7. Utilización avanzada de clases.

Autor	Ⓧ Xerach Casanova
Clase	Programación
Fecha	@Mar 6, 2021 1:36 PM

1. Relaciones entre clases

A la hora de diseñar un conjunto de clases para modelar el conjunto de información a automatizar, es importante establecer apropiadamente las clases relacionales que existen entre ellas.

Una clase puede ser una especialización de otra, o bien una generalización. Una clase puede contener objetos en el interior de otra, o utiliza otra clase. Se pueden distinguir distintos tipos de relaciones entre clases:

- **Clientela.** Cuando una clase utiliza objetos de otra, por ejemplo al pasarlos como parámetros a través de un método. Es la relación fundamental y más habitual entre clases. Por ejemplo, usando objetos de tipo String en una clase, esta clase será cliente de la clase String.
- **Composición.** Cuando alguno de los atributos de una clase es un objeto de otra clase. Por ejemplo, si escribes una clase donde uno de sus atributos es de tipo String, tu clase está compuesta por un objeto de tipo String.
- **Anidamiento.** Cuando se definen clases en el interior de otra clase. Es la menos habitual, implica declarar clases dentro de otras clases, en algunos casos puede resultar útil para tener un nivel más de encapsulamiento y ocultación.
- **Herencia.** Cuando una clase comparte determinadas características de otra (clase base), añadiéndole funcionalidad específica (especialización).

Podría decirse que la composición y la anidación son casos particulares de clientela, ya que en realidad en todos los casos una clase está haciendo uso de otra, al contener atributos que son objetos de otra clase, al definir clases dentro de otras clases, al utilizar objetos en paso de parámetros, al declarar variables locales utilizando otras clases, etc...

1.1. Composición

La composición se da cuando una clase contiene algún atributo que es una referencia a un objeto de otra clase. Es decir, el objeto de la clase A contiene uno o varios objetos de la clase B.

Por ejemplo, si describes una entidad País compuesta por una serie de atributos, entre los cuales se encuentra una lista de comunidades autónomas, podrías decir que los objetos de la clase País contienen varios objetos de la clase ComunidadAutonoma.

La composición se puede encadenar todas las veces que sea necesario hasta llegar a objetos básicos del lenguaje o hasta tipos primitivos, que ya no contendrán objetos en su interior. Es una manera eficiente y sencilla de gestionar la reutilización de todo el código ya escrito.

La forma más sencilla de plantearte si una relación existemte entre dos clases A y B es de composición es usando la expresión "tiene un" entre las dos clases.

- Un coche tiene un motor y tiene cuatro ruedas.
- Una persona tiene una cuenta bancaria asociada para ingresar nómina.

1.2. Herencia

El concepto de herencia es algo básico pero potente. Cuando se define una nueva clase y ya existen clases que de alguna manera implementan parte de la funcionalidad que necesita, es posible crear una nueva clase derivada de la que ya tienes. Esto posibilita la reutilización de todos los atributos y métodos de la clase padre o superclase, sin necesidad de tener que escribirlos de nuevo.

Una subclase hereda todos los miembros de la clase padre: atributos, métodos y clases internas.

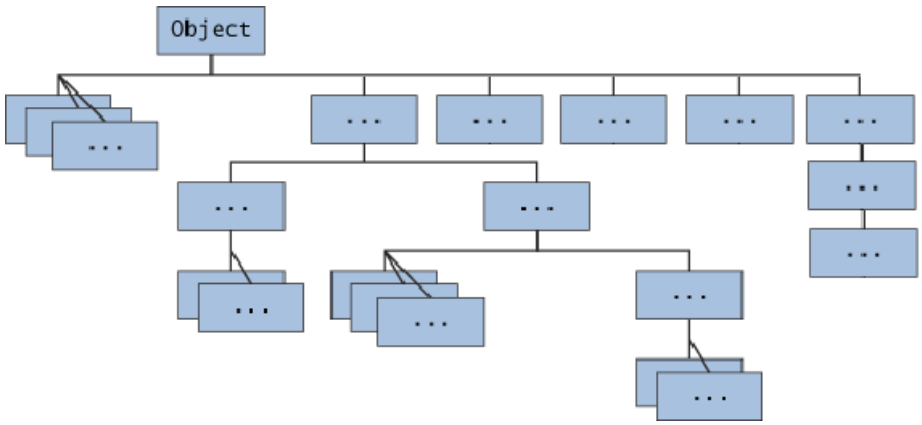
Los constructores no se heredan, pero se pueden invocar desde la subclase.

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

La expresión idiomática para plantear si existe herencia entre dos clases A y B es "es un". La clase A "es un" tipo específico de la clase B (especialización), o visto de otro modo: la clase B es un caso general de la clase A.

Java implementa la herencia mediante la utilización de la palabra reservada extends. La clase Object, dentro del paquete java.lang, define e implementa el comportamiento común de todas las clases, incluyendo las propias. Cualquier clase deriva en última instancia de la clase Object.

Todas las clases tienen una clase padre, que a su vez posee una superclase y así llegar hasta la clase Object.



1.3. ¿Herencia o composición?

Cuando escribimos nuestras propias clases se debe tener claro cuando utilizar la composición y cuando la herencia.

En el caso de la composición, una clase puede estar formada por objetos de otras clases, pero no necesariamente tienen que compartir características. Estos objetos incluidos no son más que atributos miembros de la clase que estamos definiendo.

En el caso de la herencia, una clase cumple todas las características de otra y además las suyas propias (especialización, particularización, extensión o restricción). O lo que es lo mismo, una clase base es una generalización de las clases derivadas.

Ejemplo:

Por ejemplo, imagina que dispones de una clase Punto (ya la has utilizado en otras ocasiones) y decides definir una nueva clase llamada Círculo. Dado que un punto tiene como atributos sus coordenadas en plano (x1, y1), decides que es buena idea aprovechar esa información e incorporarla en la clase Circulo que estás escribiendo. Para ello utilizas la herencia, de manera que al derivar la clase Círculo de la clase Punto, tendrás disponibles los atributos x1 e y1. Ahora solo

faltaría añadirle algunos atributos y métodos más como por ejemplo el radio del círculo, el cálculo de su área y su perímetro, etc.

En principio parece que la idea pueda funcionar pero es posible que más adelante, si continúas construyendo una jerarquía de clases, observes que puedas llegar a conclusiones incongruentes al suponer que un círculo es una especialización de un punto (un tipo de punto). ¿Todas aquellas figuras que contengan uno o varios puntos deberían ser tipos de punto? ¿Y si tienes varios puntos? ¿Cómo accedes a ellos? ¿Un rectángulo también tiene sentido que herede de un punto? No parece muy buena idea. En este caso habría resultado mejor establecer una relación de composición. Analízalo detenidamente: ¿cuál de estas dos situaciones te suena mejor?

