Programación

Tema 5

Utilización avanzada de clases en Java

Índice

Tema 5

Sobrecarga de métodos

Sobrecarga de métodos

Antes de ver la sobrecarga es necesario saber que es la **signatura** o **firma** de un método:

En Java , la signatura o firma de un método es parte de la declaración del método.

Es la combinación del nombre del método y la lista de parámetros.

```
public void setMapReference (int xCoordinate, int yCoordinate)
{
// código de método
}
```

setMapReference(int, int)

```
public void setMapReference (posición del punto)
{
// código de método
}
```

setMapReference(Posicion)

Sobrecarga de métodos

En principio podrías pensar que un método puede aparecer una sola vez en la declaración de una clase (no se debería repetir el mismo nombre para varios métodos). Pero no tiene porqué siempre suceder así. Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la **sobrecarga de métodos**.

El lenguaje Java soporta la característica conocida como **sobrecarga de métodos**.

Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre.

La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre **será mediante la lista de parámetros del método**: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.

Sobrecarga de métodos

Imagínate que estás desarrollando una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (**int**), un número real (**double**) o una cadena de caracteres (**String**). Una primera opción podría ser definir un nombre de método diferente dependiendo de lo que se vaya a escribir en el lienzo.

Por ejemplo:

- Método **pintarEntero (int entero).**
- Método **pintarReal (double real).**
- Método **pintarCadena (double String).**
- Método **pintarEnteroCadena (int entero, String cadena).**

Y así sucesivamente para todos los casos que desees contemplar...

Sobrecarga de métodos

La posibilidad que te ofrece la sobrecarga es utilizar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: pintar). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros).

Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo **pintar**, para todos los métodos anteriores:

- Método **pintar (int entero).**
- Método **pintar (double real).**
- Método **pintar (double String).**
- Método **pintar (int entero, String cadena).**

Sobrecarga de métodos

En este caso el compilador no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

Lo que sí habría producido un error de compilación habría sido por ejemplo incluir otro método **pintar (int entero)**, pues es imposible distinguirlo de otro método con el mismo nombre y con la misma lista de parámetros (ya existe un método **pintar** con un único parámetro de tipo **int**).

También debes tener en cuenta que el **tipo devuelto** por el método **no es considerado a la hora de identificar un método**, así que un tipo devuelto diferente no es suficiente para distinguir un método de otro. Es decir, no podrías definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El compilador producirá un error de duplicidad en el nombre del método y no te lo permitirá.

Es conveniente no abusar de sobrecarga de métodos y utilizarla con cierta moderación (cuando realmente puede beneficiar su uso), dado que podría hacer el código menos legible.

Ejercicio

En el ejercicio de la calculadora, sobrecargar todos sus métodos para que acepte valores enteros, flotantes, doubles, string y char.
Dentro de cada método hacer las conversiones correspondientes.
Los métodos a sobrecargar son:

- Sumar
- Restar
- Multiplicar
- Dividir

Probar el constructor desde el método Main()

Sobrecarga de constructores

Si los métodos pueden estar sobrecargados y los constructores no son más que unos métodos especiales, es lógico pensar que los constructores también pueden estar sobrecargados.

Los constructores sobrecargados permiten a los objetos de una clase inicializarse de distintas formas. Para ello, simplemente hay que proporcionar varias declaraciones del constructor con distintas listas de parámetros, al igual que sucede con los métodos normales.

Sobrecarga de métodos. Constructores

Ejemplo

```
//Demostración de Sobrecarga de constructores
class MiClase{
    int x;

    MiClase(){
        System.out.println("Dentro de MiClase().");
        x=0;
    }

    MiClase(int i){
        System.out.println("Dentro de MiClase(int).");
        x=i;
    }

    MiClase(double d){
        System.out.println("Dentro de MiClase(double).");
        x=(int)d;
    }

    MiClase(int i, int j){
        System.out.println("Dentro de MiClase(int, int).");
        x=i*j;
    }
}
```

```
class DemoSobrecargaConstructor{
    public static void main(String[] args) {
        MiClase t1=new MiClase();
        MiClase t2=new MiClase(28);
        MiClase t3=new MiClase(15.23);
        MiClase t4=new MiClase(2,4);

        System.out.println("t1.x: "+ t1.x);
        System.out.println("t2.x: "+ t2.x);
        System.out.println("t3.x: "+ t3.x);
        System.out.println("t4.x: "+ t4.x);
    }
}
```

Salida

```
Dentro de MiClase().
Dentro de MiClase(int).
Dentro de MiClase(double).
Dentro de MiClase(int, int).
t1.x: 0
t2.x: 28
t3.x: 15
t4.x: 8
```

Constructores de copia

Una forma de iniciar un objeto podría ser mediante la copia de los valores de los atributos de otro objeto ya existente. Imagina que necesitas varios objetos iguales (con los mismos valores en sus atributos) y que ya tienes uno de ellos perfectamente configurado (sus atributos contienen los valores que tú necesitas).

Estaría bien disponer de un constructor que hiciera copias idénticas de ese objeto. Durante el proceso de creación de un objeto puedes generar objetos exactamente iguales (basados en la misma clase) que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase que además de generar un objeto nuevo, que lo haga con los mismos valores que tenga otro objeto ya existente.

Es decir, algo así como si pudieras **clonar** el objeto tantas veces como te haga falta.

A este tipo de mecanismo se le suele llamar **constructor copia** o **constructor de copia**.

Sobrecarga de métodos. Constructores de copia

Un constructor copia es un método constructor como los que ya has utilizado pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar.

Este método revisa cada uno de los atributos del objeto recibido como parámetro y se copian todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.

```java
package objetos;
public class Persona {
    private String dni;
    private String nombre;
    private int edad;

    // Constructor sin parámentros
    public Persona() {
        this.dni = "";
        this.nombre = "";
        this.edad = 0;
    }

    // Constructor con parámentros
    public Persona(String dni, String nombre, int edad) {
        this.dni = dni;
        this.nombre = nombre;
        this.edad = edad;
    }

    // Constructor copia
    public Persona(Persona objPersona){

        // pasamos a cada variable lo que contiene
        // el cada persona...
        this.dni=objPersona.dni;
        this.nombre=objPersona.nombre;
        this.edad=objPersona.edad;
    }
}
```

Ejercicio

Declarar el constructor de copia en las clases creadas con anterioridad:

- Persona.
- Cuenta Corriente.
- Punto.

Probar el constructor desde el método Main()

Índice

La referencia *this* y el método *this*()

La palabra reservada ***this*** consiste en una referencia al objeto actual. El uso de este operador puede resultar muy útil a la hora de evitar la ambigüedad que puede producirse entre el nombre de un parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo identificador (mismo nombre).

En tales casos el parámetro "oculta" al atributo y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al atributo). En estos casos la referencia ***this*** nos permite acceder a estos atributos ocultados por los parámetros.

La referencia *this* y el método *this*()

Dado que *this* es una referencia a la propia clase en la que te encuentras en ese momento, puedes acceder a sus atributos mediante el operador punto (.) como sucede con cualquier otra clase u objeto.

Por tanto, en lugar de poner el nombre del atributo (que estos casos haría referencia al parámetro), podrías escribir **this.nombreAtributo**, de manera que el compilador sabrá que te estás refiriendo al atributo y se eliminará la ambigüedad.

En algunos casos puede resultar útil hacer uso de la referencia **this** aunque no sea necesario, pues puede ayudar a mejorar la legibilidad del código.

