

# **UT7 Orientación a Objetos Avanzado**

## 7.0. Introducción

### 7.0.1 Conceptos

**a) Encapsulamiento**

Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama ***encapsulamiento***. Encapsulamiento es el ocultamiento del estado, es decir, de los datos miembro de un objeto, de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto. El encapsulamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma, el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse solo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

El encapsulamiento de variables y métodos en un componente de software provee de dos principales ***beneficios*** a los desarrolladores de software:

* **Modularidad**. Esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos.

La *modularidad* es la capacidad que tiene un [sistema](http://es.wikipedia.org/wiki/Sistema" \t "Sistema) de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo. Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de [módulo](http://es.wikipedia.org/wiki/Módulo_(programación)" \t "Módulo (programación)).

* **Ocultamiento de la información.** Es decir, un objeto tiene una "interfaz publica" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier momento sin afectar a los demás objetos que dependan de ello.

Los **objetos** tienen la ventaja de la modularidad y el ocultamiento de la información. Las **clases** proveen el beneficio de la **reutilización**. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

En las implantaciones orientadas a objetos se percibe un objeto como un paquete de datos y procedimientos que se pueden llevar a cabo con estos datos. Esto encapsula los datos y los procedimientos.

Los **atributos** se relacionan al objeto o instancia y los métodos a la clase. ¿Por qué se hace así? Los atributos son variables comunes en cada objeto de una clase y cada uno de ellos puede tener un valor para cada variable, diferente al que tienen para esa misma variable los demás objetos.

Los **métodos**, por su parte, pertenecen a la clase y no se almacenan en cada objeto, puesto que sería un desperdicio almacenar el mismo procedimiento varias veces y ello va contra el principio de reutilización de código.

**b) Herencia**

La *herencia* es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

**La herencia está fuertemente ligada a la reutilización del código en la POO.**

En Java, como en otros lenguajes de programación orientados a objetos, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama **subclase**. La clase de la que está derivada se denomina **superclase**.

De hecho en Java, todas las clases deben derivar de alguna clase. Lo que nos lleva a la cuestión ¿Dónde empieza todo esto? La clase más alta, la clase de la que todas las demás descienden, es la clase **Object**, definida en java.lang. Object es la raíz de la herencia de todas las clases.

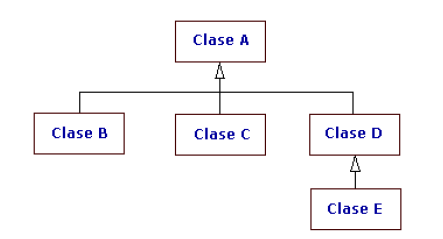
Hay dos **tipos de herencia**: Herencia Simple y Herencia Múltiple. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. **Java solo permite herencia simple**.

**c) Superclase y Subclases**

El concepto de herencia en Java conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la POO todas las relaciones entre clases deben ajustarse a dicha estructura.

**En esta estructura jerárquica, cada clase tiene sólo una clase padre**. La clase padre de cualquier clase es conocida como su **superclase**. La clase hija de una superclase es llamada una **subclase**.

* Una superclase puede tener cualquier número de subclases.
* Una subclase puede tener solo una superclase.



* A es la superclase de B, C y D.
* D es la superclase de E.
* B, C y D son subclases de A.
* E es una subclase de D.

**d) Polimorfismo**

Otro concepto de la POO es el polimorfismo. Un objeto solamente tiene una forma (la que se le asigna cuando se construye ese objeto) pero la referencia al objeto es polimórfica porque puede referirse a objetos de diferentes clases (es decir, la referencia toma múltiples formas). Para que esto sea posible **debe haber una relación de herencia entre esas clases**. Por ejemplo, considerando la figura anterior de herencia se tiene que:

* Una referencia a un objeto de la clase B también puede ser una referencia a un objeto de la clase A.
* Una referencia a un objeto de la clase C también puede ser una referencia a un objeto de la clase A.
* Una referencia a un objeto de la clase D también puede ser una referencia a un objeto de la clase A.
* Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase D.
* Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase A.

**e) Abstracción**

Volviendo a la figura anterior de relación de herencia entre clases, se puede pensar en una jerarquía de clases como la definición de conceptos **más abstractos en lo alto de la jerarquía** y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; **pueden agregar variables y métodos** además de los que ya heredan de sus clases padres. Las clases hijas pueden también, **sobrescribir los métodos** que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un solo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y cuantos más niveles se desciende en una clase, más especializada será su conducta.

La Herencia presenta los siguientes **beneficios**:

Las subclases proveen conductas especializadas sobre la base de elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden *reutilizar el código de la superclase* muchas veces.

Los programadores pueden implementar superclases llamadas **clases abstractas** que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente la conducta, pero parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

### 7.0.2 Heredando clases en Java

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la POO todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene solo una clase padre. La clase padre de cualquier clase es conocida como su superclase. La clase hija de una superclase es llamada una subclase.

De manera automática, una subclase hereda los atributos y métodos de su superclase.

Además, una subclase puede agregar nueva funcionalidad (atributos y métodos) que la

superclase no tenía.

**Los constructores no son heredados por las subclases.**

Para crear una subclase, se incluye la palabra clave **extends** en la declaración de la clase.

|  |
| --- |
| class nombreSubclase **extends** nombreSuperclase {  ................  } |

En Java, la clase padre de todas las clases es la clase **Object** y cuando una clase no tiene una superclase explícita, su superclase es Object.

### 7.0.3 Sobrecarga de métodos y de constructores

Cada método tiene una **"firma"** por así decirlo, que son su nombre, el tipo y el número de sus parámetros. Existe una característica para tener dos métodos (o constructores) con el mismo nombre. Esta característica se denomina **sobrecarga de métodos**.

La **cabecera de un método** es la combinación del tipo de dato que devuelve, su nombre y su lista de argumentos.

Java diferencia los métodos sobrecargados en base al **número y tipo de argumentos** que tiene el método y **no por el tipo que devuelve**.

También existe la **sobrecarga de constructores**. Cuando en una clase existen varios constructores, se dice que hay sobrecarga de constructores.

Veamos un ejemplo en el caso de una clase llamada Publicacion y cómo los constructores nos dan un ejemplo de sobrecarga de métodos:



El compilador resolverá qué constructor debe ejecutar en cada momento en función del número de parámetros y su tipo. Si se llama al constructor sin parámetros se ejecutará el primer constructor y en caso de hacerlo con dos parámetros String, se ejecutará el segundo.

|  |
| --- |
| **Nota:** El concepto de **sobrecarga de métodos** se puede aplicar siempre que los parámetros sean diferentes, bien por su tipo, bien porque el número de parámetros de un método u otro es diferente. Hay que tener cuidado con los tipos: int, byte y short ya que aunque son tipos diferentes, si hacemos la llamada a un método con un número entero no sabría a cuál de los métodos llamar, ya que un entero puede ser considerado de las tres formas. Con double y float no pasa, porque acordaos que hemos de forzar a que Java entienda un decimal como float, por defecto lo entiende como double. |

Otros ejemplos:

int calculaSuma(int x, int y, int z){

...