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

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

Composición

2.1. Sintaxis de la composición

En la composición no es necesaria ninguna sintaxis especial. cada uno de esos objetos es un atributo:

```
class Rectangulo {

     private Punto vertice1;

     private Punto vertice2;


}
```

Los métodos de esta clase deben tener en cuenta que ya no hay cuatro atributos de tipo double, sino dos atributos de tipo punto, cada uno de los cuales contiene en su interior dos atributos de tipo double.

Ejercicio resuelto:

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

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

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

```
public double calcularSuperficie () {

    double area, base, altura;   // Variables locales



    base= vertice2.obtenerX () - vertice1.obtenerX ();  // Antes era x2 - x1

    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1

    area= base * altura;

    return area;

}

public double CalcularPerimetro () {

    double perimetro, base, altura;   // Variables locales



    base= vertice2.obtenerX () - vertice1.obtenerX ();  // Antes era x2 - x1

    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1

    perimetro= 2*base + 2*altura;

    return perimetro;

}
```

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

Cuando se escriben clases que contienen objetos de otras clases, se debe tener precaución con aquellos métodos que devuelven información de atributos de la clase (métodos get).

Los atributos suelen ser privados o protegidos, para ocultarlos a los posibles clientes de la clase. Para que otros objetos puedan acceder a la información contenida en los atributos, deben hacerlo a través de los métodos que sirvan de interfaz (métodos getter y setter).

Al igual que con los atributos primitivos, podemos devolver un objeto completo, pero **se debe tener en cuenta que si en un método de la clase devuelves un objeto que es atributo, se está ofreciendo directamente una referencia a un objeto atributo que probablemente se ha definido como privado**. De esta forma estás volviendo a hacer público un atributo que inicialmente era privado.

Para evitar estas situaciones, se opta por diversas alternativas, procurando evitar la devolución directa de un atributo que sea un objeto.

- Devolver siempre tipos primitivos.
- Dado que esto siempre no es posible, otra posibilidad es crear un nuevo objeto que sea copia del atributo a devolver y utilizar ese objeto como valor de retorno.

También se debe tener en cuenta que es posible que en algunos casos sí se necesite realmente la referencia al atributo original (algo habitual en atributos estáticos). En tales casos no hay problema

en devolver directamente el atributo para que el código llamante (cliente) haga uso de él.

Ejercicio resuelto:

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

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

```
public Punto obtenerVertice1 ()
{

     return vertice1;

}

public Punto obtenerVertice2 ()
{

     return vertice2;

}
```

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

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

Aquí tienes algunas posibilidades:

```
public Punto obtenerVertice1 ()   // Creación de un nuevo punto extrayendo sus atributos
{

     double x, y;

     Punto p;

     x= this.vertice1.obtenerX();

     y= this.vertice1.obtenerY();

     p= new Punto (x,y);

     return p;

}

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

     Punto p;

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

     return p;

}
```

2.3. Uso de la composición (II). Llamadas a constructores

A la hora de escribir clases que contienen objetos de otras clases como atributos, es su comportamiento cuando se instancian. Durante el proceso de creación del objeto de la clase contenedora, también se debe tener en cuenta la creación de aquellos objetos contenidos.

El constructor de la clase contenedora debe invocar a los constructores de las clases de los objetos contenidos.

Además, hay que tener cuidado con las referencias a objetos pasados como parámetros para rellenar el contenido de los atributos. **Es conveniente hacer una copia de esos objetos y utilizar esas copias para los atributos,** ya que si se utiliza la referencia que se ha pasado como parámetro, el código cliente de la clase puede tener acceso a ella sin necesidad de pasar por la interfaz de la clase y volveríamos a dejar abierta una puerta pública a algo que quizás sea privado

Si el objeto parámetro que se pasa al constructor formaba parte de otro objeto, esto podría ocasionar un efecto colateral si esos objetos son modificados desde el código cliente de la clase. Podemos sin querer compartir esos objetos con otras partes del código sin ningún tipo de control de acceso.

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

Hay que recordar que solo se crean objetos cuando se llama a un constructor, y que si se realizan reasignaciones o pasos de parámetros, no se están copiando o pasando copias de los objetos, sino referencias de ellas.

En este caso, el objeto se está adueñando de un parámetro objeto que pertenece a otro y que es posible que en un futuro haga uso de él.

3. Herencia

La clase de la que se hereda se llama clase base, clase padre o superclase, la clase hereda es la clase hija, derivada o subclase.

Una clase derivada puede ser clase padre de otra que herede de ella, dando lugar a jerarquía de clases.

Una clase hija no tiene acceso a miembros privados de su clase padre, tan solo a sus públicos o a sus protegidos, a los que solo tienen acceso clases derivadas y las del mismo paquete. Los miembros que sean privados de la clase base son también heredados, pero el acceso a ellos es restringido al propio funcionamiento de la superclase y solo se puede acceder a ellos si la superclase dejó algún medio indirecto para hacerlo. Por ejemplo, a través de un método.

Los miembros de la superclase son heredados por la subclase, algunos pueden ser redefinidos o sobrescritos (overriden), y también se pueden añadir nuevos miembros (especialización).

3.1. Sintaxis de la herencia

En java la herencia se indica mediante la palabra reservada extends.

```
[modificador] class ClasePadre {

     // Cuerpo de la clase

     …

}

[modificador] class ClaseHija extends ClasePadre {

// Cuerpo de la clase

…

}
```

Ejemplo. Una clase Persona contiene atributos: nombre, apellido y fecha de nacimiento.

```
public class Persona {

     String nombre;

     String apellidos;

     GregorianCalendar fechaNacim;

     …

}
```

Pero podemos necesitar la clase Alumno, que comparte esos atributos, pero además tiene características propias (especialización.

```
public class Alumno extends Persona {

     String grupo;

     double notaMedia;

     …

}
```

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

3.2. Acceso a miembros heredados

No es posible acceder a miembros privados de una superclase. Para ello existe el modificador protected, el cual permite el acceso a todas las clases heredadas, pero no desde fuera de las clases (estrictamente hablando, desde fuera del paquete, que serían como miembros privados).