La referencia *this* y el método *this*()

Ejemplo:

```
2  public class Potencia {
3      private double base;
4      private int exp;
5      private double valor;
6
7      public double getBase() {
8          return base;
9      }
10
11     public void setBase(double base) {
12         base = base;
13     }
14
15     public int getExp() {
16         return exp;
17     }
18
19     public void setExp(int exp) {
20         exp = exp;
21     }
22
23     public double getValor() {
24         if(exp == 0) return 0;
25         valor = 1;
26         for(; exp > 0; exp--) valor = valor * base;
27         return valor;
28     }
29
30     public void setValor(double valor) {
31         valor = valor;
32     }
33
34 }
```

Desde el método Main

```
Potencia p = new Potencia();

p.setBase(4);
p.setExp(2);

System.out.println(p.getValor());
```

La referencia *this* y el método *this*()

Ejemplo:

```java
public class Potencia {
    private double base;
    private int exp;
    private double valor;

    public double getBase() {
        return this.base;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public int getExp() {
        return exp;
    }

    public void setExp(int exp) {
        this.exp = exp;
    }

    public double getValor() {
        if(this.exp == 0) return 0;
        this.valor = 1;
        for(; exp > 0; exp--) this.valor = this.valor * this.base;
        return this.valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }
}
```

Desde el método Main

```java
Potencia p = new Potencia();

p.setBase(4);
p.setExp(2);

System.out.println(p.getValor());
```

La referencia *this* y el método *this*()

```
2  public class Potencia {
3
4      private double valor;
5
6      Potencia(double base, int exp) {
7          this.valor = 1;
8          if (exp == 0) return;
9          for(; exp > 0; exp--) this.valor = this.valor * base;
10     }
11
12     public double getValor() {
13         return this.valor;
14     }
15 }
```

Desde el método Main

```
Potencia p = new Potencia(4, 2);
System.out.println(p.getValor());
```

Ejercicio

Corregir las clases creadas hasta el momento:

- Persona.
- Cuenta Corriente.
- Punto.

y hacer uso del operador this para acceder en todos sus atributos a través de este operador, comprobando que los programas siguen funcionando correctamente.

La referencia *this* y el método *this*()

Pero aparte de la referencia *this*, también existe lo que se conoce como el **método this()**.

Veamos un ejemplo:

Imagina una clase con un constructor sobrecargado:

- Uno de los métodos constructores crea el objeto dándole valores iniciales al NIF y al nombre del trabajador.
- Otro de los métodos constructores crea el objeto dándole valores iniciales al NIF, al nombre del trabajador y a la fecha de alta en la empresa.

La referencia *this* y el método *this*()

Como vemos, el segundo de los métodos constructores hace lo mismo que el primero, y además hace alguna otra cosa más. ¿No sería bueno no tener que repetir en el segundo de los constructores el mismo código que ya usamos en el primero?

Pues esto es posible haciendo uso del método this().

El método this() se usa para hacer referencia dentro de un constructor de una clase a otro constructor sobrecargado de la misma clase, aquel que coincida con la lista de parámetros de la llamada.

Importante: la sentencia this(), debe ser la primera línea dentro del constructor.

La referencia *this* y el método *this*()

```
14⊖     public Persona() {
15          System.out.println("Llamada a Persona()");
16          numeroObjetos++;
17      }
18
19⊖     public Persona(String nombre) {
20          this();
21          System.out.println("Llamada a Persona(String nombre)");
22          this.nombre = nombre;
23      }
24
25⊖     public Persona(String nombre, String apellidos) {
26          this(nombre);
27          System.out.println("Llamada a Persona(String nombre, String apellidos)");
28          this.apellidos = apellidos;
29      }
```

La referencia *this* y el método *this*()

```
14      public Persona() {
15          System.out.println("Llamada a Persona()");
16          numeroObjetos++;
17      }
18
19      public Persona(String nombre) {
20          System.out.println("Llamada a Persona(String nombre)");
21          this();
22          this.n
23      }
24
25      public Persona(String nombre, String apellidos) {
26          this(nombre);
27          System.out.println("Llamada a Persona(String nombre, String apellidos)");
28          this.apellidos = apellidos;
29      }
```

Constructor call must be the first statement in a constructor

Press 'F2' for focus

Ejercicio.

- Probar que el anterior ejemplo con la clase Persona, la clase Punto y la clase Cuenta Corriente.
- Sobrecargar el constructor con los atributos que queráis.
- Llamar a los constructores correspondientes con el método this().
- Llamar a cada uno de los constructores desde el método main y comprobar que el funcionamiento es el correcto.

Índice

Tema 5

Relación entre las clases

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

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

En algunos casos es posible que no exista relación alguna entre unas clases y otras, pero lo más habitual es que sí exista: una clase puede ser una **especialización** de otra, o bien una **generalización**, o una clase contiene en su interior objetos de otra, o una clase utiliza a otra, etc. Es decir, que entre unas clases y otras habrá que definir cuál es su relación (si es que existe alguna).

Relación entre las clases

Se pueden distinguir diversos **tipos de relaciones** entre clases:

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

Relación entre las clases.

La relación de **clientela** la llevas utilizando desde que has empezado a programar en Java, pues desde tu clase principal (clase con método **main**) has estado declarando, creando y utilizando objetos de otras clases.

Por ejemplo: si utilizas un objeto **String** dentro de la clase principal de tu programa, éste será cliente de la clase **String** (como sucederá con prácticamente cualquier programa que se escriba en Java).

Es la relación fundamental y más habitual entre clases (la **utilización** de unas clases por parte de otras) y, por supuesto, la que más vas a utilizar tú también, de hecho, ya la has estado utilizando y lo seguirás haciendo.

Relación entre las clases.

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

Por ejemplo, si escribes una clase donde alguno de sus atributos es un objeto de tipo **String**, ya se está produciendo una relación de tipo **composición** (tu clase "tiene" un **String**, es decir, está compuesta por un objeto **String** y por algunos elementos más).

Relación entre las clases.

La relación de **anidamiento** (o **anidación**) es quizá menos habitual, pues implica declarar unas clases dentro de otras (**clases internas** o **anidadas**). En algunos casos puede resultar útil para tener un nivel más de encapsulamiento y ocultación de información.

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

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

Índice

Tema 5

Composición

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

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

Por otro lado, los objetos de la clase **ComunidadAutonoma** podrían contener como atributos objetos de la clase **Provincia**, la cual a su vez también podría contener objetos de la clase **Municipio**.

Composición

Como puedes observar, la **composición** puede encadenarse todas las veces que sea necesario hasta llegar a objetos básicos del lenguaje o hasta tipos primitivos que ya no contendrán otros objetos en su interior.

Ésta es la forma más habitual de definir clases: mediante otras clases ya definidas anteriormente.

Es una manera eficiente y sencilla de gestionar la reutilización de todo el código ya escrito. Si se definen clases que describen entidades distinguibles y con funciones claramente definidas, podrán utilizarse cada vez que haya que representar objetos similares dentro de otras clases.

Composición

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

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

- Un **coche** tiene un **motor** y tiene cuatro **ruedas**.
- Una **persona** tiene un **nombre**, una **fecha de nacimiento**, una **cuenta bancaria** asociada para ingresar la nómina, etc.
- Un **cocodrilo** bajo investigación científica que tiene un número de **dientes** determinado, una **edad**, unas **coordenadas** de ubicación geográfica (medidas con GPS), etc.

Composición

Para indicar que una clase contiene objetos de otra clase no es necesaria **ninguna sintaxis especial**.

