo**, el cual permite que una variable cuyo tipo es el de la superclase contenga realmente una referencia a un objeto de una de las subclases.

Es entonces ahora el momento de hacernos la siguiente pregunta: ¿qué sucede cuando se dan simultáneamente ambas circunstancias: la redefinición de métodos y el polimorfismo? Es decir, ¿qué versión del método **toString()** se ejecutaría sobre la variable **unTrabajador**, de tipo **Trabajador**, si ésta hiciese referencia a un objeto de la clase **TrabajadorFijo**? La lógica nos dice que debería ejecutarse la versión del método que hay en la clase **TrabajadorFijo**, pues el objeto es realmente de ese tipo; y así sucede, gracias a lo que en el mundo de la orientación a objetos llamamos ligadura dinámica.

Llamamos ligadura dinámica a la capacidad de los lenguajes orientados a objetos de determinar, en tiempo de ejecución, qué versión o implementación de un método tiene que ser ejecutada cuando dicho método ha sido redefinido en las subclases y, además, se invoca sobre una variable polimórfica. Este mecanismo de la orientación a objetos recibe este nombre porque, bajo estas circunstancias, la invocación de un método se liga al código a ejecutar en dicha invocación de manera dinámica en el mismo momento de la invocación. Evidentemente, se ejecutará siempre la versión asociada a la clase del objeto al que hace referencia realmente la variable polimórfica.

Para ilustrar el funcionamiento de la ligadura dinámica vayámonos a nuestro programa de gestión de trabajadores. Observa la parte del código de la clase **GestionTrabajadores** que se encarga de localizar a aquellos trabajadores que tienen un sueldo superior a un sueldo dado; es decir, vayámonos al método **listadoMayorDeSueldo()**.

```
public static void listadoMayorDeSueldo(int sueldo){

   Nodo nodoAux=lista.primerNodo;

   int personasEnListado=0;

   System.out.println("LISTADO DE TRABAJADORES QUE GANAN MÁS DE " + sueldo + " EUROS

   System.out.println("=====================================================");
```

```
        while(nodoAux!=null){
            if (nodoAux.dato.getSueldo()>= sueldo){
                System.out.println(nodoAux + " sueldo: " + nodoAux.dato.getSueldo());
                personasEnListado++;
            }
            nodoAux=nodoAux.siguienteNodo;
        }
        System.out.println("SE HAN INCLUIDO " + personasEnListado +
                           " TRABAJADORES QUE GANAN MÁS DE " + sueldo +
                           " EUROS EN EL LISTADO");
    }
```

En dicho método se va recorriendo la lista de trabajadores e invocando, para cada trabajador, al método **`getSueldo()`**, para poder saber lo que cobra el trabajador y poder comprobar si esa cantidad es superior a la cantidad dada para, consecuentemente, saber si dicho trabajador tiene que ser seleccionado en nuestra búsqueda. En nuestra lista de trabajadores tendremos tanto "trabajadores fijos" como "trabajadores temporales", por lo que, en tiempo de ejecución, se irá ejecutando la versión del método asociada al objeto sobre el que se invoca.

Autoevaluación

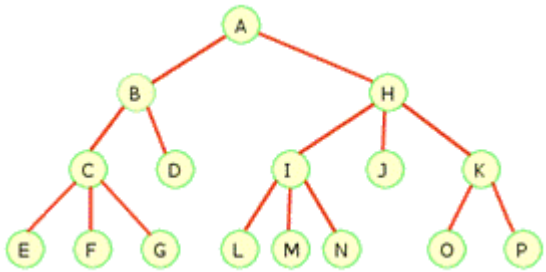
PARA SABER MÁS.
En este enlace encontrarás muy buena información sobre la ligadura dinámica en el apartado 5 del documento. Te recomendamos que leas todo el documento ya que sintetiza y explica muy bien los fundamentos de la programación orientada a objetos.
Polimorfismo y ligadura dinámica [Versión en caché]

Clases abstractas

*Este tema le está suponiendo a **Víctor** un verdadero dolor de cabeza, no hacen más que presentarle conceptos nuevos que les cuesta muchísimo comprender. Parece que el que ha decidido estos nombres era un verdadero desequilibrado. Ahora resulta que la clase Trabajador, según **José**, es una clase abstracta. ¿Y qué se supone que es una clase abstracta? **José** le explica que algo que viene utilizando en todos y cada uno de los ejemplos sobre los que está trabajando.*

***José** sólo venía a interesarse por los progresos de **Víctor** y la verdad es que lo ha encontrado algo alterado y un poco desesperado. **José** le tranquiliza y le explica que es posible que los nombres o términos que se utilizan sean complejos, pero la idea es bastante sencilla, sólo tiene que comprender la idea y todo será más fácil.*

Cuando pensamos en una clase, suponemos que los programas crearán objetos de ese tipo, pero esto no tiene por qué ser así siempre. ¿Cómo? ¿Estamos insinuando que puede haber clases de las que no se instancien objetos? ¿Qué utilidad puede tener una clase de este tipo?