Cuadro de niveles accesibilidad a los atributos de una clase

	Misma clase	Subclase	Mismo paquete	Otro paquete
Sin modificador (paquete)	X		X	X
public	X	X	X	X
private	X			
protected	X	X	X	

Al definir la clase Alumno como heredera de Persona, no habrías tenido acceso a esos atributos si declaramos los atributos como protected, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar esos atributos como protected o bien sin modificador (para que también tengan acceso a ellos otras clases del mismo paquete, si es que se considera oportuno):

```
public class Persona {

     protected String nombre;

     protected String apellidos;

     …

}
```

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

Los atributos heredados por una clase son a efectos prácticos iguales que aquellos definidos en la clase derivada.

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

3.3.1. Utilización de miembros heredados (II). Métodos

Los métodos también se heredan en la clase derivada, sumándose a los que se implementen en ella.

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

- Seis por ser Persona (getNombre, getApellidos, getFechaNacim, setNombre, setApellidos, setFechaNacim).
- Otros cuatro más por ser Alumno (getGrupo, setGrupo, getNotaMedia, setNotaMedia).

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

3.4. Redefinición de métodos heredados

Una clase puede redefinir algunos métodos que ha heredado de su clase base, este nuevo método (especializado) sustituye al heredado. También se conoce como sobreescritura de métodos.

Igualmente, se puede acceder al método original con la referencia super, aunque haya sido sobrescrito o redefinido, pero solo se podrá hacer eso con los métodos heredados de la clase padre inmediatamente superior.

Los métodos redefinidos pueden ampliar su accesibilidad con respecto a la del método original, pero nunca la restringen. Un método declarado protected o de paquete en la clase padre, puede redefinirse como public en la clase derivada.

Los métodos estáticos o de clase no pueden ser sobreescritos. Los originales de la clase base permanecen inalterables.

Cuando sobrescribas un método heredado en Java puedes incluir la anotación @Override. Esto indicará al compilador que tu intención es sobrescribir el método de la clase padre. De este modo, si te equivocas (por ejemplo, al escribir el nombre del método) y no lo estás realmente sobrescribiendo, el compilador producirá un error y así podrás darte cuenta del fallo. En cualquier caso, no es necesario indicar @Override

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

```
@override

public String getApellidos () {

     return “Alumno: “ + apellidos;

}
```

Ejercicio resuelto

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

Clase Alumno.

Al heredar de la clase Persona tan solo es necesario escribir métodos para los nuevos atributos (métodos especializados de acceso a los atributos especializados), pues los métodos genéricos (de acceso a los atributos genéricos) ya forman parte de la clase al haberlos heredado. Esos son los métodos que se implementaron en el ejercicio anterior (getGrupo, setGrupo, etc.).

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

```
// Método getNombre

@Override

public String getNombre (){
```

```
            return “Alumno: “ + this.nombre;
}
```

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

Clase Profesor.

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

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

3.5. Ampliación de métodos heredados

En algunas ocasiones, podemos ampliar el comportamiento de un método en vez de ampliarlo.

Para poder preservar el comportamiento del método de la superclase y añadir el nuevo se puede invocar desde el método ampliador de la clase derivada al método ampliado, usando la referencia super.

Esta referencia es una referencia a la clase padre de la que te encuentres en cada momento.

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

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

Este tipo de ampliaciones de métodos resultan especialmente útiles por ejemplo en el caso de los constructores, donde se podría ir llamando a los constructores de cada superclase encadenadamente hasta el constructor de la clase en la cúspide de la jerarquía (el constructor de la clase Object).

Ejercicio resuelto

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

Método mostrar de la clase Persona.

```
public void mostrar () {

    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");

    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());

    System.out.printf ("Nombre: %s\n", this.nombre);

    System.out.printf ("Apellidos: %s\n", this.apellidos);

    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);

}
```

Método mostrar de la clase Profesor.

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

```
public void mostrar () {

    super.mostrar ();     // Llamada al método “mostrar” de la superclase

    // A continuación mostramos la información “especializada” de esta subclase

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

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

}
```

Método mostrar de la clase Alumno.

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

```
public void mostrar () {

    super.mostrar ();

    // A continuación mostramos la información “especializada” de esta subclase

    System.out.printf ("Grupo: %s\n", this.grupo);

    System.out.printf ("Nota media: %5.2f\n", this.notaMedia);

}
```

3.6. constructores y herencia

Un constructor de una clase puede llamar a otro constructor de la misma clase a través de la referencia this. En estos casos la utilización de this solo puede hacerse en la primera línea del

código constructor.

Un constructor de una clase derivada puede hacer algo parecido para llamar al constructor de su clase base con la palabra super. El constructor de una clase derivada puede llamar primero al constructor de su clase base para que inicialice los atributos heredados y después inicializar los atributos específicos de la clase.

Esta llamada también debe ser la primera sentencia de un constructor (a excepción de que exista una llamada a otro constructor de la clase mediante this).

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

En el caso del constructor por defecto de la subclase si el programador no ha escrito ninguno, antes de la inicialización de los atributos a sus valores por defecto, se hace una llamada al constructor de la clase base mediante super.

Cuando se destruye un objeto (método finalize), es importante llamar a los finalizadores en orden inverso a como fueron llamados los constructores. Primero se liberan los recursos de la clase derivada y después los de la clase base mediante super.finalize()

Si la clase Persona tuviera un constructor de ete tipo:

```
public Persona (String nombre, String apellidos, GregorianCalendar fechaNacim) {

     this.nombe= nombre;

     this.apellidos= apellidos;

     this.fechaNacim= new GregorianCalendar (fechaNacim);

}
```

Puedes llamarlo desde un constructor de una clase derivada (Alumno) de la siguiente forma:

```
public Alumno (String nombre, String apellidos, GregorianCalendar fechaNacim, String grupo, double notaMedia) {

     super (nombre, apellidos, fechaNacim);

     this.grupo= grupo;

     this.notaMedia= notaMedia;

}
```

3.7. Cración y utilización de clases derivadas

La idea de la herencia es simplificar los programas al máximo y procurar que haya que escribir la menor cantidad posible de código repetitivo para facilitar la realización de cambios.