Cada uno de esos objetos no es más que un atributo y, por tanto, debe ser declarado como tal:

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

Composición

Ejercicio

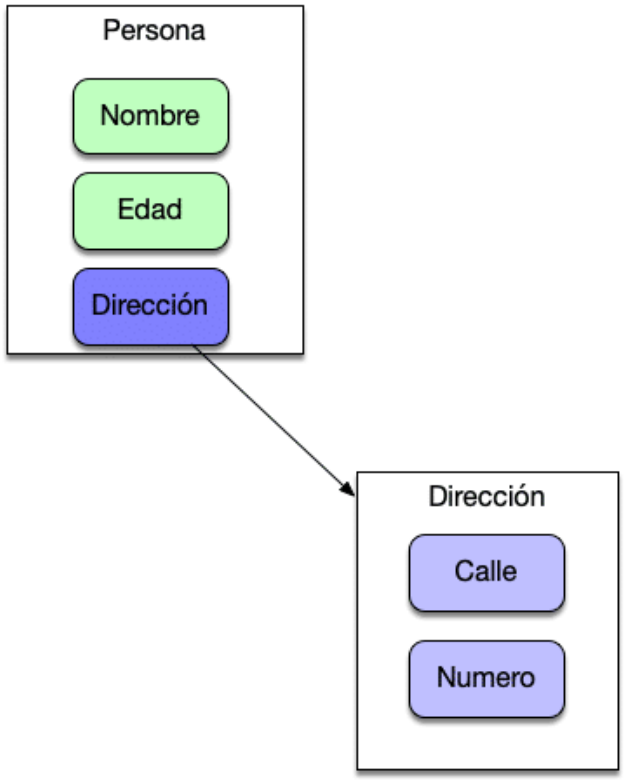
Implementar el siguiente diagrama:

Crear la clase Persona, con los atributos:

- Nombre
- Edad
- Direccion

Crear la clase Direccion, con los atributos:

- Calle
- Número



Composición

En unidades anteriores has trabajado con la clase **Punto**, que definía las coordenadas de un punto en el plano, con su posición x y con su posición y.

Vamos a definir la clase **Rectángulo,** que define una figura de tipo rectángulo como una figura en el plano que se define a partir de dos de sus **vértices** (**inferior izquierdo** y **superior derecho**).

- Atributos **x1**, **y1**, que representan la coordenadas del vértice inferior izquierdo del rectángulo. Ambos de tipo **double** (números reales).
- Atributos **x2**, **y2**, que representan las coordenadas del vértice superior derecho del rectángulo. También de tipo **double** (números reales).

Con estos dos puntos (x1, y1) y (x2, y2) se puede definir perfectamente la ubicación de un rectángulo en el plano.

Escribe una clase que contenga todos esos atributos teniendo en cuenta que queremos que sea una clase visible desde cualquier parte del programa y que sus atributos sean también accesibles desde cualquier parte del código.

Composición

Tal y como hemos formalizado ahora los tipos de relaciones entre clases, parece bastante claro que aquí tendrías un caso de **composición**: "**un rectángulo contiene puntos**".

Por tanto, podrías ahora redefinir los atributos de la clase **Rectángulo** (cuatro **números reales**) como dos objetos de tipo **Punto**:

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

Composición. Preservación de la ocultación

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

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

Composición. Llamadas a constructores

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

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

Composición. Preservación de la ocultación

```java
public class Rectangulo {

    private Punto vertice1;
    private Punto vertice2;

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

    public Punto getVertice1() {
        return vertice1;
    }

    public void setVertice1(Punto vertice1) {
        this.vertice1 = vertice1;
    }

    public Punto getVertice2() {
        return vertice2;
    }

    public void setVertice2(Punto vertice2) {
        this.vertice2 = vertice2;
    }

    @Override
    public String toString() {
        return "Rectangulo [vertice1=" + vertice1 + ", vertice2=" + vertice2 + "]";
    }
}
```

Método Main

```java
Rectangulo rectangulo = new Rectangulo();
Punto p1 = rectangulo.getVertice1();
p1.setPosicionX(54);
p1.setPosicionY(12.5);
```

Composición. Preservación de la ocultación

```java
public class Punto {

    private int posicionX;
    private int posicionY;

    public Punto() {
        this.posicionX = (int)Math.random() * 100 + 1;
        this.posicionY = (int)Math.random() * 100 + 1;
    }

    public Punto(int posicionX, int posicionY) {
        this.posicionX = posicionX;
        this.posicionY = posicionY;
    }

    @Override
    public String toString() {
        return "Punto [posicionX=" + posicionX + ", posicionY=" + posicionY + "]";
    }

    public int getPosicionX() {
        return posicionX;
    }
    public void setPosicionX(int posicionX) {
        this.posicionX = posicionX;
    }
    public int getPosicionY() {
        return posicionY;
    }
    public void setPosicionY(int posicionY) {
        this.posicionY = posicionY;
    }
}
```

Composición. Preservación de la ocultación

Como ya viste en la unidad dedicada a la creación de clases, lo normal suele ser declarar los **atributos** como **privados** (o **protegidos**, como veremos un poco más adelante) para ocultarlos a los posibles **clientes** de la clase (otros objetos que en el futuro harán uso de la clase).

Para que otros objetos puedan acceder a la información contenida en los **atributos,** o al menos a una parte de ella, deberán hacerlo a través de **métodos que sirvan de interfaz**, de manera que sólo se podrá tener acceso a aquella información que el creador de la clase haya considerado oportuna.

Del mismo modo, los **atributos** solamente serán modificados desde los métodos de la clase, que decidirán cómo y bajo qué circunstancias deben realizarse esas modificaciones. Con esa metodología de acceso se tenía perfectamente separada la parte de manipulación interna de los atributos de la **interfaz** con el exterior.

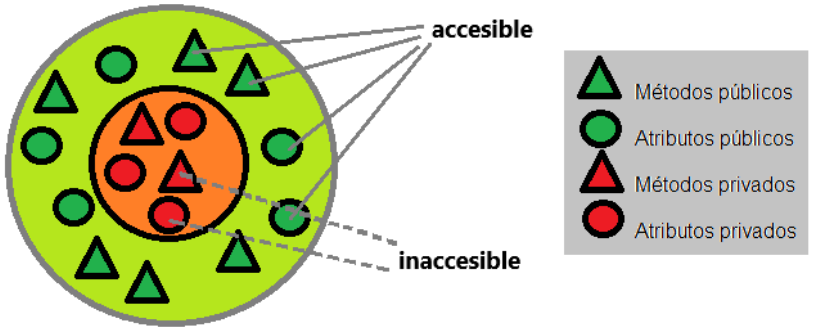
Composición. Preservación de la ocultación

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

Composición. Preservación de la ocultación

Ahora bien, cuando vayas a devolver un objeto habrás de obrar con mucha precaución. Si en un método de la clase devuelves directamente un objeto que es un atributo, estarás ofreciendo directamente una **referencia** a un objeto atributo que probablemente has definido como privado.