Si lo pensamos detenidamente, esto no debería ser algo que nos sorprendiese, sobre todo una vez que conocemos la **importancia y beneficio de la identificación y uso de jerarquías de clases en el modelado y programación de nuestras aplicaciones**, como concepto clave que nos abre las puertas tanto de la herencia como del polimorfismo. Existen casos en los que **es conveniente declarar clases para las cuales el programador no pretende instanciar objetos, sino simplemente utilizarlas como superclases en alguna jerarquía de herencia**, de tal manera que obtiene programas más reutilizables y extensibles gracias al uso de la herencia y el polimorfismo.

No hace falta ir muy lejos para ilustrar esto de lo que hablamos. Pensemos en nuestra aplicación de gestión. En ella contamos con tres clases distintas para representar trabajadores: `Trabajador`, `TrabajadorFijo` y `TrabajadorTemporal`. Sin embargo, realmente sólo va a haber objetos de dos de ellas: `TrabajadorFijo` y `TrabajadorTemporal`, pues todos los trabajadores que manejemos caerán obligatoriamente en una de estas dos categorías. Entonces, si no se van a instanciar objetos de ella, ¿para qué sirve la clase `Trabajador`? Evidentemente, aunque no se vayan a tener objetos `Trabajador`, nosotros ya sabemos que la importancia de esta clase es muy grande, pues es la que nos permite tener y hacer uso de variables polimórficas, con todos los beneficios que hemos visto a lo largo del tema que esto nos reporta.

Pero vayamos aún más lejos. **Es posible que en una clase de este tipo, es decir, en una clase cuya única función es la de ser superclase en una jerarquía, haya métodos para los que no exista implementación**. Por ejemplo, si en nuestra jerarquía de trabajadores, la forma de calcular el sueldo de un trabajador depende del tipo concreto de trabajador que sea, "trabajador fijo" o "trabajador temporal", ¿para qué necesito darle implementación al método `getSueldo()` en la clase `Trabajador`? Como este método va a ser obligatoriamente redefinido en las subclases no tiene sentido darle una implementación en la superclase. Además, no existe ninguna fórmula general común a todos los trabajadores para calcular su sueldo, por lo que no es algo que se pueda establecer en la superclase.

Métodos que no proporcionan ninguna implementación

Un método que no proporciona ninguna implementación recibe el nombre de método abstracto y en su definición deberá aparecer la palabra clave `abstract` después del modificador de acceso del atributo y antes del tipo del valor de retorno del mismo:

```
[private | protected | public] abstract tipo_retorno nombre_método(parámetros) excep
```

Así, por ejemplo, observa cómo en la clase Trabajador definimos el método abstracto `getSueldo()`, sin implementación alguna, de la siguiente manera:

```
public abstract double getSueldo();
```

Una clase que contiene algún método que sea abstracto recibe, a su vez, **el nombre de clase abstracta y en su definición deberá aparecer la palabra clave `abstract` después del modificador**

de acceso de la clase y antes de la palabra clave `class`:

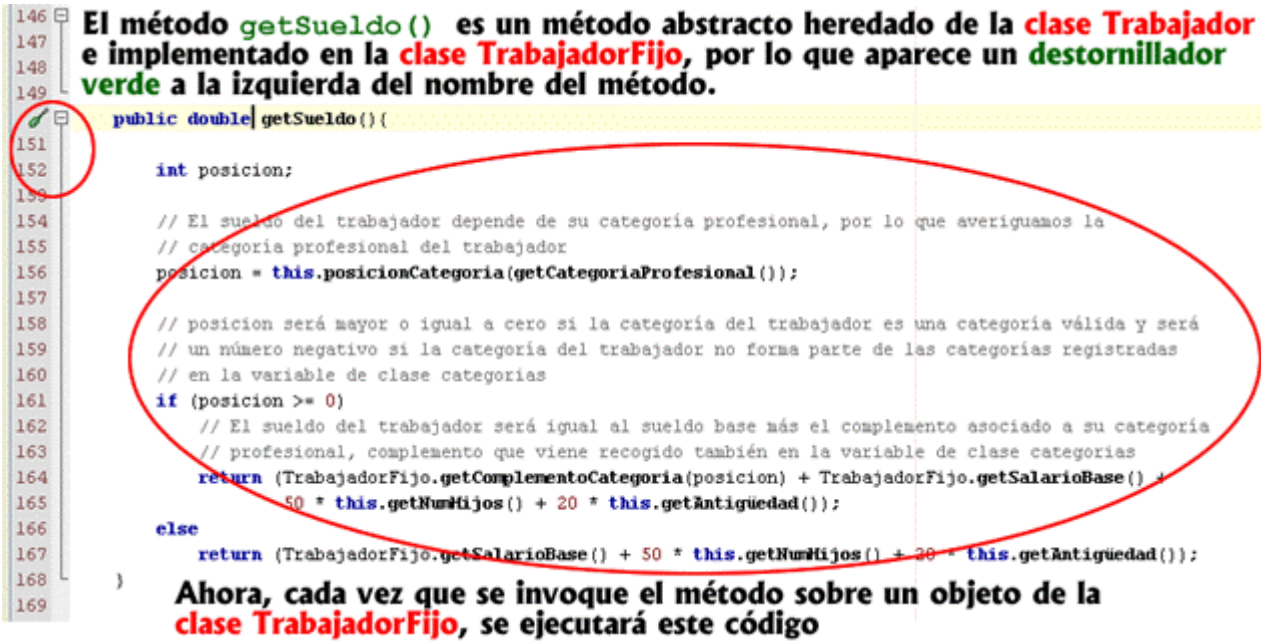
```
[public] abstract class Nombre_de_la_clase [extends superclase] implements lista_de_
```