}

int calculaSuma(double x, double y, double z){

...

}

/\* **Error**: los métodos siguientes no están sobrecargados \*/

int calculaSuma(int x, int y, int z){

...

}

double calculaSuma(int x, int y, int z){

...

}

**Ej**:

package usuario;

public class Usuario{

String nombre;

int edad;

String direccion;

//El constructor está sobrecargado

public Usuario(){

nombre = null;

edad = 0;

direccion = null;

}

public Usuario(String nombre, int edad, String direccion){

this.nombre = nombre;

this.edad = edad;

this.direccion = direccion;

}

public Usuario(Usuario usr){

nombre = usr.getNombre();

edad = usr.getEdad();

direccion = usr.getDireccion();

}

public void setNombre(String n){

nombre = n;

}

public String getNombre(){

return nombre;

}

//El metodo setEdad() está sobrecargado

public void setEdad(int e){

edad = e;

}

public void setEdad(double e){

edad = (int)e;

}

public int getEdad(){

return edad;

}

public void setDireccion(String d){

direccion = d;

}

public String getDireccion(){

return direccion;

}

// Añadir un método para imprimir un usuario para eliminarlo

// del main

}

package prueba;

import usuario.Usuario;

public class Main{

void imprimeUsuario(Usuario usr){

System.out.println("\nNombre: " + usr.getNombre());

System.out.println("Edad: " + usr.getEdad());

System.out.println("Direccion: " + usr.getDireccion());

}

public static void main(String args[]){

Main prog = new Main();

//La otra opción sería que imprimeUsuario fuera static

Usuario usr1, usr2;

usr1 = new Usuario();

prog.imprimeUsuario(usr1);

usr2 = new Usuario ("Eduardo",24,"Mi direccion");

prog.imprimeUsuario(usr2);

usr1 = new Usuario(usr2);

usr1.setEdad(50);

usr2.setEdad(30.45);

prog.imprimeUsuario(usr1);

prog.imprimeUsuario(usr2);

}

}

### 7.0.4 Sobreescritura de métodos

Una subclase hereda todos los métodos de su superclase que son accesibles a dicha subclase a menos que la subclase **sobreescriba los métodos**.

Una **subclase sobreescribe un método de su superclase**, cuando define un método con las mismas características (nombre, número y tipo de argumentos y retorno) que el método de la superclase.

Las subclases emplean la sobreescritura de métodos la mayoría de las veces para agregar o modificar la funcionalidad del método heredado de la clase padre.

Ej:

class ClaseA{

void miMetodo(int var1, int var2){ ... }

String miOtroMetodo(){ ... }

}

class ClaseB extends ClaseA{

//Estos métodos sobreescriben a los métodos de la clase padre

void miMetodo(int var1 ,int var2){ ... }

String miOtroMetodo(){ ... }

}

### 7.0.5 Clases abstractas

Una clase que declara la existencia de métodos, pero no la implementación de dichos métodos, se considera una clase abstracta.

Una clase abstracta puede contener métodos no-abstractos.

Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

abstract class Drawing {

abstract void miMetodo(int var1, int var2);

String miOtroMetodo(){ ... }

}

Una clase abstracta no se puede instanciar, no se puede hacer new Drawing, pero sí se puede heredar de ella y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

Ej:

Public abstract class FiguraGeometrica {

. . .

abstract double Area();

. . .

}

class Circulo extends FiguraGeometrica {

double radio;

double Area() {

return 3.14\*radio\*radio;

}

}

Se pueden crear referencias a clases abstractas como cualquier otra. No hay ningún problema en poner:

FiguraGeometrica figura;

Sin embargo, una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta. El compilador producirá un error si se intenta:

FiguraGeometrica figura = new FiguraGeometrica();

Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

Sin embargo, utilizando el **up-casting**\* se puede escribir:

FiguraGeometrica figura = new Circulo(. . .);

figura.Area();

(\*) Es la operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases. Cuando se realiza este tipo de operaciones, hay que tener cuidado porque la referencia es a los miembros de la clase hija, aunque la referencia apunte a un objeto de este tipo.

### 7.0.6 Interfaces (se ampliará para ver novedades -> v8)

Una interface es una variante de una clase abstracta con la condición de que todos sus métodos deben ser abstractos (lo serán aunque no se especifique). Si la interface va a tener atributos, estos deben llevar las palabras reservadas **static final** y con un valor inicial ya que funcionan como constantes.

Ej:

interface Nomina{

public static final String EMPRESA = "Patito, S. A.";

public void detalleDeEmpleado(Nomina obj);

}

Una clase implementa una o más interfaces (separadas con comas ",") con la palabra reservada implements. Con el uso de interfaces se puede "simular" la herencia múltiple que Java no soporta.

class Empleado implements Nomina {

...

}

La clase que implementa una interface tiene dos opciones:

1) Implementar todos los métodos de la interface.

2) Implementar solo algunos de los métodos de la interface pero esa clase entonces debe ser una clase abstracta.

***(Ver ampliación Interfaces)***

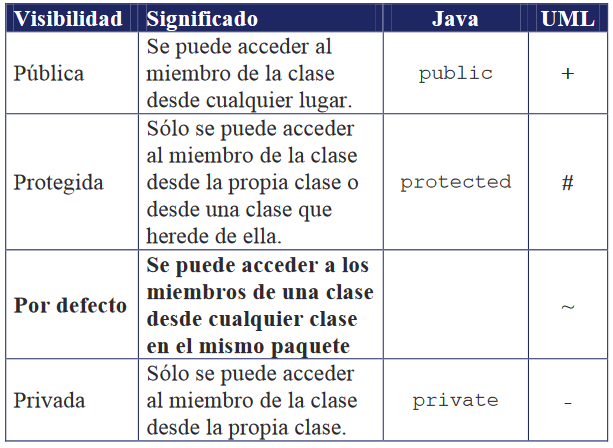
### 7.0.7 Control de acceso a miembros de una clase

Los modificadores más importantes son los que permiten controlar la visibilidad y acceso a los métodos y variables que están dentro de una clase.

Uno de los beneficios de las clases, es que pueden proteger a sus variables y métodos (tanto de instancia como de clase) del acceso desde otras clases.

Java soporta cuatro niveles de acceso a variables y métodos. En orden, del más público al menos público son: público (public), protegido (protected), sin modificador (también conocido como package) y privado (private).