¡De esta forma estás **volviendo a hacer público un atributo que inicialmente era privado!**




```
Rectangulo rectangulo = new Rectangulo();
Punto p1 = rectangulo.getVertice1();
p1.setPosicionX(54);
p1.setPosicionY(12.5);
```

Composición. Preservación de la ocultación

```java
package Clases;

public class Rectangulo {

    private Punto vertice1;
    private Punto vertice2;

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

    public Punto getVertice1() {
        return vertice1;
    }

    public void setVertice1(Punto vertice1) {
        this.vertice1 = vertice1;
    }

    public Punto getVertice2() {
        return vertice2;
    }

    public void setVertice2(Punto vertice2) {
        this.vertice2 = vertice2;
    }

    @Override
    public String toString() {
        return "Rectangulo [vertice1=" + vertice1 + ", vertice2=" + vertice2 + "]";
    }
}
```

Composición. Preservación de la ocultación

```
8⊖    public static void main(String[] args) {
9         // TODO Auto-generated method stub
10
11        Rectangulo rectangulo = new Rectangulo();
12        System.out.println("Valores de rectángulo: ");
13        System.out.println(rectangulo);
14
15        System.out.println("Modificamos los valores de rectángulo: ");
16        Punto p1 = rectangulo.getVertice1();
17        p1.setPosicionX(54);
18        p1.setPosicionY(12);
19
20        System.out.println(rectangulo);
21
```

```
Valores de rectángulo:
Rectangulo [vertice1=Punto [posicionX=58, posicionY=3], vertice2=Punto [posicionX=86, posicionY=40]]
Modificamos los valores de rectángulo:
Rectangulo [vertice1=Punto [posicionX=54, posicionY=12], vertice2=Punto [posicionX=86, posicionY=40]]
```

Composición. Preservación de la ocultación

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

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

Composición. Preservación de la ocultación

```java
package Clases;

public class Punto {

    private double posicionX;
    private double posicionY;

    public Punto() {
        this.posicionX = Math.random() * 100;
        this.posicionY = Math.random() * 100;
    }

    public Punto(double posicionX, double posicionY) {
        this.posicionX = posicionX;
        this.posicionY = posicionY;
    }

    public Punto(Punto punto) {
        this.posicionX = punto.posicionX;
        this.posicionY = punto.posicionY;
    }

    @Override
    public String toString() {
        return "Punto [posicionX=" + posicionX + ", posicionY=" + posicionY + "]";
    }

    public double getPosicionX() {
        return posicionX;
    }
    public void setPosicionX(double posicionX) {
        this.posicionX = posicionX;
    }
    public double getPosicionY() {
        return posicionY;
    }
    public void setPosicionY(double posicionY) {
        this.posicionY = posicionY;
    }
}
```

Composición. Preservación de la ocultación

```
3  public class Rectangulo {
4
5      private Punto vertice1;
6      private Punto vertice2;
7
8      public Rectangulo() {
9          this.vertice1 = new Punto();
10         this.vertice2 = new Punto();
11     }
12
13     public Punto getVertice1() {
14         return new Punto(this.vertice1);
15     }
16
17     public void setVertice1(Punto vertice1) {
18         this.vertice1 = vertice1;
19     }
20
21     public Punto getVertice2() {
22         return new Punto(this.vertice2);
23     }
24
25     public void setVertice2(Punto vertice2) {
26         this.vertice2 = vertice2;
27     }
28
29     @Override
30     public String toString() {
31         return "Rectangulo [vertice1=" + vertice1 + ", vertice2=" + vertice2 + "]";
32     }
33 }
```

Composición. Preservación de la ocultación

```
8⊖    public static void main(String[] args) {
9         // TODO Auto-generated method stub
10
11        Rectangulo rectangulo = new Rectangulo();
12        System.out.println("Valores de rectángulo: ");
13        System.out.println(rectangulo);
14
15        System.out.println("Modificamos los valores de rectángulo: ");
16        Punto p1 = rectangulo.getVertice1();
17        p1.setPosicionX(54);
18        p1.setPosicionY(12);
19
20        System.out.println(rectangulo);
21
```

```
Valores de rectángulo:
Rectangulo [vertice1=Punto [posicionX=85, posicionY=59], vertice2=Punto [posicionX=75, posicionY=8]]
Modificamos los valores de rectángulo:
Rectangulo [vertice1=Punto [posicionX=85, posicionY=59], vertice2=Punto [posicionX=75, posicionY=8]]
```

Composición. Preservación de la ocultación

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

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

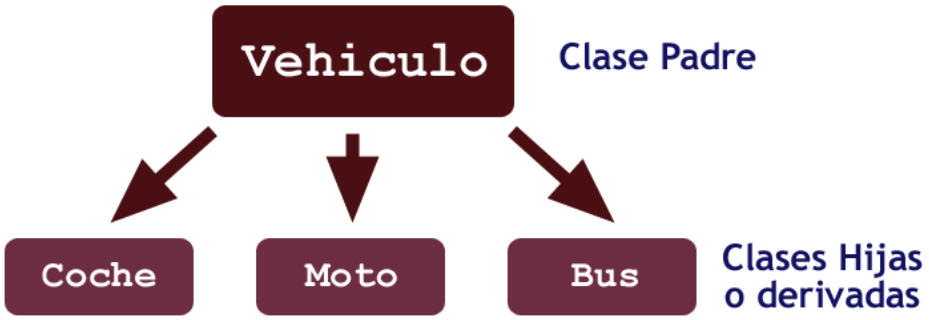
Índice

Tema 5

Herencia

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

El concepto de **herencia** es algo bastante simple y sin embargo muy potente: cuando se desea definir una nueva clase y ya existen clases que, de alguna manera, implementan parte de la funcionalidad que se necesita, es posible crear una nueva **clase derivada** de la que ya tienes. Al hacer esto se posibilita la reutilización de todos los atributos y métodos de la clase que se ha utilizado como **base** (**clase padre** o **superclase**), sin la necesidad de tener que escribirlos de nuevo.



Herencia

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

Algunos ejemplos de herencia podrían ser:

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

Herencia

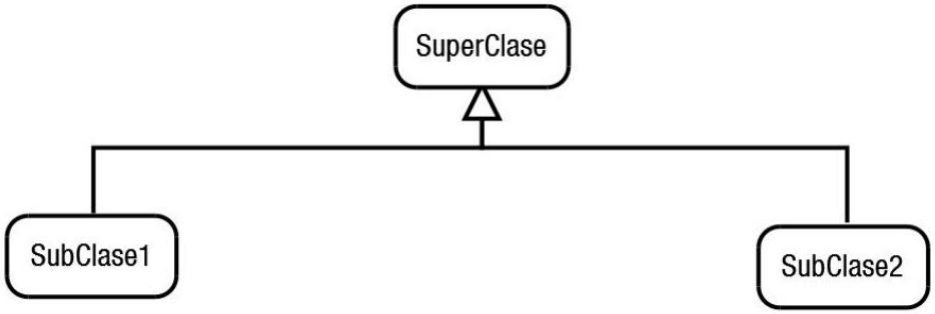
En este caso la expresión idiomática que puedes usar para plantearte si el tipo de relación entre dos clases A y B es de herencia podría ser "**es un**": "la clase A es un tipo específico de la clase B" (**especialización**), o visto de otro modo: "la clase B es un caso general de la clase A" (**generalización**).

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

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

Herencia

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



La clase de la que se hereda suele ser llamada clase **base**, clase **padre** o **superclase**. A la clase que hereda se le suele llamar clase hija, clase derivada o subclase.

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

Herencia

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

Herencia

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

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

Herencia. Sintaxis

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

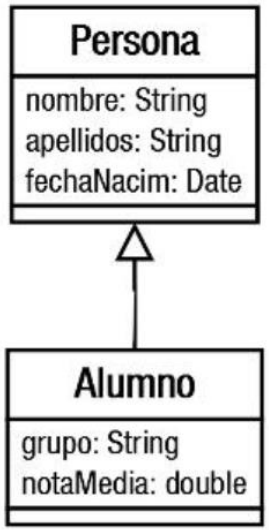
```
[modificador] class ClasePadre {
    // Cuerpo de la clase
    …
}

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

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

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

Herencia. Sintaxis

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




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

}
```

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

Herencia. Acceso a miembros heredados

Como ya has visto anteriormente, no es posible acceder a miembros **privados** de una superclase. Para poder acceder a ellos podrías pensar en hacerlos **públicos**, pero entonces estarías dando la opción de acceder a ellos a cualquier objeto externo y es probable que tampoco sea eso lo deseable.