Así, por ejemplo, la definición de la clase Trabajador se declararía como sigue:

```
public abstract class Trabajador implements Serializable, ReceptorDeMensajes {

// código de la clase Trabajador

}
```

Cuando trabajamos con clases abstractas debemos tener en cuenta lo siguiente:

- **Una clase que contenga métodos abstractos debe estar definida obligatoriamente como clase abstracta**. No hacerlo producirá un error en tiempo de compilación.
- Las clases abstractas son demasiado genéricas como para crear objetos reales, sólo especifican lo que sus subclases tienen en común. Es decir, **las clases abstractas se utilizan sólo como superclases en las jerarquías de herencia** y no se pueden utilizar para instanciar objetos, algo lógico si tenemos en cuenta que son clases incompletas al carecer de implementación, algunos de sus métodos. Por lo tanto, **si se intenta instanciar un objeto de una clase abstracta se producirá un error en tiempo de compilación**.
- Hemos dicho que el propósito de una clase abstracta es proporcionar una superclase apropiada a partir de la cual puedan heredar otras clases. **Dichas subclases deberán dar implementación a los métodos abstractos y recibirán el nombre de clases concretas.** Evidentemente, de estas subclases sí que se podrán instanciar objetos, al proporcionar éstas los aspectos específicos que hacen que sea razonable el crear instancias de objetos. Esto sucede en nuestra aplicación de gestión con las subclases concretas **`TrabajadorFijo`** y **`TrabajadorTemporal`**, las cuales implementan el método abstracto **`getSueldo()`** heredado de la superclase abstracta **`Trabajador`**.




```
146  public double getSueldo(){
147
148      int posicion;
149
150      // El sueldo del trabajador depende de su categoría profesional, por lo que averiguamos la
151      // categoría profesional del trabajador
152      posicion = this.posicionCategoria(getCategoriaProfesional());
153
154      // posicion será mayor o igual a cero si la categoría del trabajador es una categoría válida y será
155      // un número negativo si la categoría del trabajador no forma parte de las categorías registradas
156      // en la variable de clase categorias
157      if (posicion >= 0)
158          // El sueldo del trabajador será igual al sueldo base más el complemento asociado a su categoría
159          // profesional, complemento que viene recogido también en la variable de clase categorias
160          return (TrabajadorFijo.getComplementoCategoria(posicion) + TrabajadorFijo.getSalarioBase() +
161                  50 * this.getNumHijos() + 20 * this.getAntigüedad());
162      else
163          return (TrabajadorFijo.getSalarioBase() + 50 * this.getNumHijos() + 20 * this.getAntigüedad());
164  }
165
166
167
168
169
```

- Realmente una jerarquía de herencia no necesita contener clases abstractas. Sin embargo, comúnmente se utilizan jerarquías de clases encabezadas por superclases abstractas para reducir las dependencias de código cliente en las subclases específicas. Además, en ocasiones las clases abstractas constituyen varios niveles de la jerarquía. **Es decir, es posible que una subclase de una clase abstracta sea también abstracta**, retrasándose un nivel más en la jerarquía la implementación de los métodos abstractos. Por su parte, **las clases terminales de una jerarquía de herencia serán obligatoriamente clases concretas**.
- **Los métodos constructores** no se heredan, por lo que **no pueden declararse como abstract**.