La siguiente tabla muestra el **nivel de acceso permitido por cada modificador**:



Como se observa en la tabla anterior:

* Una clase puede ver/usar dentro de ella misma todo tipo de variables y métodos (desde los public hasta los private);
* Las demás clases del mismo paquete (ya sean subclases o no) tienen acceso a los miembros public, protected y sin modificador.
* El resto del universo de clases (que no sean ni del mismo paquete ni subclases) pueden ver solo los miembros public.

## 7.1. Modificadores de acceso en Java

En su soporte para la **encapsulación**, la clase proporciona dos beneficios principales. *Primero*, vincula datos con el código que lo manipula. En *segundo* lugar, proporciona los medios por los que se puede controlar el acceso a los miembros.

Aunque el enfoque de Java es un poco más sofisticado, en esencia, hay dos tipos básicos de miembros de la clase: público (public) y privado (private). Se puede acceder libremente a un miembro público mediante un código definido fuera de su clase. Se puede acceder a un miembro privado solo por otros métodos definidos por su clase. A través del uso de miembros privados **el acceso está controlado**.

Restringir el acceso a los miembros de una clase es una parte fundamental de la **[programación orientada a objetos](https://javadesdecero.es/poo" \t "_blank)**, ya que ayuda a evitar el mal uso de un objeto.

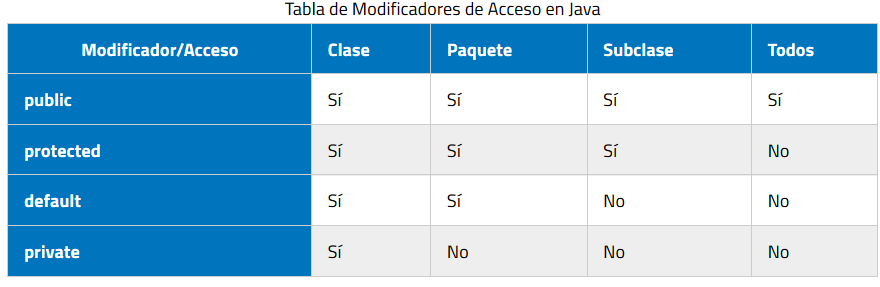
Al permitir el acceso a datos privados solo a través de un conjunto de métodos bien definidos, **podemos evitar que se asignen valores incorrectos a esos datos**, por ejemplo, realizando una verificación de rango.

* No es posible que el código fuera de la clase establezca el valor de un miembro privado directamente.
* También podemos controlar con precisión cómo y cuándo se utilizan los datos dentro de un objeto.

|  |
| --- |
| Por lo tanto, cuando se implementa correctamente, una clase crea una “caja negra” que se puede usar, pero cuyo funcionamiento interno no está abierto a alteraciones. |

Como su nombre indica, los modificadores de acceso en Java ayudan a restringir el alcance de una clase, constructor, variable, método o miembro de datos. Hay cuatro tipos de modificadores de acceso disponibles en Java:

1. Default – No se requiere palabra clave
2. Private
3. Protected
4. Public



### 7.1.1 Modificador de acceso por defecto (default)

Cuando no se especifica ningún modificador de acceso para una clase, método o miembro de datos, se dice estar teniendo **modificador de acceso default** por defecto.

Los miembros de datos, clase o métodos que no se declaran utilizando ningún modificador de acceso, es decir, que tengan un modificador de acceso predeterminado, solo son accesibles **dentro del mismo paquete**.

En este ejemplo, crearemos dos paquetes y las clases en los paquetes tendrán los modificadores de acceso predeterminados e intentaremos acceder a una clase de un paquete desde otra clase del segundo paquete.

// Programa Java para ilustrar el modificador default

package p1;

// La clase DemoDefault tiene modificador de acceso default

class DemoDefault {

void mostrar()

{

System.out.println("Hola Mundo!");

}

}

// Programa Java para ilustrar el error  
// Al usar la clase de un paquete diferente con  
// modificador default

package p2;

import p1.\*;

// Esta clase tiene un modificador de acceso predeterminado

public class DemoDefaultEjecutar {

public static void main(String args[])

{

DemoDefault obj = new DemoDefault();

obj.mostrar();

}

}

Salida:

Error:(12, 9) java: cannot find symbol

symbol: class DemoDefault

location: class p2.DemoDefaultEjecutar

### 7.1.2 Modificador de acceso privado (private)

El modificador de acceso privado se especifica con la palabra clave private. Los métodos o los miembros de datos declarados como privados solo son accesibles dentro de la clase en la que se declaran.

* Cualquier otra clase del mismo paquete no podrá acceder a estos miembros.
* Las clases e interfaces no se pueden declarar como privadas (**private**).

En este ejemplo, crearemos dos clases A y B dentro del mismo paquete p1. Declararemos un método en la clase A como privado e intentaremos acceder a este método desde la clase B y veremos el resultado.

// Programa Java para ilustrar el error

// al usar la clase desde un mismo paquete

// con modificador private

package p1;

class A {

private void mostrar() {

System.out.println("Ejemplo modificadores Java ");

}

}

class B {

public static void main(String[] args) {

A obj= new A();

//tratando de acceder al método privado de otra clase

obj.mostrar();

}

}

Salida:

Error:(15, 12) java: mostrar() has private access in p1.A

### 7.1.3 Modificador de acceso protegido (protected)

El modificador de acceso protegido se especifica con la palabra clave protected.

* Los métodos o miembros de datos declarados como **protected** son accesibles dentro del mismo paquete o subclases en paquetes diferentes.

En este ejemplo, crearemos dos paquetes p1 y p2. La clase A en p1 es public, para acceder a ella desde p2. El método que se muestra en la clase A está protegido y la clase B se hereda de la clase A y, a continuación, se accede a este método protegido creando un objeto de clase B.

// Programa Java para ilustrar  
// el modificador protected

package p1;

public class A {

protected void mostrar(){

System.out.println("Java ejemplo");

}

}

// Programa Java para ilustrar el modificador protected

package p2;

// importar todas las clases en el paquete p1

import p1.\*;

public class B extends A {

public static void main(String[] args) {

B obj = new B();

obj.mostrar();

}

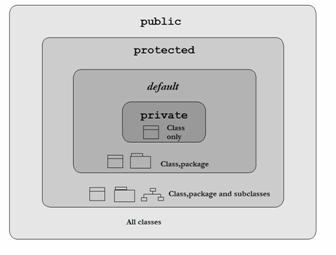
}

Salida:

Java ejemplo

En resumen **a un miembro protected se puede acceder desde su mismo paquete y desde cualquier clase que extienda la clase en que se encuentra**, independientemente de si esta se encuentra en el mismo paquete o no. Este modificador, como private, no tiene sentido a nivel de clases o interfaces no internas.