Para ello se inventó el modificador **protected** (**protegido**) que permite el **acceso desde clases heredadas**, pero no desde fuera de las clases (estrictamente hablando, desde fuera del paquete), que serían como miembros **privados**.

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

En la diapositiva siguiente tienes, de nuevo, el resumen:

Herencia. Acceso a miembros heredados

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	

Herencia. Acceso a miembros heredados

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

```
public class Persona {
    private String nombre;
    private String apellidos;
    …
}
```

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

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

Herencia. Acceso a miembros heredados

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

Herencia. Atributos y miembros heredados

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

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

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

Herencia. Atributos y miembros heredados

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

- Seis por ser **Persona** (**getNombre**, **getApellidos**, **getFechaNacim**, **setNombre**, **setApellidos**, **setFechaNacim**).

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

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

Ejercicio

En el método Main, declara un objeto de la clase derivada y accede a alguno de sus métodos.

Además intenta acceder a métodos de la clase base.

Herencia. Redefinición de métodos heredados

Una clase puede **redefinir** algunos de los métodos que ha heredado de su **clase base**.

En tal caso, el nuevo método (**especializado**) sustituye al **heredado**. Este procedimiento también es conocido como de **sobrescritura de métodos**.

En cualquier caso, aunque un método sea **sobrescrito** o **redefinido**, aún es posible acceder a él a través de la referencia **super**, aunque sólo se podrá acceder a métodos de la **clase padre** y no a métodos de clases superiores en la **jerarquía de herencia**.

Herencia. Redefinición de métodos heredados

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

```
public String getApellidos () {
    return "Alumno: " + apellidos;
}
```

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

En cualquier caso, no es necesario indicar **@Override**, pero puede resultar de ayuda a la hora de localizar este tipo de errores (crees que has sobrescrito un **método heredado** y al confundirte en una letra estás realmente creando un nuevo método diferente). En el caso del ejemplo anterior quedaría:

```
@Override
public String getApellidos ()
```

Ejercicio

Redefinir el método getApellidos en la clase hija y llamarlo desde el método Main.

Declarar el método getApellidos en la clase hija como Override y llamarlo desde el método Main.

Al método getApellidos declarado como Override, cambiarle el nombre a getApellido → ¿funciona?

Herencia. Ampliación de métodos heredados

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

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

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

Herencia. Ampliación de métodos heredados

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

Podría hacerse de una manera tan sencilla como la siguiente:

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

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

Ejercicio

Definir el anterior método mostrar() tal y como está en la transparencia anterior.

Llamar al método desde el método Main.

Herencia. Constructores y herencia

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

Como ya has visto, un **constructor** de una **clase derivada** puede hacer algo parecido para llamar al **constructor** de su **clase base** mediante el uso de la palabra **super**. De esta manera, el **constructor** de una **clase derivada** puede llamar primero al **constructor** de su **superclase** para que inicialice los **atributos heredados** y posteriormente se inicializarán los **atributos específicos** de la clase: los no heredados. Nuevamente, esta llamada también **debe ser la primera sentencia de un constructor** (con la única excepción de que exista una llamada a otro constructor de la clase mediante **this**).

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

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

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

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

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

```
public Alumno (String nombre, String apellidos, Fecha fechaNacim,
                        String grupo, double notaMedia) {
      super (nombre, apellidos, fechaNacim);
      this.grupo= grupo;
      this.notaMedia= notaMedia;
   }
```

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

Ejercicio

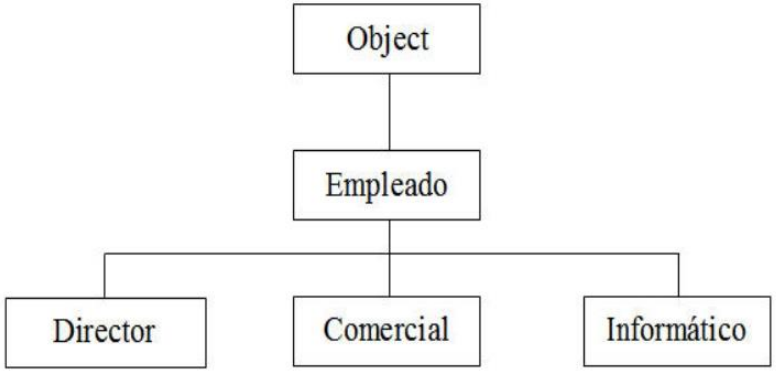
Declarar los anteriores constructores y hacer las pruebas necesarias desde el main para probarlos.

Herencia. La clase Object en Java

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

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

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



Herencia. La clase Object en Java

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

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

Ejercicio

Extender la clase Persona a la clase Object.

Pulsando la tecla control y pulsando con el ratón en la clase Object nos abrirá la clase padre de la que heredan todos los objetos.

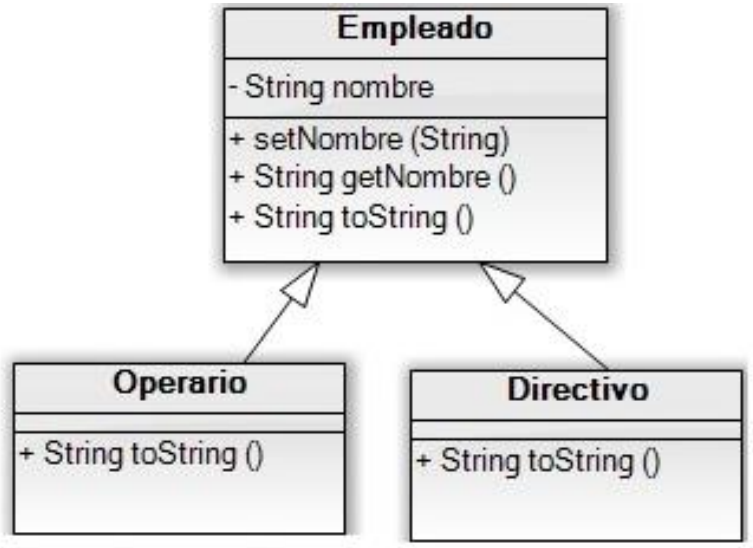
Buscar el método toString() → ¿Os suena la salida?

Ejercicio

Implementar el siguiente diagrama.

La clase empleado cuenta sólo con el atributo nombre. El método toString muestra: “Soy el empleado Pepe”

Las clases hijas únicamente sobreescriben el método toString y llaman al toString de su clase Padre. Además muestran el texto: “Soy el operacio Pepito” y “Soy la directiva Carlota”



Índice

Tema 5

Clases Abstractas

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

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

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

Clases Abstractas

La posibilidad de **declarar clases abstractas es una de las características más útiles de los lenguajes orientados a objetos**, pues permiten dar unas líneas generales de cómo es una clase sin tener que implementar todos sus métodos o implementando solamente algunos de ellos.

Esto resulta especialmente útil cuando las distintas **clases derivadas** deban proporcionar los mismos métodos indicados en la clase **base abstracta**, pero su i**mplementación sea específica** para cada **subclase**.