Autoevaluación

Interfaces

La programación en Java está directamente relacionada con la Programación Orientada a Objetos, y esto no debe ser un inconveniente, sino una gran ventaja que le va a proporcionar al programador una serie potentes herramientas y una forma de trabajo que le va a permitir la reutilización de código con todas las ventajas que eso conlleva. Pero ***Víctor*** *no está de acuerdo, sostiene que esto es una complicación y que siguiendo esta compleja filosofía de trabajo tarda mucho más en desarrollar y además no tiene efectos en el resultado final, ya que casi todo es código oculto que no produce ningún efecto sobre el uso que el usuario hace de la aplicación.*

Carmen *le explica razonadamente que eso no es correcto. En primer lugar demuestra que no es más costoso el diseño de la aplicación siguiendo el paradigma de la Programación Orientada a Objetos. Después le convence sobre el uso de clases abstractas en la creación de objetos y finalmente le explica las ventajas que puede tener el uso de interfaces. También le comenta que de hecho las interfaces se usan muchísimo en java, aunque él mismo será consciente de toda su potencia cuando comience con la programación de aplicaciones con interfaz gráfica (ventanas). En ese momento Víctor verá claramente que no sólo son muy útiles, si no que son imprescindibles.*

De todo lo aprendido, si tuviésemos que quedarnos con una única frase que resumiese la filosofía de este paradigma de programación, sería la siguiente: **la programación orientada a objetos es una programación orientada a la interfaz que pretende ocultar la implementación**; es decir, es una filosofía de programación que se centra en el qué se hace en vez de en el cómo se hacen las cosas. Además, cuando se desea construir una especialización o una extensión de una clase existente, no hace falta programarla de nuevo desde cero, sino que **podemos heredar de la clase existente, lo cual implica que la nueva clase asume y hace propia la interfaz de la clase de la cual deriva**.

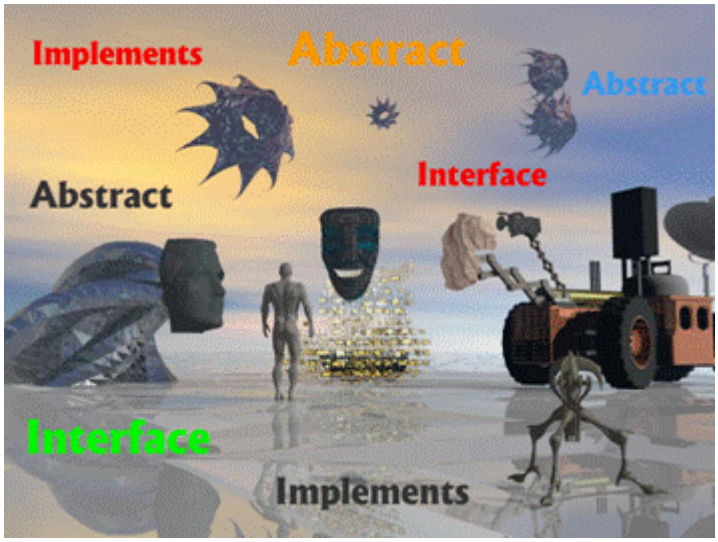
Un **ejemplo** muy ilustrativo de la puesta en práctica de esta filosofía de programación orientada a la interfaz la hemos visto en este tema cuando hablamos de los métodos abstractos que se definen en una superclase: métodos de los cuales sólo se da su declaración o interfaz, pero que carecen de implementación. La razón es que ésta poco importa, **lo único que se pretende es establecer un comportamiento común que deben soportar todos los objetos que hereden de ella (interfaz), y ya se encargarán sus subclases de preocuparse del cómo se lleva a cabo dicho comportamiento (implementación)**.

¿Se puede llevar la idea de las clases abstractas a su máxima expresión? Es decir, ¿podría tener una clase abstracta donde todos sus métodos fuesen abstractos? Perfectamente, y en ese caso lo que estaría definiendo la clase no sería más que una interfaz completa para las clases concretas que hereden de ella. Sería como definir la estructura básica de las clases de la jerarquía pero sin entrar en detalles.

Para cerrar el tema anterior introdujimos el concepto de Interface de Java como otro mecanismo distinto para llevar a cabo esta última idea: tener una clase totalmente abstracta. **Mediante la construcción de una interface, el programador pretende especificar qué comportamiento caracteriza a una colección de objetos e, igualmente, especificar qué comportamiento deben reunir los objetos que quieran entrar dentro de esa categoría o colección**.

Imagina cualquier **dispositivo** en el que puedas almacenar información, desde una libreta hasta un CD-ROM. ¿Qué tienen en común todos ellos? En principio podríamos decir que más bien poco ¿verdad? Sin embargo, si no nos quedamos simplemente en la superficie del aspecto y profundizamos un poco en lo que es la utilidad que tienen, podremos vislumbrar que todos los artefactos o dispositivos que hayas

podido pensar, desde el más moderno al más arcaico, tienen en común lo siguiente:



- Permiten **escribir** información en el dispositivo,
- **almacenan** internamente, de algún modo, dicha información
- y permiten **leer** o recuperar la información almacenada.