En otras palabras, si determinada clase Hijo hereda el comportamiento de una clase Padre, la clase Hijo tendrá acceso a todos aquellos campos/métodos definidos como *protected* en Padre, pero no aquellos declarados como private en Padre.



### 7.1.4 Modificador de acceso público (public)

El modificador de acceso público se especifica con la palabra clave **public**.

* El modificador de acceso público tiene el alcance más amplio entre todos los demás modificadores de acceso.
* Las clases, métodos o miembros de datos que se declaran como públicos son accesibles desde cualquier lugar del programa. No hay restricciones en el alcance de los miembros de datos públicos.

// Programa Java para ilustrar el modificador public

package p1;

public class A {

public void mostrar(){

System.out.println("Java desde Cero");

}

}

// Programa Java para ilustrar el modificador protected

package p2;

// importar todas las clases en el paquete p1

import p1.\*;

public class B extends A {

public static void main(String[] args) {

A obj = new A();

obj.mostrar();

}

}

### 7.1.5 Modificadores que no son de acceso

**En Java, tenemos 7 modificadores que no son de acceso o, a veces, también llamados** **especificadores**. Se usan con clases, métodos, variables, constructores, etc. para proporcionar información sobre su comportamiento a la [JVM](https://javadesdecero.es/fundamentos/como-funciona-maquina-virtual/). Y son:

* static
* final
* abstract
* synchronized
* transient
* volatile
* native

**STATIC:**

Sirve para crear miembros que pertenecen a la clase, y no a una instancia de la clase. Esto implica, entre otras cosas, que **no es necesario crear un objeto de la clase para poder acceder a estos atributos y métodos**.

* En ocasiones es necesario o conveniente generar elementos que tomen un mismo valor para cualquier número de instancias generadas o bien invocar/llamar métodos sin la necesidad de generar instancias, y es bajo estas dos circunstancias que es empleado el calificador *static*.
* Dos aspectos característicos de utilizar el calificador static en un elemento Java son los siguientes:
  + **No puede ser generada ninguna instancia** (uso de *new*) de un elemento *static* puesto que solo existe una instancia.
  + Todos los elementos definidos dentro de una estructura *static* deben ser *static* ellos mismos, o bien, poseer una instancia ya definida para poder ser invocados.

*NOTA*: Lo anterior no implica que no puedan ser generadas instancias dentro de un elemento *static*; no es lo mismo llamar/invocar que crear/generar.

**FINAL:**

Indica que una variable, método o clase no se va a modificar, lo cuál puede ser útil para añadir más semántica, por cuestiones de rendimiento, y para detectar errores.

* Si una **variable** se marca como *final*, **no se podrá asignar un nuevo valor** a la variable.
* Si una **clase** se marca como *final*, **no se podrá extender** la clase.
* Si es un **método** el que se declara como *final*, **no se podrá sobreescribir**.

Algo a tener en cuenta a la hora de utilizar este modificador es que si es un objeto lo que hemos marcado como *final*, esto no nos impedirá modificar el objeto en sí, sino tan solo usar el operador de asignación para cambiar la referencia. Por lo tanto:

// El siguiente código NO funcionaría:

public class Ejemplo {

public static void main(String[] args) {

final String cadena = "Hola";

cadena = new String("Adios");

}

}

// El siguiente código SI funcionaría:

public class Ejemplo {

public static void main(String[] args) {

final String cadena = "Hola";

cadena.concat(" mundo");

}

***NOTA***: Una variable con modificadores *static* y *final* sería lo más cercano en Java a las constantes de otros lenguajes de programación.

**ABSTRACT:**  
  
La palabra clave *abstract* indica que **no se provee una implementación para un cierto método**, sino que la implementación vendrá dada por las clases que extiendan la clase actual. **Una clase que tenga uno o más métodos abstract debe declararse como *abstract*** a su vez.

* **Mecanismos de sincronización: VOLATILE y SYNCHRONIZED.**

**VOLATILE**:  
  
volatile es más simple y más sencillo que synchronized, lo que implica también un mejor rendimiento. Sin embargo **volatile, a diferencia de synchronized, no proporciona** [atomicidad](http://es.wikipedia.org/wiki/Atomicidad" \t "Atomicidad) (que o se ejecuten todos los pasos o ninguno.

Una operación como el incremento, por ejemplo, no es atómica. El operador de incremento se divide en realidad en 3 instrucciones distintas (primero se lee la variable, después se incrementa, y por último se actualiza el valor) por lo que algo como lo siguiente podría causarnos problemas a pesar de que la variable sea volatile:

//Ejemplo de volatile sobre una variable

volatile int contador;

public void aumentar() {

contador++;

}

Las **variables volátiles** en el lenguaje Java son declaradas anteponiendo el modificador de visibilidad de memoria ***volatile***. Estas no son guardadas en el caché del procesador, es decir, toda lectura/escritura de la misma se realiza **directamente contra la memoria principal**.

Anteponiendo volatile a la definición de la variable \_stop estamos diciendo al compilador que el valor contenido en esa dirección de memoria puede modificarse en cualquier momento, y queremos que dicha modificación sea visible inmediatamente para todos los hilos que están accediendo a ella, no queremos que su valor sea copiado en la caché del procesador

**SYNCHRONIZED:**

En caso de que necesitemos atomicidad podemos recurrir a synchronized o a cosas más avanzadas, como las clases del API *java.util.concurrent* de Java 5.

*synchronized* se diferencia de *volatile* entre otras cosas en que este modificador **se utiliza sobre bloques de código y métodos**, y no sobre variables. Al utilizar *synchronized* sobre un bloque se añade entre paréntesis una referencia a un objeto que utilizaremos a modo de lock.

//Ejemplo de synchronized sobre un bloque de código

int contador;

public void aumentar() {

synchronized(this) {

contador++;

}

}

//Ejemplo de synchronized sobre un metodo

int contador;

public void synchronized aumentar() {

contador++;

}

### 7.1.6 Puntos importantes

Si otros programadores usan tu clase, intenta usar el nivel de acceso más restrictivo que tenga sentido para un miembro en particular. Usa private a menos que tengas una buena razón para no hacerlo. También evita los campos públicos, excepto las constantes.

<https://javadesdecero.es/poo/modificadores-de-acceso/>

[https://medium.com/@pablocastelnovo/variables-vol%C3%A1tiles-en-java-f5ae078bf8b9](https://medium.com/@pablocastelnovo/variables-volátiles-en-java-f5ae078bf8b9)

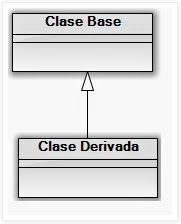
## 7.2. Herencia en Java

La herencia es una de las características fundamentales de la Programación Orientada a Objetos.