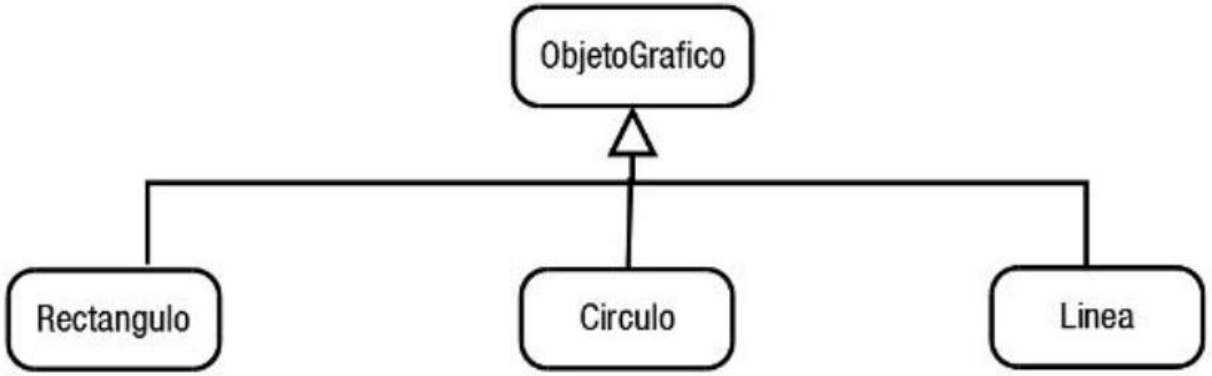
Clases Abstractas

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).

Clases Abstractas

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.



Clases Abstractas. Declaración

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

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

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

Clases Abstractas. Declaración

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

Clases Abstractas. Declaración

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

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

- Una **clase abstracta** sólo puede usarse para crear nuevas clases derivadas. No se puede hacer un **new** de una **clase abstracta**. Se produciría un **error de compilación**.

- Una **clase abstracta** puede contener **métodos totalmente definidos** (**no abstractos**) y **métodos sin definir** (**métodos abstractos**).

Clases Abstractas. Métodos abstractos

Un **método abstracto** es un método cuya implementación no se define, sino que se declara únicamente su **interfaz** (cabecera) para que su cuerpo sea implementado más adelante en una **clase derivada**.

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

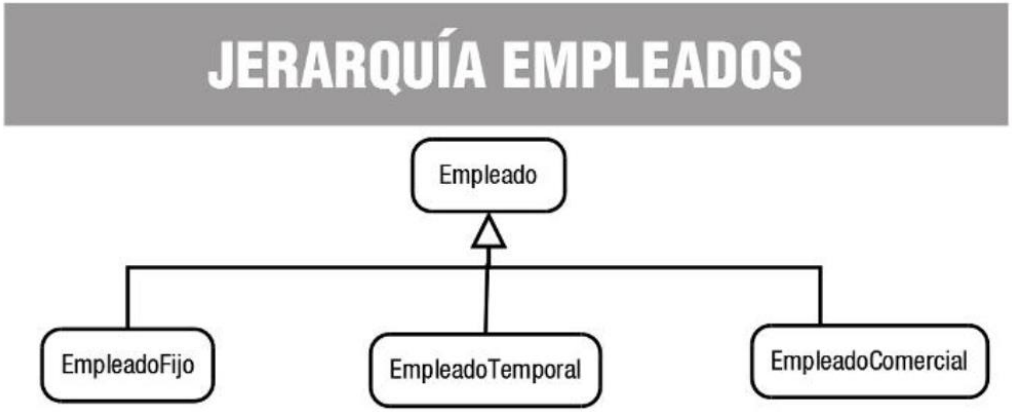
```
[modificador_acceso] abstract <tipo> <nombreMetodo> ([parámetros]) [excepciones];
```

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

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

Clases Abstractas. Métodos abstractos

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



Clases Abstractas. Métodos abstractos

Debes tener en cuenta al trabajar con métodos abstractos:

- Un **método abstracto** implica que la clase a la que pertenece tiene que ser **abstracta**, pero eso no significa que todos los métodos de esa clase tengan que ser abstractos.

- Un **método abstracto** no puede ser **privado** (no se podría implementar, dado que las **clases derivadas** no tendrían acceso a él).

- Los **métodos abstractos** no pueden ser **estáticos**, pues los **métodos estáticos** no pueden ser redefinidos (y los **métodos abstractos** necesitan ser redefinidos).

Clases Abstractas. Clases y métodos finales

En unidades anteriores has visto el modificador **final**, aunque sólo lo has utilizado por ahora para **atributos** y **variables** (por ejemplo para declarar **atributos constantes**, que una vez que toman un valor ya no pueden ser modificados).

Pero este modificador también puede ser utilizado con clases y con métodos (con un comportamiento que no es exactamente igual, aunque puede encontrarse cierta analogía: no se permite heredar o no se permite redefinir).

Una clase declarada como final no puede ser heredada, es decir, **no puede tener clases derivadas**. La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas):

```
[modificador_acceso] final class nombreClase [herencia] [interfaces]
```

Un **método** también puede ser declarado como **final**, en tal caso, ese método no podrá ser redefinido en una **clase derivada**:

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

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

Ejercicio

Definir la siguientes clases: **Figura**, **Cuadrado** y **Triangulo**).

- En la clase **Figura** se ha definido un atributo color, un constructor y dos métodos, calcularArea y getColor.
- En la clase **Cuadrado** se ha definido un atributo lado, un constructor y un método calcularArea.
- En la clase **Triangulo** se han definido dos atributos base y altura, un constructor y un método calcularArea.
- Definir el método calcularArea abstracto abstract en la superclase abstracta Figura, indicándose solamente su signatura.
- En cada una de las subclases, Cuadrado y Triangulo, se implementar dicho método.

Realizar una clase TestFiguras, donde se prueben todos los métodos de cada una de las clases.

Índice

Tema 5

Polimorfismo

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

La **encapsulación** te ha permitido agrupar **características** (**atributos**) y **comportamientos** (**métodos**) dentro de una misma unidad (**clase**), pudiendo darles un mayor o menor componente de **visibilidad**, y permitiendo separar al máximo posible la **interfaz** de la **implementación**.

Polimorfismo

Por otro lado la **herencia** te ha proporcionado la posibilidad de tratar a los objetos como pertenecientes a una **jerarquía de clases**. Esta capacidad va a ser fundamental a la hora de poder manipular muchos posibles objetos de clases diferentes como si fueran de la misma clase (**polimorfismo**).

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

Polimorfismo

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

Un método "**polimórfico**" ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en **tiempo de ejecución** en lugar de en **tiempo de compilación**.

Polimorfismo

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

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

Polimorfismo

El **polimorfismo** ofrece la **posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una de sus subclases**. Esto te va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus **superclases**.

El **polimorfismo** puede llevarse a cabo tanto con **superclases** (abstractas o no) como con **interfaces**.

Dada una **superclase X**, con un método **m**, y dos **subclases A** y **B**, que redefinen ese método **m**, podrías declarar un objeto **O** de tipo **X** que durante la **ejecución** podrá ser de tipo **A** o de tipo **B** (algo desconocido en **tiempo de compilación**).

Esto significa que al invocarse el método **m** de **X** (**superclase**), se estará en realidad invocando al método **m** de **A** o de **B** (alguna de sus **subclases**).

Polimorfismo

Por ejemplo:

```
// Declaración de una referencia a un objeto de tipo X
ClaseX obj; // Objeto de tipo X (superclase)
...

// Zona del programa donde se instancia un objeto de tipo A (subclase) y se le asigna a
// la referencia obj.
// La variable obj adquiere la forma de la subclase A.
obj = ClaseA ();
...

// Otra zona del programa.
// Aquí se instancia un objeto de tipo B (subclase) y se le asigna a la referencia obj.
// La variable obj adquiere la forma de la subclase B.
obj = ClaseB ();
...

// Zona donde se utiliza el método m sin saber realmente qué subclase se está utilizando.
// (Sólo se sabrá durante la ejecución del programa)
obj.m () // Llamada al método m (sin saber si será el método m de A o de B).
...
```

Polimorfismo

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**.