Además, cualquier artefacto nuevo que inventemos y que cumpla dichas características, podremos considerarlo también un dispositivo de almacenamiento. Es cierto que cada uno de estos artefactos almacena la información de un modo distinto y tiene mecanismos distintos para leer y escribir la información; pero también es cierto que todos ellos ofrecen, de un modo u otro, la misma funcionalidad. Podríamos entonces considerar estas características que definen la funcionalidad de "ser dispositivo de almacenamiento" como una **interface**, la cual **especifica el comportamiento que debe tener todo artefacto que quiera ser considerado "dispositivo de almacenamiento"**. Por su parte, cualquier artefacto, sea éste del tipo que sea, lo único que tiene que hacer para ser considerado un "dispositivo de almacenamiento" es ofrecer la funcionalidad o servicios especificados en la interface correspondiente, especificando en su código cómo va a llevar a cabo dichos servicios; es decir, proporcionando una implementación para la interface.

Una interface en Java es, básicamente, **una declaración de métodos sin implementación, aunque también se pueden declarar constantes,que definen el comportamiento que deben soportar los objetos que quieran pertenecer al "club" de esa interface**. Las clases que quieran pertenecer a dicho "club" deberán entonces dar una implementación a todos y cada uno de los métodos que definen la interface, acción que recibe el nombre de **implementar la interface**.

¿Es entonces lo mismo una interface que una clase abstracta? No, no es lo mismo. A simple vista podemos ver la siguiente diferencia: **una interface es simplemente una lista de métodos no implementados, mientras que una clase abstracta puede incluir métodos implementados y no implementados y, además, variables de instancia y de clase**. Pero la diferencia es un poco más profunda:

- **Una clase abstracta define una interfaz que sólo puede ser utilizada a través de la herencia**. Por lo tanto, sólo las clases que compartan la misma jerarquía de clases tendrán en común la interfaz. Utilizar este mecanismo implica establecer vínculos fuertes entre las clases, como son los vínculos de herencia, y solamente deberá aplicarse cuando realmente existan conceptualmente esas relaciones entre las clases: **no debemos forzar una relación "es un" si conceptualmente ésta no existe**. Recordemos que establecer una jerarquía de herencia supone definir una estructura común a varias clases, entendiendo por estructura tanto comportamiento o métodos como atributos o variables de instancia y clase.
- Sin embargo, **una interface puede ser implementada por cualquier clase**. Por lo tanto, clases que no tengan ninguna relación entre sí, entiéndase ninguna relación de herencia, pueden compartir una interfaz; es decir, pueden compartir un mismo comportamiento que las hace tener algo en común. **Deberemos utilizar interfaces cuando queramos definir un comportamiento, y sólo comportamiento, que pueda ser común a clases entre las que no haya relaciones de**

herencia. Debemos tener siempre presente que lo único que puede definir una interface son métodos y constantes, una interface no puede definir variables de ningún tipo. Si lo que necesitamos definir es una estructura común a varias clases, incluyendo comportamiento y atributos, entonces deberemos optar siempre por definir una jerarquía de herencia con clases abstractas.

Interfaces en Java

Volvamos a nuestra aplicación de gestión de trabajadores. En ella hemos establecido una jerarquía de herencia, de tal manera que en la clase abstracta **`Trabajador`** se define una estructura, atributos y comportamiento, común a todos los trabajadores de nuestra empresa. De manera similar, si ampliamos nuestra aplicación para que gestione, no sólo los trabajadores, sino también los clientes y proveedores de la empresa, podríamos elaborar jerarquías similares que partiesen de clases abstractas como **`Cliente`** y **`Proveedor`**.

Así mismo, imagina que en nuestra **empresa** los flujos de comunicación con nuestros trabajadores se hacen mayoritariamente a través del correo electrónico. De hecho, a cada trabajador se le da una dirección en el servidor de la empresa. De igual manera, la empresa mantiene flujos de comunicación con los clientes, por ejemplo, para mandarles ofertas de nuevos productos, y con los proveedores, para solicitar pedidos. Es entonces necesario que nuestra aplicación proporcione algún mecanismo para poder mandar un mensaje de correo a un trabajador, a un cliente y a un proveedor:

- Podríamos pensar en añadir un método, **`enviarMensaje()`**, a cada una de las clases: Trabajador, Cliente y Proveedor; que implementase el mecanismo para enviar un correo electrónico al trabajador, cliente o proveedor representado por el objeto. Pero esto sería incorrecto, pues estaríamos repitiendo el mismo código en las tres clases, que tienen un comportamiento común.
- Podríamos rectificar la jerarquía de herencia introduciendo una superclase llamada **`ReceptorDeMensajes`**, que definiría el comportamiento que deben de tener todos los elementos que quieran ser capaces de recibir un mensaje de correo electrónico, y de la que heredarían tanto la clase Trabajador como las clases Cliente y Proveedor. Quizá sea forzar demasiado la herencia, aunque sería una solución correcta. Sin embargo, si analizamos la clase resultante, vemos que sería una clase muy liviana, pues lo único que haría es definir un método, **`enviarMensaje()`**. Además, lo que define es más un comportamiento que otra cosa. Por lo tanto, puede que sea mejor emplear la siguiente de las soluciones.
- Como tercera solución, y la más correcta, podríamos definir una interface llamada **`ReceptorDeMensajes`**, que definiría el comportamiento común que deben de tener los elementos que quieran ser capaces de recibir un mensaje de correo electrónico. Dicha interface debería entonces ser implementada por las clases **`Trabajador`**, **`Cliente`** y **`Proveedor`**, donde cada una de ellas se encargaría de establecer cómo se manda el correo a un trabajador, un cliente y a un proveedor, respectivamente.

Veamos cómo llevar a cabo la tercera de las soluciones en nuestro programa.

Para definir una interface en Java haremos uso de la siguiente sintaxis:

```
Modificador_acceso interface Nombre_Interface {

// Declaración de los métodos que constituyen la interface y definen el comportamiento com

// a las clases que vayan a implementarla

}
```

Como vemos, **las interfaces también llevan modificador de acceso**, que puede ser **`public`** o puede

no aparecer nada, siendo entonces, por defecto, considerada la interface como de ámbito **`package`**. Además, como el resto de las clases públicas de Java, **tendrán asociado un archivo propio con extensión `.java` y que tendrá el mismo nombre que la interface**.

Dentro de la interface hay que declarar los **métodos** que la constituyen y que definirán el comportamiento representado por la misma. Para definirlos haremos uso de la misma sintaxis que usamos para cualquier otro método, con las siguientes particularidades:

- A los métodos no se les da implementación, sólo se define su interfaz. Por lo tanto, son lo que en Java llamamos métodos **`abstract`**, aunque no es necesario ponerlo expresamente.
- Todos los métodos serán públicos, por lo que realmente no hay que declararlos expresamente como tales, aunque sí conveniente.

Por lo tanto, en nuestra aplicación de gestión de trabajadores, la interface **`ReceptorDeMensajes`**, que tiene un método llamado **`enviarMensaje()`** se definiría así:

```
public interface ReceptorDeMensajes {

    public void enviarMensaje(String subject, String cuerpo);

}
```

Pero, además de los métodos, **una interface también puede definir constantes** de la siguiente forma:

```
public static final tipoConstante nombreConstante = valorConstante;
```

Observa que debe ir obligatoriamente definida con los siguientes modificadores: **`public static final`** y deberá estar inicializada con un valor, pues se trata de una constante.

Por su parte, cuando una clase quiere soportar el comportamiento definido por una interface, debe implementarla. Para ello, deberá indicarlo así en su cabecera de declaración de clase, haciendo uso de la palabra clave implements, seguida del nombre de la lista de interfaces que implementa:

```
[public][final | abstract] class Nombre_de_la_clase
                                                [extends superclase][implements lista_
```

No debemos olvidar nunca lo siguiente:

- **Una clase puede implementar varias interfaces**, para ello sólo tiene que enumerarlas separadas por comas detrás de la palabra clave **`implements`**.
- **Una clase tan sólo puede extender o heredar de una superclase**. Es decir, detrás de la palabra clave **`extends`** sólo puede aparecer el nombre de una clase.
- **Una clase que implementa una interface debe de proporcionar implementación para todos y cada uno de los métodos definidos en la misma**, pues a todos los efectos, esos métodos forman parte de la interfaz pública de la propia clase. Si para algún método no quisiese proporcionar implementación, entonces la clase tendría que ser declarada como abstracta, al tener métodos sin implementación. No hacerlo así produciría un error en tiempo de compilación al ser un error de sintaxis.
- **Las clases que implementan una interfaz que tiene definidas constantes pueden usarlas en cualquier parte del código de la clase, simplemente con su nombre.** Inclusive una clase puede utilizar una constante de este tipo importando la interfaz y después haciendo referencia a la constante como **`NombreInterface.NombreConstante`**.