En este enlace se muestran ejemplos de utilización de herencia y clases derivadas:

https://www.youtube.com/watch?v=Nu2ziz9Sq0g

3.8. La clase Object en java

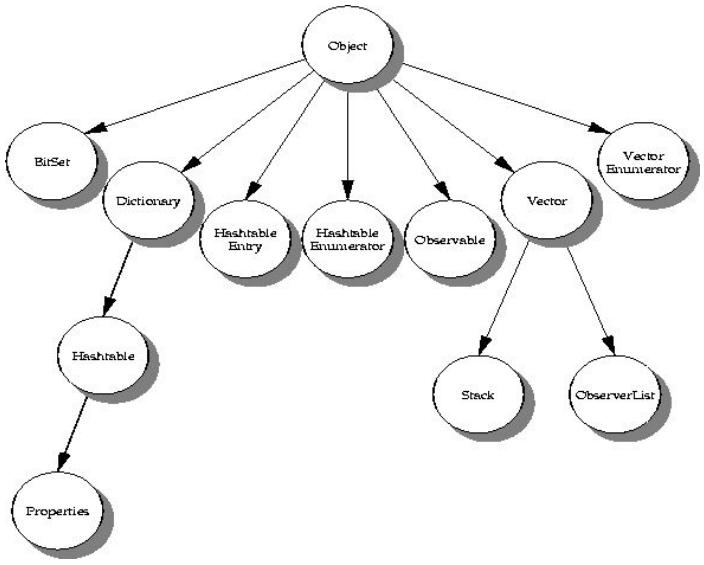
Todas las clases en Java son descendentes de la clase object, la cual define estados y comportamientos básicos que deben tener los objetos. Entre ellos se encuentra:

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

Principales métodos de la clase Object

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

La clase Object representa la superclase que se encuentra en la cúspide de la jerarquía de herencia en Java.



3.9. Herencia múltiple

Se podría considerar la posibilidad de necesitar heredar de más de una clase y disponer de los miembros de dos o más clases disjuntas. El problema en estos casos es la posibilidad de producir ambigüedades, por ejemplo, si tenemos miembros con el mismo identificador en clases distintas, ¿qué miembro se hereda? Los compiladores suelen solicitar que ante casos de ambigüedad, se especifique de manera explícita la clase de la cual se quiere determinar un miembro ambiguo.

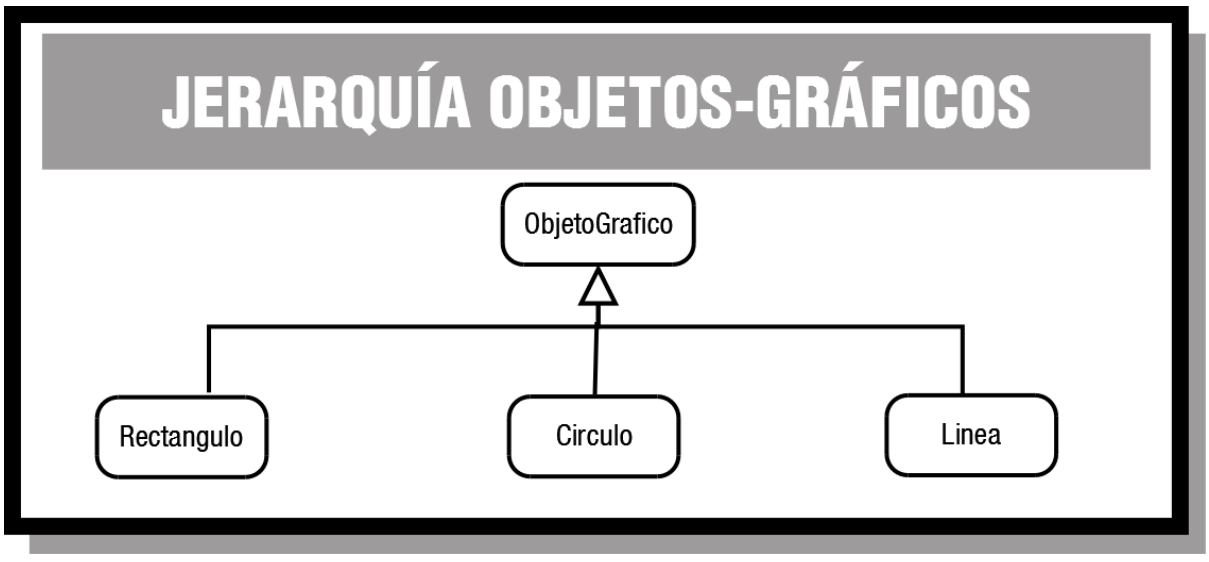
No todos los lenguajes de programación soportan la herencia múltiple y java es uno de ellos.

4. Clases abstractas

En algunos casos puede resultar útil disponer de clases que no van a ser instanciadas, pero que proporcionan un marco o modelo a seguir por sus clases derivadas dentro de la jerarquía de herencia.

Permiten dar unas líneas generales de cómo es una clase sin tener que implementar todos sus métodos, o implementando solo algunos de ellos. De esta manera, las clases derivadas podrán usar los mismos métodos de la clase abstracta, especificando su implementación para cada subclase.

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



4.1. Declaración de una clase abstracta

Una clase abstracta no se puede instanciar. La idea es permitir a otras clases que heredan de ella, proporcionar un modelo genérico y algunos métodos de utilidad general.

```
[modificador_acceso] abstract class nombreClase  [herencia] [interfaces] {

     …

}
```

Una clase puede contener en su interior métodos abstractos, de los cuales solo se indica la cabecera pero no su implementación. En ese caso, la clase necesariamente será abstract y esos métodos serán implementados en sus clases derivadas.

Cuando se trabaja con clases abstractas se debe tener en cuenta:

- Solo se puede usar para crear clases derivadas. No se puede hacer un new de una clase abstracta.
- Puede contener métodos totalmente definidos y métodos abstractos.

Ejercicio resuelto

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

En este caso lo único que habría que hacer es añadir el modificador abstract a la clase Persona. El resto de la clase permanecería igual y las clases Alumno y Profesor no tendrían porqué sufrir ninguna modificación.

```
public abstact class Persona {

        protected String nombre;

        protected String apellidos;

        protected GregorianCalendar fechaNacim;

        ...

}
```

4.2. Métodos abstractos

Su implementación no se define, se declara únicamente su interfaz o cabecera y su cuerpo será implementado en la clase derivada.

Un método se declara como abstracto con el modificador abstract.

```
[modificador_acceso] abstract  <tipo> <nombreMetodo> ([parámetros]) [excepciones];
```

Estos deben ser obligatoriamente definidos en las clases derivadas. Si se deja algún método abstracto sin implementar, esa clase derivada también será una clase abstracta.

Con métodos abstracto debemos tener en cuenta:

- implica que la clase a la que pertenece debe ser abstracta, pero no que todos los métodos tengan que serlo.

- Un método abstracto no puede ser privado ya que no se podría implementar.
- No pueden ser estáticos ya que los métodos estáticos no pueden ser redefinidos.

Ejercicio resuelto

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

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

Dado que el método mostrar no va a ser implementado en la clase Persona, será declarado como abstracto y no se incluirá su implementación:

```
protected abstract void mostrar ();
```

Recuerda que el simple hecho de que la clase Persona contenga un método abstracto hace que sea clase sea abstracta (y deberá indicarse como tal en su declaración): public abstract class Persona.