Polimorfismo

Ejercicio

- Crear la clase **Persona**.
- Agrega la clase **Alumno** y **Profesor** que hereden de la clase Persona (abstracta)
- En la clase Persona definir un método abstracto queSoy()
- Redefinir el método en las clases hijas.
- En el caso de la clase Alumno ese método debe devolver “Soy un alumno”.
- En el caso de la clase Profesor ese método debe devolver “Soy un profesor”.

- Desde la clase Main, probar de la siguiente manera:

```
Persona p = new Profesor();
System.out.println(p.queSoy());
p = new Alumno();
System.out.println(p.queSoy());
```

```
Soy un profesor
Soy un alumno
```

Polimorfismo

Ejercicio

Se va a implementar un simulador de Vehículos. Existen dos tipos de Vehículo: Coche y Camión.

- Sus características comunes son la matrícula y la velocidad. En el momento de crearlos, la matrícula se recibe por parámetro y la velocidad se inicializa a 0. El método toString() de los vehículos devuelve información acerca de la matrícula y la velocidad. Además se pueden acelerar, pasando por parámetro la cantidad en km/h que se tiene que acelerar.

- Los coches tienen además un atributo para el número de puertas, que se recibe también por parámetro en el momento de crearlo. Tiene además un método que devuelve el número de puertas.

- Los camiones tienen un atributo de tipo Remolque que inicializa a null (para indicar que no tiene un remolque). Además, tiene un método ponRemolque(), que recibe el Remolque por parámetro, y otro quitaRemolque(). Cuando se muestre la información de un camión que lleve remolque, además de la matrícula y velocidad del camión, debe aparecer la información del remolque.
- En esta clase, hay que sobreescribir el método acelerar de manera que si el camión tiene remolque y la velocidad más la aceleración superan los 100 km/h, no suba mas la velocidad.

- Hay que implementar la clase Remolque. Esta clase tiene un atributo de tipo entero que es el peso y cuyo valor se le da en el momento de crear el objeto. Debe tener un método toString() que devuelva la información del remolque.

- Hacer uso de **herencia** y **polimorfismo** para realizar las pruebas en el método main TestVehiculos.

Índice

Tema 5

Interfaces

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

Estos métodos sin implementar indican un **comportamiento**, un tipo de conducta, aunque no especifican cómo será ese **comportamiento** (**implementación**), pues eso dependerá de las características específicas de cada clase que decida implementar esa **interfaz**.

Podría decirse que una **interfaz** se encarga de establecer qué **comportamientos** hay que tener (qué **métodos**), pero no dice nada de cómo deben llevarse a cabo esos **comportamientos** (**implementación**). Se indica sólo la **forma**, no la **implementación**.

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

Interfaces

En conclusión: **una interfaz se encarga de establecer unas líneas generales sobre los comportamientos (métodos) que deberían tener los objetos de toda clase que implemente esa interfaz, es decir, que no indican lo que el objeto es** (de eso se encarga la clase y sus **superclases**)**, sino acciones (capacidades) que el objeto debería ser capaz de realizar**.

Es por esto que el nombre de muchas interfaces en Java termina con sufijos del tipo "**-able**", "**-or**", "**-ente**" y cosas del estilo, que significan algo así como **capacidad o habilidad** para hacer o ser receptores de algo (**configurable**, **serializable**, **modificable**, **clonable**, **ejecutable**, **administrador**, **servidor, buscador**, etc.), dando así la idea de que se tiene la capacidad de llevar a cabo el conjunto de acciones especificadas en la **interfaz**.

Interfaces

Imagínate por ejemplo la clase **Coche**, **subclase** de **Vehículo**. Los coches son **vehículos a motor**, lo cual implica una serie de acciones como, por ejemplo, **arrancar el motor** o **detener el motor**. Esa acción no la puedes heredar de **Vehículo**, pues no todos los vehículos tienen porqué ser a motor (piensa por ejemplo en una clase **Bicicleta**), y no puedes heredar de otra clase pues ya heredas de **Vehículo**. Una solución podría ser crear una **interfaz Arrancable**, que proporcione los métodos típicos de un **objeto a motor** (no necesariamente vehículos).

De este modo la clase **Coche** sigue siendo subclase de **Vehículo**, pero también implementaría los comportamientos de la interfaz **Arrancable**, los cuales podrían ser también implementados por otras clases, hereden o no de **Vehículo** (por ejemplo una clase **Motocicleta** o bien una clase **Motosierra**). La clase **Coche** implementará su método **arrancar** de una manera, la clase **Motocicleta** lo hará de otra (aunque bastante parecida) y la clase **Motosierra** de otra forma probablemente muy diferente, pero todos tendrán su propia versión del método **arrancar** como parte de la interfaz **Arrancable**.

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

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

Interfaces. Definición

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

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

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

Interfaces. Definición

Su sintaxis quedaría entonces:

```
[public] interface <NombreInterfaz> {
    [public] [final] <tipo1> <atributo1>= <valor1>;
    [public] [final] <tipo2> <atributo2>= <valor2>;
    ...
    [public] [abstract] <tipo_devuelto1> <nombreMetodo1> ([lista_parámetros]);
    [public] [abstract] <tipo_devuelto2> <nombreMetodo2> ([lista_parámetros]);
    ...
                                        }
```

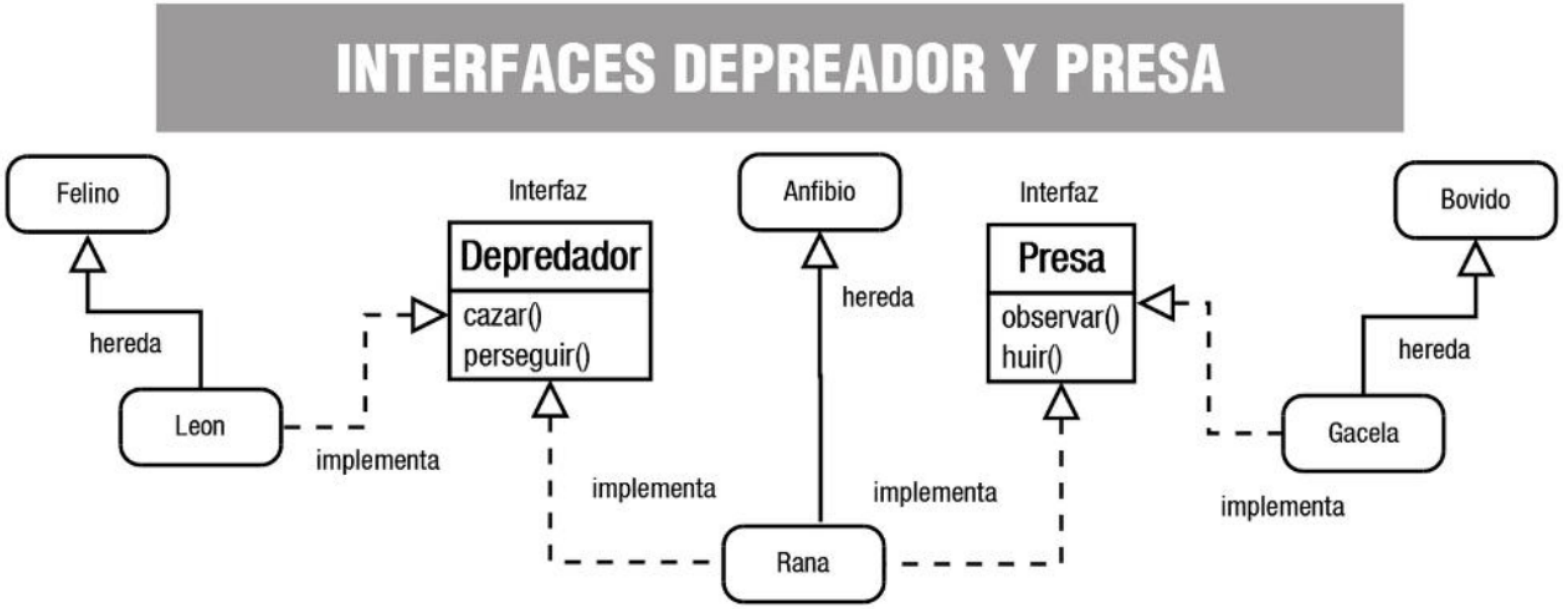
Si te fijas, la declaración de los métodos termina en punto y coma, pues no tienen cuerpo, al igual que sucede con los **métodos abstractos** de las **clases abstractas**.