Así, en nuestro programa, la clase Trabajador tendría la siguiente cabecera:

```
public abstract class Trabajador implements Serializable, ReceptorDeMensajes {

}
```

Por su parte, dentro de la clase debería implementarse el método de la interface **enviarMensaje()**, que en nuestro caso simplemente imprime un mensaje por pantalla:

```
public void enviarMensaje(String subject, String cuerpo){
    // Aquí iría el código para enviar el correo electrónico, que no veremos.
    // Nosotros simplemente mostramos un mensaje por pantalla
        System.out.println("Correo electrónico enviado a la dirección " + this.emai
}
```

Polimorfismo con interfaces

Una de las principales ventajas de **la definición de jerarquías de herencia** era que esto **me abría las puertas al polimorfismo**, es decir, a la posibilidad de poder tratar objetos de las distintas subclases de una superclase igual que si fuesen objetos de dicha superclase común. Esto me permitía, básicamente, utilizar una variable del tipo de la superclase para tratar de igual forma a objetos distintos, con la única condición de que éstos fuesen objetos de una de sus subclases.

Las interfaces, por su parte, me permiten diseñar un comportamiento común entre objetos que no tengan una jerarquía en común, y hemos visto que, en muchas ocasiones, su uso es la solución a utilizar. ¿Quiere esto decir que con el uso de interfaces pierdo la posibilidad de hacer uso del **polimorfismo**? Ésta es una cuestión muy importante, pues si esto fuese así, el uso de interfaces perdería casi todo su interés. Afortunadamente, también **se puede hacer un uso polimórfico en base a una interface, de tal manera que puedo tratar de la misma forma objetos distintos con la única condición de que todos ellos implementen la misma interface**. Esto, como puedes imaginarte, otorga a las interfaces una gran potencia, al proporcionarles todos los beneficios asociados al polimorfismo.

Para poder hacer uso del polimorfismo con interfaces deben cumplirse básicamente tres condiciones:

- **El tipo de la variable polimórfica debe ser la propia interface**. Es decir, si la interface tiene el nombre de **`InterfaceA`**, la declaración de la variable se hará de la siguiente manera:
  `InterfaceA variable;`
- **Todos los objetos a los que haga referencia la variable polimórfica deben implementar obligatoriamente la interface sobre la que está definida la variable**. Es decir, si la variable ha sido definida sobre la interface **`InterfaceA`**, entonces todos los objetos a los que haga referencia la variable deben implementar dicha **`InterfaceA`**.
- **Sobre una variable polimórfica sólo se podrán invocar métodos que vengan definidos en la propia interface y ningún otro**. Esto es lógico si tenemos en cuenta que es lo único que tenemos garantizado que van a tener en común los distintos objetos a los que puede hacer referencia dicha variable.

Vayamos a nuestro programa de gestión para ver el uso que en el mismo se ha hecho del polimorfismo con interfaces. Observa el código del método **`enviarMensajeTodosTrabajadores()`** de la clase **`GestionTrabajadores`**. La misión de este método es mandar un mensaje de correo electrónico a todos y cada uno de los trabajadores de la empresa, para lo cual deberá ser invocado el método **`enviarMensaje()`** sobre todos y cada uno de los objetos Trabajador. Dicho método, recordemos, que es un método que viene dado por la interface **`ReceptorDeMensajes`** que la clase **`Trabajador`** implementa.

Primeramente, en el método se define una variable local del tipo de la interface:

```
ReceptorDeMensajes receptor;
```

Posteriormente, cada objeto de la lista de trabajadores lo vamos asignando a la variable **`receptor`**, lo cual es posible pues la clase Trabajador implementa la interfaz **`ReceptorDeMensajes`**, y vamos invocando sobre los mismos el método **`enviarMensaje()`**:

```
Nodo nodoAux=lista.primerNodo;

while(nodoAux != null){

        receptor=nodoAux.dato;

        receptor.enviarMensaje(asunto, cuerpo);

        nodoAux=nodoAux.siguienteNodo;

}
```

Realmente, no hubiese hecho falta definir la variable receptor, pues todos los objetos con los que vamos a trabajar son del tipo Trabajador o de alguna de sus subclases, por lo que hubiésemos podido utilizar simplemente una variable polimórfica de tipo Trabajador. Hemos utilizado una variable del tipo de la interface para ilustrar su uso. No obstante, si quisiésemos que el método, en vez de mandarle un correo a los trabajadores, lo enviase a un conjunto de personas seleccionadas, ya sean éstas trabajadores, clientes o proveedores, todas ellas implementando la interface en cuestión, entonces sí que es obligatorio utilizar una variable del tipo **`ReceptorDeMensajes`**, tal y como hemos hecho en nuestro código.

Autoevaluación

Simulación de la herencia múltiple mediante interfaces.

Hemos visto que una clase puede heredar de otra, absorbiendo y haciendo propias su estructura y sus características: atributos y métodos; incluso asumiendo las implementaciones hechas en la superclase, si bien es cierto que también sabemos que dicha implementación puede ser sustituida en la subclase. ¿Es posible que una clase necesite heredar y herede de más de una superclase? Ya sabemos que esto es perfectamente posible y que este hecho se conoce con el nombre de herencia múltiple.