Mediante la herencia podemos **definir una clase a partir de otra ya existente**.

La clase nueva se llama **clase derivada** o subclase y la clase existente se llama **clase base** o superclase.

En UML (Unified Modelling Language) la herencia se representa con una flecha apuntando desde la clase derivada a la clase base.



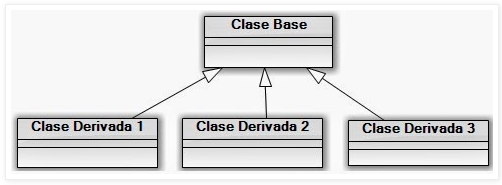
La clase derivada hereda los componentes (atributos y métodos) de la clase base.La finalidad de la herencia es:

**Extender** la funcionalidad de la clase base: en la clase derivada se pueden **añadir** atributos y métodos nuevos.

**Especializar** el comportamiento de la clase base: en la clase derivada se pueden **modificar** (sobrescribir, override) los métodos heredados para adaptarlos a sus necesidades.

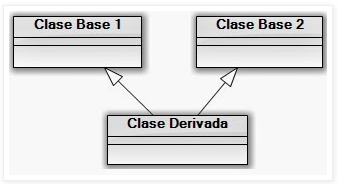
La herencia permite la **reutilización del código**, ya que evita tener que reescribir de nuevo una clase existente cuando necesitamos ampliarla en cualquier sentido. Todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

**Reutilización de código:** El código se escribe una vez en la clase base y se utiliza en todas las clases derivadas.

Una clase base puede serlo de tantas derivadas como se desee: Un solo padre, varios hijos.

**Herencia múltiple en Java: Java no soporta la herencia múltiple**. Una clase derivada solo puede tener una clase base.

Diagrama UML de herencia múltiple no permitida en Java



La herencia expresa una relación **“ES UN/UNA”** entre la clase derivada y la clase base.

Esto significa que **un objeto de una clase derivada es también un objeto de su clase base**.

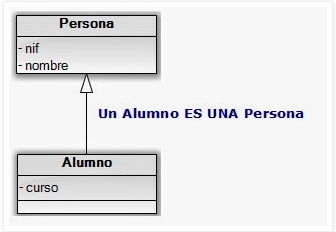
**Al contrario, NO es cierto**. Un objeto de la clase base no es un objeto de la clase derivada.

Por ejemplo, supongamos una clase Vehiculo como la clase base de una clase Coche. Podemos decir que un Coche es un Vehiculo pero un Vehiculo no siempre es un Coche, puede ser una moto, un camión, etc.

Un objeto de una clase derivada es a su vez un objeto de su clase base, por lo tanto, se puede utilizar en cualquier lugar donde aparezca un objeto de la clase base. Si esto no fuese posible entonces la herencia no está bien planteada.

**Ejemplo de herencia bien planteada:**

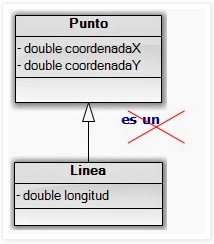
A partir de una clase Persona que tiene como atributos el nif y el nombre, podemos obtener una clase derivada Alumno. Un Alumno es una Persona que tendrá como atributos nif, nombre y curso.



**Ejemplo de herencia mal planteada:**

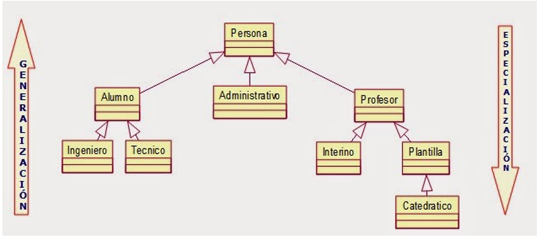
Supongamos una clase Punto que tiene como atributos coordenadaX y coordenadaY.

Se puede crear una clase Linea a partir de la clase Punto. Simplificando mucho para este ejemplo, podemos considerar una línea como un punto de origen y una longitud. En ese caso podemos crear la Clase Linea como derivada de la clase Punto, pero el planteamiento no es correcto ya que no se cumple la relación *ES UN*



Una Linea NO ES un Punto. En este caso no se debe utilizar la herencia.

Una clase derivada a su vez puede ser clase base en un nuevo proceso de derivación, formando de esta manera **una Jerarquía de Clases**.



Las clases más generales se sitúan en lo más alto de la jerarquía. Cuánto más arriba en la jerarquía, menor nivel de detalle.

Cada clase derivada debe implementar únicamente lo que la distingue de su clase base.

En java todas las clases derivan directa o indirectamente de la clase Object.

**Object** es la clase base de toda la jerarquía de clases Java. Todos los objetos en un programa Java son Object.

**CARACTERÍSTICAS DE LAS CLASES DERIVADAS**

Una clase derivada hereda de la clase base sus componentes (atributos y métodos).

Los constructores no se heredan. Las clases derivadas deberán implementar sus propios constructores.

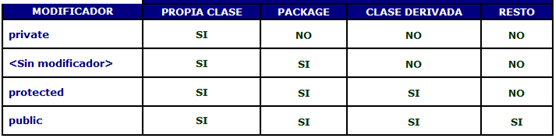
* Una clase derivada **puede acceder a los miembros públicos y protegidos** de la clase base como si fuesen miembros propios.
* Una clase derivada **no tiene acceso a los miembros privados** de la clase base. Deberá acceder a través de métodos heredados de la clase base.
* Si se necesita tener acceso directo a los miembros privados de la clase base se deben declarar **protected** en lugar de **private** en la clase base.

Una clase derivada puede añadir a los miembros (atributos) heredados, sus propios atributos y métodos (extender la funcionalidad de la clase). También puede modificar los métodos heredados (especializar el comportamiento de la clase base).

Una clase derivada puede a su vez, ser una clase base, dando lugar a una jerarquía de clases.

**MODIFICADORES DE ACCESO JAVA**

El siguiente gráfico indica el acceso a elementos de una clase según el tipo de modificador:



Los métodos también pueden no tener modificador. Hacer ejemplo con la clase Empleado (sacamos una de las clases derivadas fuera del paquete para comprobarlo, con el método aumentar sueldo)

El significado concreto del modificador de acceso **protected** varía ligeramente entre Java y C++: En C++, un atributo que sea declarado como “protected” en una clase será visible únicamente en dicha clase y en las subclases de la misma. En Java, un atributo que sea declarado como “protected” será visible en dicha clase, en el “package” que se encuentre la misma, y en las subclases de la misma.

**HERENCIA EN JAVA. SINTAXIS**

La herencia en Java se expresa mediante la palabra **extends**

Por ejemplo, para declarar una clase B que hereda de una clase A:

public class B extends A {

….