En el caso de la clase Alumno habrá que hacer una implementación específica del método mostrar y lo mismo para el caso de la clase Profesor.

1. Método mostrar para la clase Alumno.

```
// Redefinición del método abstracto mostrar en la clase Alumno

public void mostrar () {

    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");

    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());

    System.out.printf ("Nombre: %s\n", this.nombre);

    System.out.printf ("Apellidos: %s\n", this.apellidos);

    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);

    System.out.printf ("Grupo: %s\n", this.grupo);

    System.out.printf ("Grupo: %5.2f\n", this.notaMedia);

}
```

2. Método mostrar para la clase Profesor.

```
// Redefinición del método abstracto mostrar en la clase Profesor

public void mostrar () {

    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");

    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());
```

```
    System.out.printf ("Nombre: %s\n", this.nombre);

    System.out.printf ("Apellidos: %s\n", this.apellidos);

    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);

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

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

}
```

4.3. Clases y métodos finales

El modificador final en clases y métodos tienen un comportamiento diferente al que tienen en atributos y variables.

Una clase declarada como final no puede ser heredada (no puede tener clases derivadas. Un método declarado como final no podrá ser redefinido en la clase derivada.

```
[modificador_acceso] final class nombreClase  [herencia] [interfaces]
[modificador_acceso] final <tipo> <nombreMetodo> ([parámetros]) [excepciones]
```

El modificador final también puede ir acompañando a un parámetro de un método, en ese caso no se puede modificar el parámetro dentro del código del método:

```
public final metodoEscribir (int par1, final int par2).
```

5. Interfaces

Podemos llegar a tener una clase abstracta donde todos sus métodos sean abstractos. De este modo, solo damos a las subclases el marco de comportamiento, sin ningún método implementado. La idea de la interfaz (o interface) es disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación, no necesariamente jerárquica.

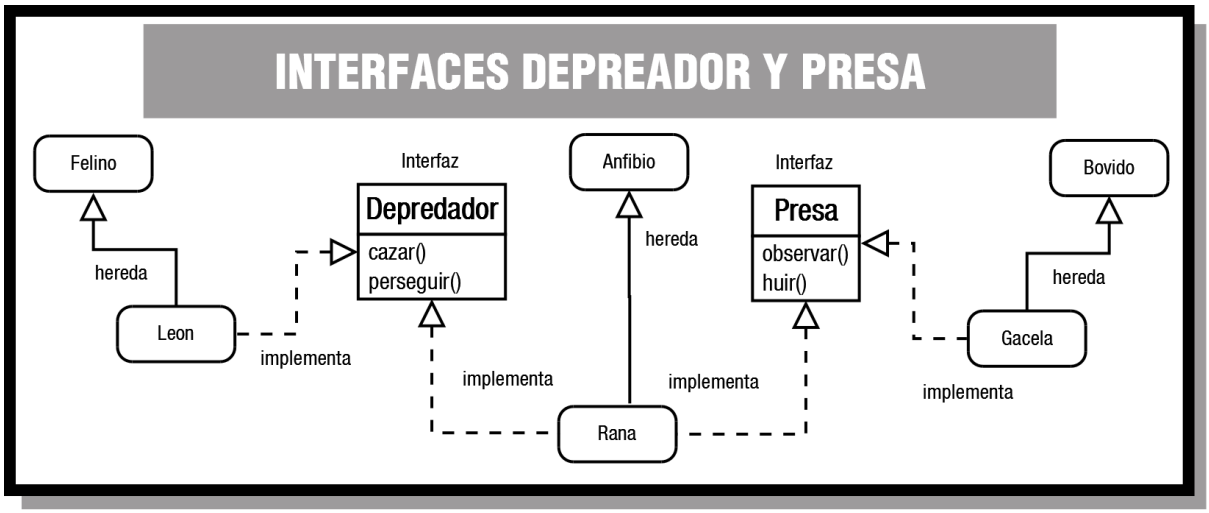
Una interfaz es una lista de declaraciones de métodos sin implementar que caracterizan un determinado comportamiento. Si se desea que una clase tenga ese comportamiento, debe implementar todos los métodos de la interfaz.

La diferencia entre herencia e interfaz es que en la herencia, la clase A es una especialización de B y en el caso de la interfaz A implementa el comportamiento o los métodos establecidos en B.

Ejemplo:

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieres que lleven a cabo están relacionadas con el hecho de que algunos animales sean depredadores (por ejemplo: observar una presa, perseguirla, comérsela, etc.) o sean presas (observar, huir, esconderse, etc.). Si creas la clase León, esta clase podría implementar una interfaz Depredador, mientras que otras clases como Gacela implementarían las acciones de la interfaz Presa. Por otro lado, podrías tener también el caso de la clase Rana, que implementaría las acciones de la interfaz Depredador (pues es cazador de

pequeños insectos), pero también la de Presa (pues puede ser cazado y necesita las acciones necesarias para protegerse).



5.1. Concepto de interfaz

Una interfaz se encarga de establecer que comportamientos hay que tener (qué métodos), pero no dice como se deben llevar a cabo (implementación).

Los métodos de la interfaz son públicos y se deben definir todos y cada uno de ellos en las subclases.

En definitiva, una interfaz se encarga de establecer unas líneas generales sobre los comportamientos de todos los objetos que implementan esa interfaz, pero no indican lo que el objeto es, que de eso se encarga la misma clase y sus superclases. Solo indica las acciones que el objeto debe ser capaz de realizar. Por eso en java muchas interfaces terminan con el sufijo -able, -or, o -ente (capacidad o habilidad): configurable, serializable, modificable, administrador, servidor, buscador... dando así la idea de que se tiene que llevar a cabo el conjunto de acciones especificadas en la interfaz.

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

Ejemplo:

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

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

5.1.1. ¿Clase abstracta o interfaz?

Existe un parecido formal entre una clase abstracta y una interfaz, pudiéndose en ocasiones utilizar de manera indistinta para el mismo fin, pero existen algunas diferencias muy importantes.

- Una clase no puede heredar varias clases, aunque sean abstractas (herencia múltiple), pero si puede implementar una o varias interfaces y además seguir heredando de una clase.
- Una interfaz no puede definir métodos, tan solo declara o enumera.
- Una interfaz puede hacer que dos clases tengan un mismo comportamiento, aunque no hereden de la misma superclase.
- Las interfaces permiten establecer un comportamiento sin apenas dar detalles, ya que esos detalles dependen de como cada clase decida implementar la interfaz.
- Las interfaces tienen su propia jerarquía, diferente e independiente a la jerarquía de clases.

Una clase abstracta proporciona una interfaz disponible solo a través de la herencia, es decir, solo quienes hereden de esa clase abstracta dispondrán de esa interfaz.