El ejemplo de la interfaz **Depredador** que hemos visto antes podría quedar entonces así:

```
public interface Depredador {
    void perseguir (Animal presa);
    void cazar (Animal presa);
    ...
}
```

Interfaces. Definición

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



Interfaces. Implementación

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

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

```
class NombreClase implements NombreInterfaz {
```

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

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

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

Interfaces. Implementación

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

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

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

```
class Leon implements Depredador {
```

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

```
void perseguir(Animal presa) {
    // Implementación del método perseguir para un león
    ...
}
```

Interfaces. Implementación

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

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

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

Interfaces. ¿Clase abstracta o interfaz?

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

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

- **Una clase no puede heredar de varias clases**, aunque sean abstractas (**herencia múltiple**). Sin embargo sí puede **implementar una o varias interfaces** y además seguir heredando de una clase.
- **Una interfaz no puede definir métodos** (**no implementa su contenido**), tan solo los declara o enumera.
- **Una interfaz puede hacer que dos clases tengan un mismo comportamiento** independientemente de sus ubicaciones en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).

Interfaces. ¿Clase abstracta o interfaz?

- **Una interfaz permite establecer un comportamiento de clase sin apenas dar detalles**, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la **interfaz**).

- **Las interfaces tienen su propia jerarquía**, diferente e independiente de la jerarquía de clases.

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

1) Volver a escribirla para esa jerarquía de clases. Lo cual no parece una buena solución.

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

Interfaces. Simulación de herencia múltiple mediante interfaces

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

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

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

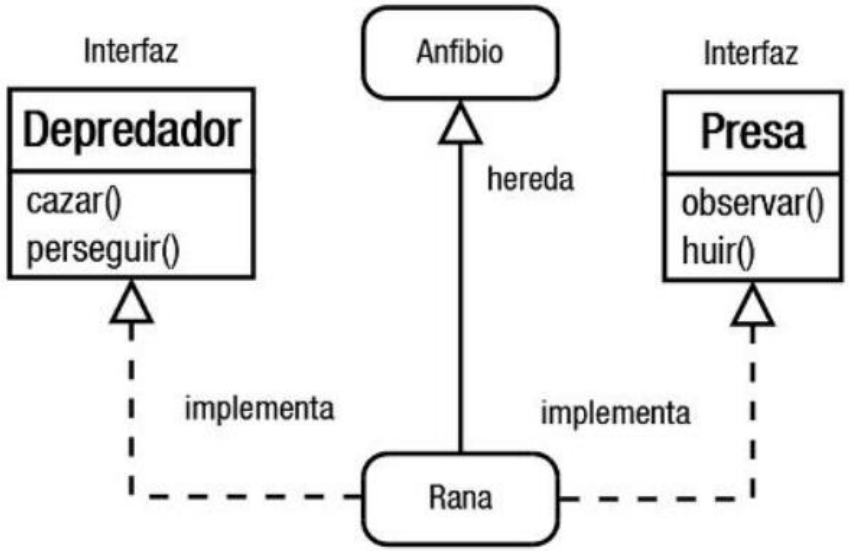
De esta manera la clase **X** podría a la vez:

1) Implementar las interfaces **A**, **B**, **C**, que la dotarían de los comportamientos que deseaba heredar de las clases **A**, **B**, **C**.

2) Heredar de otra clase **Y**, que le proporcionaría determinadas características dentro de su taxonomía o jerarquía de objeto (atributos, métodos implementados y métodos abstractos).

Interfaces. Simulación de herencia múltiple mediante interfaces

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



Interfaces. Simulación de herencia múltiple mediante interfaces

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

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

- Si los dos métodos tienen **diferentes parámetros** no habrá problema aunque tengan el mismo nombre pues se realiza una **sobrecarga** de métodos.

- Si los dos métodos tienen **un valor de retorno de un tipo diferente**, se producirá un **error de compilación** (al igual que sucede en la sobrecarga cuando la única diferencia entre dos métodos es ésa).

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

Interfaces. Herencia de interfaces

Las **interfaces**, al igual que las **clases**, también permiten la **herencia**. Para indicar que una **interfaz** hereda de otra se indica nuevamente con la palabra reservada **extends**. Pero en este caso sí se permite la **herencia múltiple de interfaces**. Si se hereda de más de una **interfaz** se indica con la lista de **interfaces** separadas por comas.

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

```
public interface InterfazUno {
    // Métodos y constantes de la interfaz Uno
}
public interface InterfazDos {
    // Métodos y constantes de la interfaz Dos
}
```

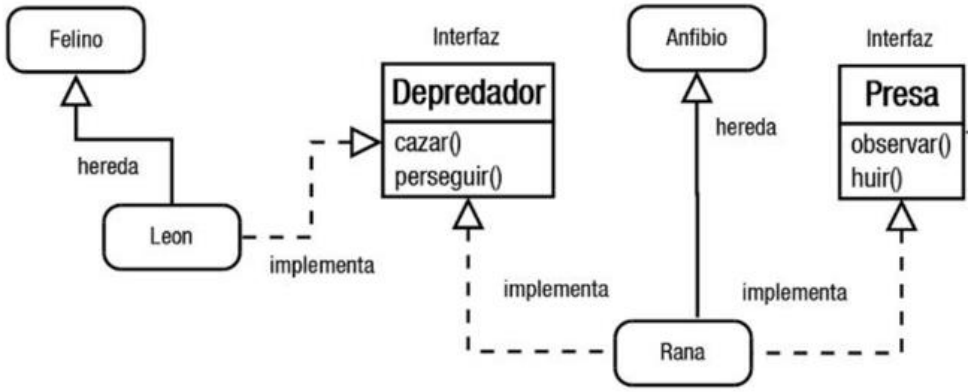
Podría definirse una nueva **interfaz** que heredara de ambas:

```
public interface InterfazCompleja extends InterfazUno, InterfazDos {
    // Métodos y constantes de la interfaz compleja
}
```

Ejercicio

Implementar en Java el siguiente diagrama. Donde:

- La clase Anfibio, tiene nombre, número de patas y temperatura corporal.
- La clase Rana, tiene un tipo únicamente, con los valores: campestre, norteña, ibérica y bermeja.
- Los felinos se caracterizan por tener un número de dientes y un hábitat natural (selva o bosque).
- El león al igual que la rana, tiene un tipo: Congo, asiático, atlas y Katanga.
- Los métodos de las interfaces únicamente sacaran un mensaje por pantalla indicando la acción que están realizando.
- Realizar una clase TestAnimales y declarar varios objetos de todas las clases posibles del diagrama y para cada una de ellas probar las llamadas a todos los métodos de cada uno de los objetos, tanto procedentes de la herencia como de los interfaces.



FIN


Kahoot!