}

**Ejemplo de herencia Java**: Disponemos de una clase Persona con los atributos nif y nombre.

//Clase Persona. Clase Base

public class Persona {

private String nif;

private String nombre;

public String getNif() {

return nif;

}

public void setNif(String nif) {

this.nif = nif;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

}

Queremos crear ahora una clase Alumno con los atributos nif, nombre y curso. Podemos crear la clase Alumno como derivada de la clase Persona. La clase Alumno contendrá solamente el atributo curso, el nombre y el nif son los heredados de Persona:

//Clase Alumno. Clase derivada de Persona

public class Alumno extends Persona{

//extends llama al constructor de persona con el nif

//y nombre a null

private String curso;

public String getCurso() {

return curso;

}

public void setCurso(String curso) {

this.curso = curso;

}

}

La clase alumno hereda los atributos nombre y nif de la clase Persona y añade el atributo propio curso. Por lo tanto:

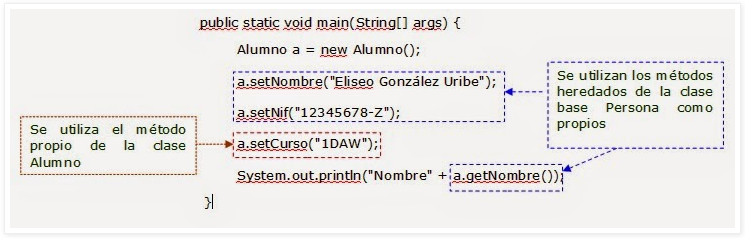
Los atributos de la clase Alumno son nif, nombre y curso.

Los métodos de la clase Alumno son: getNif(), setNif(String nif), getNombre(), setNombre(String nombre), getCurso(), setCurso(String curso).

La clase Alumno aunque hereda los atributos nif y nombre, no puede acceder a ellos de forma directa ya que son privados a la clase Persona. Se acceden a través de los métodos heredados de la clase base.

La clase Alumno puede utilizar los componentes public y protected de la clase Persona como si fueran propios.

Ejemplo de uso de la clase Alumno:



**En una jerarquía de clases**, cuando un objeto invoca a un método:

1. Se busca en su clase el método correspondiente.
2. Si no se encuentra, se busca en su clase base.
3. Si no se encuentra se sigue buscando hacia arriba en la jerarquía de clases hasta que el método se encuentra.
4. Si al llegar a la clase base raíz el método no se ha encontrado se producirá un error.

**REDEFINIR MIEMBROS DE LA CLASE BASE EN LAS CLASES DERIVADAS**

**Redefinir o modificar métodos de la clase Base (Override/Sobreescribir)**

Los métodos heredados de la clase base se pueden redefinir (también se le llama modificar o sobreescribir) en las clases derivadas.

|  |
| --- |
| El método en la clase derivada se debe escribir con el mismo nombre, el mismo número y tipo de parámetros y el mismo tipo de retorno que en la clase base. Si no fuera así estaríamos sobrecargando el método, no redefiniéndolo. |

El método sobrescrito puede tener un **modificador de acceso menos restrictivo** que el de la clase base. Si, por ejemplo, el método heredado es protected se puede redefinir como public pero no como private porque sería un acceso más restrictivo que el que tiene en la clase base.

Cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en la clase base.

Si queremos acceder al método de la clase base que ha quedado oculto en la clase derivada utilizamos:

super.metodo();

Si se quiere evitar que un método de la clase Base sea modificado en la clase derivada se debe declarar como método **final**. Por ejemplo:

public final void metodo(){

........}

**Ejemplo**:

Vamos a añadir a la clase Persona un método leer() para introducir por teclado los valores a los atributos de la clase. La clase Persona queda así:

public class Persona {

private String nif;

private String nombre;

public String getNif() {

return nif;

}

public void setNif(String nif) {

this.nif = nif;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public void leer(){

Scanner sc = new Scanner(System.in);

System.out.print("Nif: ");

nif = sc.nextLine();

System.out.print("Nombre: ");

nombre = sc.nextLine();

}

}

La clase Alumno que es derivada de Persona, heredará este método leer() y lo puede usar como propio.

Podemos crear un objeto Alumno:

Alumno a = new Alumno();

Y utilizar este método:

a.leer();

Pero utilizando este método solo se leen por teclado el nif y el nombre. En la clase Alumno se debe sobreescribir o modificar este método heredado para que también se lea el curso. El método leer() de Alumno invocará al método leer() de Persona para leer el nif y nombre y a continuación se leerá el curso.

La clase Alumno con el método leer() modificado queda así:

//Clase Alumno. Clase derivada de Persona

public class Alumno extends Persona{

private String curso;

public String getCurso() {

return curso;

}

public void setCurso(String curso) {

this.curso = curso;

}

@Override //indica que se modifica un método heredado

public void leer(){

Scanner sc = new Scanner(System.in);

super.leer();

//se llama al método leer de Persona para leer nif y nombre

System.out.print("Curso: ");

curso = sc.nextLine(); //se lee el curso

}

}

Como se ha dicho antes, cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en la clase base. Por eso para ejecutar el método leer() de Persona se debe escribir **super.leer();**

**Redefinir atributos de la clase Base**

Una clase derivada puede volver a declarar un atributo heredado (Atributo public o protected en la clase base). En este caso, el atributo de la clase base queda oculto por el de la clase derivada.

Para acceder a un atributo de la clase base que ha quedado oculto en la clase derivada se escribe: **super.atributo;**

**CONSTRUCTORES Y HERENCIA EN JAVA. CONSTRUCTORES EN CLASES DERIVADAS**

Los constructores no se heredan. Cada clase derivada tendrá sus propios constructores.

* La clase base es la encargada de inicializar sus atributos.
* La clase derivada se encarga de inicializar solo los suyos.

Cuando se crea un objeto de una clase derivada se ejecutan los constructores en este orden:

1. Primero se ejecuta el constructor de la clase base

2. Después se ejecuta el constructor de la clase derivada.

Esto lo podemos comprobar si añadimos a las clases Persona y Alumno sus constructores por defecto y hacemos que cada constructor muestre un mensaje:

public class Persona {

private String nif;

private String nombre;

public Persona() {

System.out.println("Ejecutando el constructor de Persona");

}

/////// Resto de métodos

}

public class Alumno extends Persona{

private String curso;

public Alumno() {

System.out.println("Ejecutando el constructor de Alumno");

}

/////// Resto de métodos

}

Si creamos un objeto Alumno:

Alumno a = new Alumno();

Se muestra por pantalla:

Ejecutando el constructor de Persona

Ejecutando el constructor de Alumno

Cuando se invoca al constructor de la clase Alumno se invoca automáticamente al constructor de la clase Persona y después continúa la ejecución del constructor de la clase Alumno.