Por tanto, si quisiéramos erróneamente que una clase que no hereda de una clase abstracta implemente sus métodos solo se podría realizar volviendo a escribir la jerarquía de clases (redundante) o convertir esa clase en clase derivada de la clase abstracta aunque no tengan nada que ver.

En cambio, una interfaz al poder ser implementada por cualquier clase, nos dejará compartir un determinado comportamiento sin tener que forzar una relación de herencia que no existe.

Así pues, la idea de compartir un determinado comportamiento o interfaz es otro tipo de relación entre clases. Dos clases pueden tener en común un determinado comportamiento sin tener que forzar una relación de herencia.

Recomendación:

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

5.2. Definición de interfaces

En java es similar a la declaración de una clase, con algunas variaciones.

- Se utiliza la palabra reservada interface en lugar de class.
- Puede utilizarse el modificador public. Si se incluye, la interfaz debe tener el mismo nombre que el archivo .java en el que se encuentra, si no se indica, el acceso será por omisión o de paquete como ocurre con las clases.
- Los miembros de la interfaz son public de manera implícita, no es necesario indicar el modificador pero se puede hacer.

- Todos los atributos son de tipo final y public, tampoco es necesario especificarlo. Hay que darles valor inicial.
- Todos los métodos son abstractos de manera implícita, no hace falta indicarlo. No tienen cuerpo, tan solo cabecera, se utiliza punto y coma para terminar de definirlas.

```
[public] interface <NombreInterfaz> {

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

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

                ...

                [public] [abstract] <tipo_devuelto1>  <nombreMetodo1> ([lista_parámetros]);

                [public] [abstract] <tipo_devuelto2>  <nombreMetodo2> ([lista_parámetros]);

                ...

}
```

Ejemplo:

```
public interface Depredador {

   void localizar (Animal presa);

   void cazar (Animal presa);

   ...

}
```

Ejercicio resuelto

Crea una interfaz en Java cuyo nombre sea Imprimible que contenga un método útil para mostrar el contenido de una clase:

Método devolverContenidoString, que crea un String con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves: "{<nombre_atributo_1>=<valor_atributo_1>, ..., <nombre_atributo_n>=<valor__atributo_n>}".

```
public interface  Imprimible {

                String devolverContenidoString ();

}
```

5.3. Implementación de interfaces

Todas las clases que implementan una determinada interfaz están obligadas a proporcionar una definición o implementación de los métodos de la interfaz.

Dada una interfaz, cualquier clase puede especificar dicha interfaz mediante la palabra reservada implements:

```
class NombreClase implements NombreInterfaz {
```

Es posible indicar varias interfaces separándolas por comas:

```
class NombreClase implements NombreInterfaz1, NombreInterfaz2,... {
```

Al redefinir los métodos se debe hacer con acceso público o se produce error de compilación. De modo que ni se pueden restringir los permisos de acceso a la herencia de clases, ni tampoco a la implementación de interfaces.

Recomendación:

Para añadir la implementación de los métodos de una clase que implementa una interfaz, podemos utilizar la funcionalidad de Netbeans de agregarlos de forma automática exactamente igual que hicimos con los métodos abstractos. Agiliza la inserción de código pues nos ahorramos escribir la cabecera de todos los métodos a los que tenemos que dar implementación.

Ejemplo:

```
class Leon implements Depredador {
  void localizar (Animal presa) {

  // Implementación del método localizar para un león

  ...

  }
}
```

En el caso de que se pudiera ser depredador y presa se deben implementar ambas interfaces.

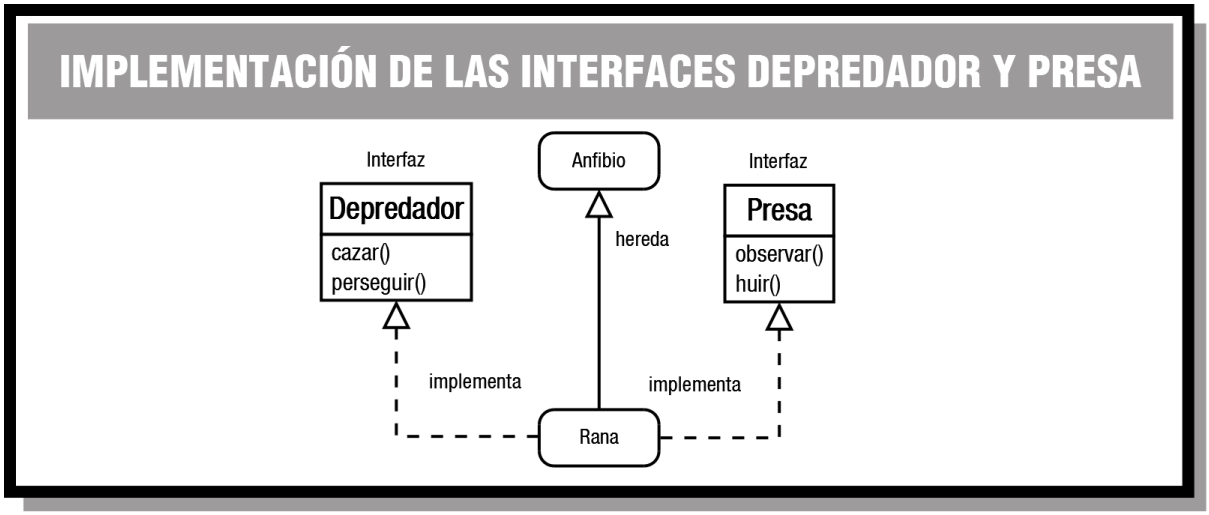
```
class Rana implements Depredador, Presa {
```

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

Una interfaz no tiene espacio de almacenamiento asociado, es decir, no tiene implementación.

En java no se permite la herencia múltiple, pero se puede simular definiendo interfaces que indiquen los comportamientos o métodos que debería tener según pertenezca a una supuesta clase, pero sin implementar ningún método concreto ni atributos de objeto, solo interfaz.

Ejemplo: Una clase X puede implementar las interfaces A, B y C, que la dotan de comportamientos que deseaba heredar de las clases A, B y C, a su vez puede heredar de otra clase Y, que le proporciona sus características dentro de su jerarquía de objeto.



De este modo, con una herencia y varias interfaces se consiguen resultados similares a la herencia múltiple.

Puede darse el caso de la colisión de nombres cuando se implementan dos interfaces con un método llamado igual.

- Si los dos métodos tienen distintos parámetros ocurrirá sobrecarga de métodos y no habrá problema.
- Si tienen un valor de retorno de un tipo diferente se produce error de compilación.
- Si los dos métodos son iguales de parámetro y tipo devuelto, solo se podrá implementar uno de los dos métodos.