Sabemos que **Java** soporta la herencia simple, pero **no soporta la herencia múltiple**, lo cual quiere decir que una clase no puede heredar directamente de más de una superclase y, por lo tanto, en su cabecera no puede aparecer nada más que un nombre después de la palabra clave **`extends`**.

Por otra parte, también sabemos que **una clase puede implementar más de una interface**, donde cada una de ellas define un comportamiento a través de un conjunto de métodos no implementados. Dadas las similitudes que tiene una interface con una clase, **podemos**, en cierto modo, **simular la herencia múltiple haciendo uso de interfaces**. Es cierto que no será exactamente lo mismo que si pudiésemos disponer de herencia múltiple, pues una interface no define ningún tipo de estructura ni ninguna implementación, sólo la especificación de un comportamiento, pero el resultado se aproxima bastante.

Como siempre que se hace uso de la herencia múltiple, aunque ésta sea simulada a través de interfaces, cabe la posibilidad que se produzcan dos problemas:

- **Colisión de nombres**.
- **Herencia repetida**.

Veamos en qué consiste cada uno de ellos.

Imagina que **una clase implementa dos interfaces**. Imagina, asimismo, que ambas interfaces tienen un método que tiene el mismo nombre en ambas. ¿Qué sucede con estos métodos de mismo nombre en la clase que quiere implementar ambas interfaces? Es evidente que en la clase que implementa las interfaces se va a producir lo que se conoce como **colisión de nombres, pues va a haber dos métodos distintos, cada uno proveniente de una de las interfaces, pero que se llaman igual**. Ante una situación de este tipo hay tres posibilidades:

- **Si ambos métodos tienen el mismo nombre pero diferentes parámetros**, entonces se produce una sobrecarga de métodos, de tal manera que se invocará a uno o a otro dependiendo de los argumentos que se utilicen en la llamada al método.
- **Si ambos métodos tienen el mismo nombre y los mismos parámetros**, pero se diferencian en el tipo del valor devuelto, se producirá un error de compilación, al igual que sucedía con la sobrecarga común.
- **Si ambos métodos coinciden exactamente en su declaración**: nombre, parámetros y tipo del valor devuelto; entonces se eliminará uno de los métodos, y sólo se podrá implementar uno de ellos.

Por su parte, **si la colisión de nombres se produjese entre constantes definidas en las interfaces en conflicto**, desde la clase que las implementa se podrá acceder a todas ellas simplemente anteponiendo al nombre de la constante el nombre de la interface en la cual está definida.

Al igual que una clase puede heredar de otra, **una interface en Java** también **puede heredar de otra interface**, absorbiendo todas las definiciones de métodos y constantes de su interface superclase.

Imagina que tenemos definida la interface **`InterfaceA`**:

```
public interface InterfaceA {

// Declaración de los métodos y constantes que constituyen la interface

}
```

Imagina también que queremos que las interfaces **`InterfaceB`** e **`InterfaceC`** la extiendan tras heredar de ella. Para conseguirlo haremos uso de la palabra clave **`extends`**, exactamente del mismo modo que haríamos si se tratase de herencia entre clases:

```
public interface InterfaceB extends InterfaceA {

// Declaración de los métodos y constantes que constituyen la interface

}

public interface InterfaceC extends InterfaceA {

// Declaración de los métodos y constantes que constituyen la interface

}
```

Imagina ahora que queremos definir una clase **`ClaseD`** que implemente las interfaces **`InterfaceB`** e **`InterfaceC`**. ¿Ves algo raro? Si observas, ambas interfaces a implementar proceden de la misma interface madre, por lo que en **`ClaseD`** nos vamos a encontrar con métodos y constantes duplicados, todos aquellos que proceden de la interface **`InterfaceA`**. **Esto es lo que se conoce como el problema de la herencia repetida** y las colisiones de nombres que se producen se resuelven siguiendo los criterios que describimos anteriormente. En este caso, como los métodos duplicados coinciden exactamente en su declaración: nombre, parámetros y tipo del valor devuelto; entonces de cada pareja de métodos repetidos se eliminará uno de ellos y sólo se podrá implementar el otro.

PARA SABER MÁS.
En este enlace encontrarás más documentación sobre interfaces donde se tocan los apartados vistos hasta ahora. Básicamente trata conceptos como la declaración e

implementación de Interfaces, o las colisiones con herencia múltiple.

Interfaces **[Versión en caché]**

Interesante documento que sintetiza muy bien el lenguaje de programación Java. Todos los conceptos vienen muy bien ilustrados en ejemplos con código muy claro y preciso. ¡Muy interesante!

Programación Java **[Versión en caché]**

Interesante presentación sobre la programación orientada a objetos en Java, en la que puedes encontrar una explicación detallada de los conceptos principales de este paradigma de la programación con Java.

Programación Orientada a Objetos en Java **[Versión en caché]**