**El constructor por defecto de la clase derivada llama al constructor por defecto de la clase base.**

La instrucción para invocar al constructor por defecto de la clase base es:

**super();**

**Todos los constructores por defecto en las clases derivadas contienen de forma implícita la instrucción super() como primera instrucción.**

public Alumno() {

super(); //esta instrucción se ejecuta siempre. No es //necesario escribirla

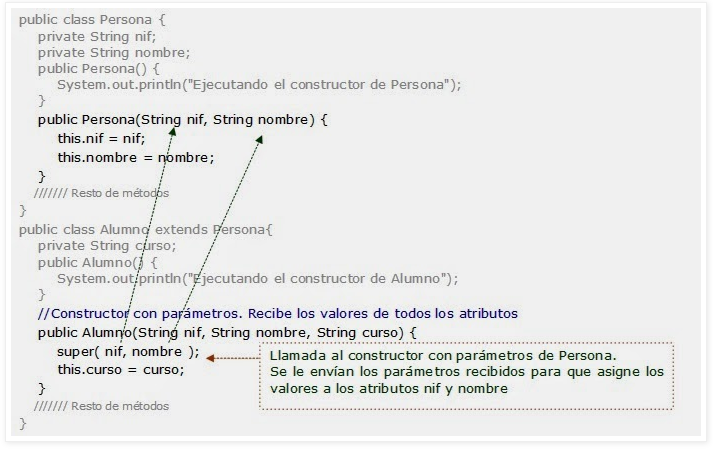
System.out.println("Ejecutando el constructor de Alumno");

}

Cuando se crea un objeto de la clase derivada y queremos asignarles valores a los atributos heredados de la clase base:

1. La clase derivada debe tener un constructor con parámetros adecuado que reciba los valores a asignar a los atributos de la clase base.
2. La clase base debe tener un constructor con parámetros adecuado.
3. El constructor de la clase derivada invoca al constructor con parámetros de la clase base y le envía los valores iniciales de los atributos. Debe ser la **primera** instrucción.
4. La clase base es la encargada de asignar valores iniciales a sus atributos.
5. A continuación, el constructor de la clase derivada asigna valores a los atributos de su clase.

**Ejemplo:** en las clases Persona y Alumno anteriores añadimos constructores con parámetros:



Ahora se pueden crear objetos de tipo Alumno y asignarles valores iniciales. Por ejemplo:

Alumno a = new Alumno ("12345678-Z","Eliseo González Manzano","1DAW");

**CLASES FINALES**

Si queremos evitar que una clase tenga clases derivadas debe declararse con el modificador **final** delante de class:

public **final** class A{

........

}

Esto la convierte en clase final. Una clase final no se puede heredar.

Si intentamos crear una clase derivada de A se producirá un error de compilación:

public class B extends A{ //extends A producirá un error de compilación

........

}

**CLASES ABSTRACTAS**

**Una clase abstracta es una clase que NO se puede instanciar,** es decir, no se pueden crear objetos de esa clase.

Se diseñan solo para que otrasclases hereden de ella.

La clase abstracta normalmente es la raíz de una jerarquía de clases y contendrá el comportamiento general que deben tener todas las subclases. En las clases derivadas se detalla la implementación.

Las clases abstractas:

* Pueden contener cero o más métodos abstractos.
* Pueden contener métodos no abstractos.
* Pueden contener atributos.

Todas las clases que hereden de una clase abstracta deben implementar todos los métodos abstractos heredados.

Si una clase derivada de una clase abstracta no implementa algún método abstracto se convierte en abstracta y tendrá que declararse como tal (tanto la clase como los métodos que siguen siendo abstractos).

Aunque no se pueden crear objetos de una clase abstracta, sí pueden tener constructores para inicializar sus atributos que serán invocados cuando se creen objetos de clases derivadas.

La forma general de declarar una clase abstracta en Java es:

[modificador] **abstract** class nombreClase{

}

**Ejemplo de clase Abstracta en Java**: Clase Polígono.

//Clase abstracta Poligono

public abstract class Poligono {

private int numLados;

public Poligono() {}

public Poligono(int numLados) {

this.numLados = numLados;

}

public int getNumLados() {

return numLados;

}

public void setNumLados(int numLados) {

this.numLados = numLados;

}

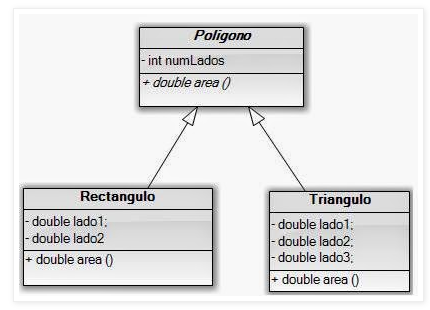
//Declaración del método abstracto area()

public abstract double area();

}

La clase Poligono contiene un único atributo *numLados.* Es una clase abstracta porque contiene el método abstracto area().

A partir de la clase Poligono vamos a crear dos clases derivadas Rectangulo y Triangulo. Ambas deberán implementar el método area(). De lo contrario también serían clases abstractas.

El diagrama UML es el siguiente:

En UML las clases abstractas y métodos abstractos se escriben con su nombre en cursiva.

//Clase Rectangulo

public class Rectangulo extends Poligono{

private double lado1;

private double lado2;

public Rectangulo() {

}

public Rectangulo(double lado1, double lado2) {

super(2);

this.lado1 = lado1;

this.lado2 = lado2;

}

public double getLado1() {

return lado1;

}

public void setLado1(double lado1) {

this.lado1 = lado1;

}

public double getLado2() {

return lado2;

}

public void setLado2(double lado2) {

this.lado2 = lado2;

}

//Implementación del método abstracto area()

//heredado de la clase Polígono

@Override

public double area(){

return lado1 \* lado2;

}

}

//Clase Triangulo

public class Triangulo extends Poligono{

private double lado1;

private double lado2;

private double lado3;

public Triangulo() {

}

public Triangulo(double lado1, double lado2, double lado3) {

super(3);

this.lado1 = lado1;

this.lado2 = lado2;

this.lado3 = lado3;

}

public double getLado1() {

return lado1;

}

public void setLado1(double lado1) {

this.lado1 = lado1;

}

public double getLado2() {

return lado2;

}

public void setLado2(double lado2) {

this.lado2 = lado2;

}

public double getLado3() {

return lado3;

}

public void setLado3(double lado3) {

this.lado3 = lado3;

}

//Implementación del método abstracto area()

//heredado de la clase Polígono

@Override

public double area(){

double p = (lado1+lado2+lado3)/2;

return Math.sqrt(p \* (p-lado1) \* (p-lado2) \* (p-lado3));

}

}

**Ejemplo** de uso de las clases:

public static void main(String[] args) {

Triangulo t = new Triangulo(3.25, 4.55, 2.71);

System.out.printf("Área del triángulo: %.2f %n”, t.area());

Rectangulo r = new Rectangulo(5.70,2.29);

System.out.printf("Área del rectángulo: %.2f %n”, r.area());

}

**CASTING: CONVERSIONES ENTRE CLASES**

**UpCasting: Conversiones implícitas**

La herencia establece una relación *ES UN* entre clases. Esto quiere decir que un objeto de una clase derivada es también un objeto de la clase base.