5.5. Herencia de interfaces

Las interfaces permiten herencia, se hace con la palabra reservada extends, pero en este caso si se permite herencia múltiple de interfaces. Si se hereda más de una interfaz se separan por comas.

```
public interface InterfazCompleja extends InterfazUno, InterfazDos {

                // Métodos y constantes de la interfaz compleja

}
```

6. Polimorfismo

Es otro de los grandes pilares en los que se sustenta la POO y también establece diferencias entre interfaz e implementación (entre el qué y el como).

El polimorfismo es fundamental a la hora de manipular muchos objetos de clases distintas como si fueran de la misma clase.

Te permite mejorar la organización y legibilidad del código, así como desarrollar aplicaciones fáciles de ampliar e incorporar nuevas funcionalidades.

6.1. Concepto de polimorfismo

Consiste en poder referenciar un objeto de una determinada clase como si fuera de otra (que sea subclase).

Un método polimórfico ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en tiempo de ejecución en lugar de en tiempo de compilación. Para eso es necesario utilizar métodos que pertenecen a una superclase y que están implementados en las subclases de una forma particular.

Solo en tiempo de ejecución (una vez instanciada una u otra clase) se conoce realmente que método de qué subclase es invocado.

Esto ayuda a desentenderte del tipo de objeto específico para centrarte en el objeto genérico y se pueden manipular objetos hasta cierto punto desconocidos en tiempo de compilación.

El polimorfismo ofrece la posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una de sus subclases. Esto te va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus superclases. Se puede llevar a cabo tanto con superclases (abstractas o no), como con interfaces.

Ejemplo:

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

6.2. Ligadura dinámica

La conexión que tiene durante una llamada a un método se suele llamar ligadura, vinculación o enlace (binding). Si al vinculación se realiza en compilación se llama ligadura estática o vinculación temprana.

En POO existe la ligadura dinámica, conocida como vinculación tardía, enlace tardío o late binding.

Hace posible que sea el tipo de objeto instanciado, obtenido mediante el constructor utilizado para crear el objeto y no el tipo de la referencia, lo que determine que versión de método será invocado. El tipo de objeto al que apunta la variable de referencia solo podrá ser conocido en la ejecución del programa.

Ejercicio resuelto

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

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

La clase Instrumento podría tener un único método (tocarNota):

```
public abstract class Instrumento {

    public void tocarNota (String nota) {

        System.out.printf ("Instrumento: tocar nota %s.\n", nota);

    }

}
```

En el caso de las clases Piano y Flauta puede ser similar, heredando de Instrumento y redefiniendo el método tocarNota:

```
public class Flauta extends Instrumento {

    @Override

    public void tocarNota (String nota) {

        System.out.printf ("Flauta: tocar nota %s.\n", nota);

    }

}

public class Piano extends Instrumento {

    @Override

    public void tocarNota (String nota) {

        System.out.printf ("Piano: tocar nota %s.\n", nota);

    }

}
```

la hora de declarar una referencia a un objeto de tipo instrumento, utilizamos la superclase (Instrumento):

```
Instrumento instrumento1; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
```

Sin embargo, a la hora de instanciar el objeto, utilizamos el constructor de alguna de sus subclases (Piano, Flauta, etc.):

```
if (<condición>) {

            // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)

            instrumento1= new Piano ();

 }

else if (<condición>) {

// Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)

   instrumento1=  new Flauta ();
```

```
 } else {

            ...

 }
```

Finalmente, a la hora de invocar el método tocarNota, no sabremos a qué versión (de qué subclase) de tocarNota se estará llamando, pues dependerá del tipo de objeto (subclase) que se haya instanciado. Se estará utilizando por tanto la ligadura dinámica:

```
// Interpretamos una nota con el objeto instrumento1

   // No sabemos si se ejecutará el método tocarNota de Piano o de Flauta

  // (dependerá de la ejecución)

        instrumento1.tocarNota ("do");   // Ejemplo de ligadura dinámica (tiempo de ejecución)
```

6.3. Limitaciones de la ligadura dinámica

Existe una importante restricción en el uso de esta capacidad, pues el tipo de referencia limita cuáles son los métodos a utilizar y los atributos alos que acceder.

No se puede acceder a los miembros específicos de una subclase a través de una referencia a una superclase. Solo se puede acceder a los miembros declarados en la superclase, aunque la definición se la de la subclase.

Ejemplo:

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

Ejercicio resuelto

Si tuviéramos diferentes variables referencia a objetos de las clases Alumno y Profesor tendrías algo así:

```
Alumno obj1;

Profesor obj2:

...

// Si se dan ciertas condiciones el objeto será de tipo Alumno y lo tendrás en obj1

System.out.printf ("Nombre: %s\n", obj1.getNombre());

// Si se dan otras condiciones el objeto será de tipo Profesor y lo tendrás en obj2

System.out.printf ("Nombre: %s\n", obj2.getNombre());
```

Pero si pudieras tratar de una manera más genérica la situación, podrías intentar algo así:

```
Persona obj;
```

```
// Si se dan ciertas condiciones el objeto será de tipo Alumno y por tanto lo instanciarás como tal

obj = new Alumno (<parámetros>);

// Si se otras condiciones el objeto será de tipo Profesor y por tanto lo instanciarás como tal

obj = new Profesor (<parámetros>);
```

De esta manera la variable obj obj podría contener una referencia a un objeto de la superclase Persona de subclase Alumno o bien de subclase Profesor (polimorfismo).

Esto significa que independientemente del tipo de subclase que sea (Alumno o Profesor), podrás invocar a métodos de la superclase Persona y durante la ejecución se resolverán como métodos de alguna de sus subclases:

```
//En tiempo de compilación no se sabrá de qué subclase de Persona será obj.

//Habrá que esperar la ejecución para que el entorno lo sepa e invoque al método adecuado.

System.out.printf ("Contenido del objeto usuario: %s\n", stringContenidoUsuario);
```

Por último recuerda que debes de proporcionar constructores a las subclases Alumno y Profesor que sean "compatibles" con algunos de los constructores de la superclase Persona, pues al llamar a un constructor de una subclase, su formato debe coincidir con el de algún constructor de la superclase (como debe suceder en general con cualquier método que sea invocado utilizando la ligadura dinámica).

6.4. Interfaces y polimorfismo

Es posible llevar a cabo el polimorfismo usando interfaces.

Un objeto puede tener una referencia cuyo tipo sea una interfaz, pero para que el compilador lo permita, la clase cuyo constructor se utilice para crear el objeto debe implementar esa interfaz, ya sea por si misma o porque la implemente alguna superclase.

Un objeto cuya referencia es de tipo interfaz, solo puede utilizar aquellos métodos definidos en la interfaz, no podrá utilizar atributos y métodos específicos de su clase.

Estas referencias permiten unificar de una manera estricta la forma de utilizarse objetos que pertenecen a clases muy diferentes pero que implementan la misma interfaz.

De esta manera podemos hacer referencias a objetos sin relación jerárquica utilizando la misma variable.

Ejemplo:

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

Otro ejemplo:

En el caso de las clases Persona, Alumno y Profesor, podrías declarar, por ejemplo, variables del tipo Imprimible:

```
Imprimible obj;                // Imprimible es una interfaz y no una clase
```

Con este tipo de referencia podrías luego apuntar a objetos tanto de tipo Profesor como de tipo Alumno, pues ambos implementan la interfaz Imprimible:

```
obj= new  Alumno (nombre, apellidos, fecha, grupo, nota); // Polimorfismo con interfaces

...

// En otras circunstancias podría suceder esto:

obj= new  Profesor (nombre, apellidos, fecha, especialidad, salario); // Polimorfismo con interfaces

...
```

```
String contenido;

contenido= obj.devolverContenidoString();  // Ligadura dinámica con interfaces
```

6.5. Conversión de paquetes

Si deseamos tener acceso a todos los métodos y atributos específicos del objeto subclase, podemos realizar una conversión explícita (casting), logrando que convierta la referencia más general en la del tipo específico.

Para ello es obligatorio que exista herencia entre ellas. Se realizará conversión implícita automática siempre que sea necesario, pues un objeto de tipo subclase contendrá toda la información necesaria para ser considerada un objeto superclase.

La conversión de superclase a subclase se debe hacer de forma explícita porque podría dar lugar a errores por falta de información (atributos) o de métodos con ClassCastException.

Ejemplo:

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

```
class ClaseA {

     public int atrib1;

}

class ClaseB extends ClaseA {

     public int atrib2;

}
```

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

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

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

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

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

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

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

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

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

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

// Funciona (la clase A tiene atrib1)

System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib1);

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

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

Anexo I - Elaboración de los constructores de la clase Rectángulo

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

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

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

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

```
public Rectangulo ()
{

     this.vertice1= new Punto (0,0);

     this.vertice2= new Punto (1,1);

}
```

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

```
public Rectangulo (double x1, double y1, double x2, double y2)
{

     this.vertice1= new Punto (x1, y1);

     this.vertice2= new Punto (x2, y2);

}
```

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

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

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

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

Aquí tienes las dos posibles versiones:

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

```
public Rectangulo (Punto vertice1, Punto vertice2)
{

     this.vertice1= vertice1;

     this.vertice2= vertice2;

}
```

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

```
public Rectangulo (Punto vertice1, Punto vertice2)
{

     this.vertice1= new Punto (vertice1.obtenerX(), vertice1.obtenerY() );

     this.vertice2= new Punto (vertice2.obtenerX(), vertice2.obtenerY() );
```

```
}
```

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

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

```
public Rectangulo (Punto vertice1, Punto vertice2)
{

     this.vertice1= new Punto (vertice1 );

     this.vertice2= new Punto (vertice2 );

}
```

Quedaría finalmente por implementar el constructor copia:

```
// Constructor copia

public Rectangulo (Rectangulo r) {

    this.vertice1= new Punto (r.obtenerVertice1() );

    this.vertice2= new Punto (r.obtenerVertice2() );

}
```

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

Anexo II - Métodos para las clases heredadas Alumno y Profesor

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

```
public class Alumno extends Persona {

        protected String grupo;

        protected double notaMedia;

        // Método getNombre

        public String getNombre (){

            return nombre;

        }
```

```
        // Método getApellidos
        public String getApellidos (){
            return apellidos;
        }
        // Método getFechaNacim
        public GregorianCalendar getFechaNacim (){
            return this.fechaNacim;
        }
        // Método getGrupo
        public String getGrupo (){
            return grupo;
        }
        // Método getNotaMedia
        public double getNotaMedia (){
            return notaMedia;
        }
        // Método setNombre
        public void setNombre (String nombre){
            this.nombre= nombre;
        }
        // Método setApellidos
        public void setApellidos (String apellidos){
            this.apellidos= apellidos;
        }
        // Método setFechaNacim
        public void setFechaNacim (GregorianCalendar fechaNacim){
            this.fechaNacim= fechaNacim;
        }
        // Método setGrupo
        public void setGrupo (String grupo){
            this.grupo= grupo;
        }
        // Método setNotaMedia
        public void setNotaMedia (double notaMedia){
            this.notaMedia= notaMedia;
        }
```

```
}
```

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

```
public class Profesor extends Profesor {

        String especialidad;

        double salario;

        // Método getNombre

        public String getNombre (){

            return nombre;

        }
        // Método getApellidos

        public String getApellidos (){

            return apellidos;

        }

        // Método getFechaNacim

        public GregorianCalendar getFechaNacim (){

            return this.fechaNacim;

        }

        // Método getEspecialidad

        public String getEspecialidad (){

            return especialidad;

        }

        // Método getSalario

        public double getSalario (){

            return salario;

        }

        // Método setNombre

        public void setNombre (String nombre){

            this.nombre= nombre;

        }

        // Método setApellidos

        public void setApellidos (String apellidos){

            this.apellidos= apellidos;

        }


        // Método setFechaNacim
```

```
        public void setFechaNacim (GregorianCalendar fechaNacim){

            this.fechaNacim= fechaNacim;

        }

        // Método setSalario

        public void setSalario (double salario){

            this.salario= salario;

        }


        // Método setESpecialidad

        public void setESpecialidad (String especialidad){

            this.especialidad= especialidad;

        }

}
```

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

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

```
public class Persona {

        protected String nombre;

        protected String apellidos;

        protected GregorianCalendar fechaNacim;

        // Método getNombre

        public String getNombre (){

            return nombre;

        }

        // Método getApellidos

        public String getApellidos (){

            return apellidos;

        }

        // Método getFechaNacim

        public GregorianCalendar getFechaNacim (){

            return this.fechaNacim;
```

```
        }
        // Método setNombre
        public void setNombre (String nombre){
            this.nombre= nombre;
        }
        // Método setApellidos
        public void setApellidos (String apellidos){
            this.apellidos= apellidos;
        }
        // Método setFechaNacim
        public void setFechaNacim (GregorianCalendar fechaNacim){
            this.fechaNacim= fechaNacim;
        }
}
```

Mapa Conceptual