Por esta razón:

*Se puede asignar de forma implícita una referencia a un objeto de una clase derivada a una referencia de la clase base. Son* ***tipos compatibles****.*

*También se llaman conversiones ascendentes o* ***upcasting****.*

En el ejemplo anterior un triángulo en un Poligono y un cuadrado es un Poligono.

Poligono p;

Triangulo t = new Triangulo(3,5,2);

Como un triángulo es un polígono, se puede hacer esta asignación:

**p=t;**

La variable p de tipo Poligono puede contener la referencia de un objeto Triangulo ya que son tipos compatibles.

Cuando manejamos un objeto **a través de una referencia a una superclase** (directa o indirecta) **solo se pueden ejecutar métodos disponibles en la superclase**.

En el ejemplo, la instrucción:

p.getLado1();

Provocará un error ya que p es de tipo Poligono y el método getLado1() no es un método de esa clase.

Cuando manejamos un objeto a **través de una referencia a una superclase** (directa o indirecta) y **se invoca a un método que está redefinido en las subclases se ejecuta el método de la clase a la que pertenece el objeto no el de la referencia**.

En el ejemplo, la instrucción

p.area();

Ejecutará el método area() de Triángulo.

**DownCasting: Conversiones explícitas**

Se puede asignar una referencia de la clase base a una referencia de la clase derivada, siempre que la referencia de la clase base sea a un objeto de la misma clase derivada a la que se va a asignar o de una clase derivada de ésta.

También se llaman conversiones descendentes o **downcasting**.

Esta conversión debe hacerse mediante un **casting**.

Siguiendo con el ejemplo anterior:

Poligono p = new Triangulo(1,3,2); //upcasting

Triangulo t;

t = (Triangulo) p; //downcasting

Esta asignación se puede hacer porque p contiene la referencia de un objeto triángulo.

Las siguientes instrucciones provocarán un error de ejecución del tipo **ClassCastException**:

Triangulo t;

Poligono p1 = new Rectangulo(3,2);

t = (Triangulo)p1; //----> Error de ejecución

p1 contiene la referencia a un objeto Rectangulo y no se puede convertir en una referencia a un objeto Triangulo. No son tipos compatibles.

**EL OPERADOR instanceof**

Las operaciones entre clases y en particular el downcasting requieren que las clases sean de tipos compatibles. Para asegurarnos de ello podemos utilizar el operador **instanceof**.

instanceof devuelve true si el objeto es instancia de la clase y false en caso contrario.

La sintaxis es:

Objeto instanceof Clase

Ejemplo:

Triangulo t;

Poligono p1 = new Rectangulo(3,2);

if(p1 instanceof Triangulo)

t = (Triangulo)p1;

else

System.out.println("Objetos incompatibles");

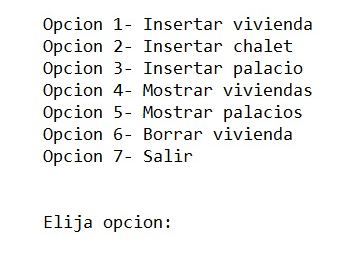
## 7.2. Ejercicios Herencia

**1.-** Realiza un programa que sirva para representar centros educativos de diferentes tipos. Los tipos que se puede tener son: públicos, concertados y privados. Para todos ellos se quiere tener su nombre y dirección. Para los públicos se tendrá la asignación anual pública recibida. Para los concertados la asignación mensual pública recibida, así como la cuota mensual que pagan los alumnos. Para los privados se tendrá la cuota mensual que pagan los alumnos. Se pide crear tres centros, uno de cada tipo y guardarlos en un arrayList en el programa principal. Imprimir a continuación los datos de cada centro educativo usando el arrayList donde están guardados.

**2.-** Realiza un programa en java con las siguientes clases:

* Vivienda:
  + - Atributos: metros cuadrados, calle y número.
    - Métodos para devolver/establecer valores a los atributos.
    - Constructor con valores por defecto (0,””,0)
    - Constructor con valores para los atributos
* Chalet: Una vivienda que siempre tiene 120m2 y puede o no tener jardín (verdadero o falso)
* Palacio: Una vivienda de la que interesa el número de habitaciones.

En el programa principal se mostrará el siguiente menú:



En las tres primeras opciones nos pedirán los datos de cada tipo de vivienda (el tipo general corresponde a una vivienda que no es ni chalet ni palacio).

En la cuarta opción se mostrarán todas las viviendas de cualquier tipo.

En la quinta opción se mostrarán los datos de los palacios.

En la sexta opción se pide el nombre de la calle y el número de la vivienda a borrar y la eliminamos de la lista.

**3.-** Diseña una jerarquía de clases para los diferentes productos que se pueden encontrar en un supermercado (Alimentación, Droguería y Textil). Todos tienen un conjunto de características comunes (código, nombre y precio) y una serie de características específicas de cada producto. Los productos de alimentación tienen fecha de caducidad (no se podrá introducir un producto si la fecha de caducidad es anterior a la fecha actual, en ese caso lanza la excepción propia FechaNoValidaException), los de droguería tienen marca y los de textil color.

La fecha será introducida por el usuario pidiéndole día, mes y año por separado (harás un paquete externo al proyecto, llamado utilidadesFecha con una clase llamada ValidacionFecha que tendrá un método que reciba dia, mes y año y devuelva si es correcta la fecha, una fecha en formato LocalDate y en caso contrario una excepción denominada FechaNoValidaException)

<https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html>

--suponer que el usuario siempre mete las cosas bien

El código debe adaptarse a un tamaño de 5 caracteres de los cuales el primero y el último son letras en mayúscula y el resto números. Utiliza expresiones regulares para controlarlo, lanzando una excepción CodigoInvalidoException que crearás si no cumple la notación.

//con los codigos de las letras ASCII se puede hacer sin excepciones

Crea un carrito de la compra en el que se pueden incluir productos y emitir un ticket en el que figuren los datos de cada producto del carrito, incluyendo su precio y el importe total de la compra. Si se quiere más de una unidad de un producto, se pide varias veces, para simplificar.

Para ello, el programa al arrancar mostrará el menú:

**PROCESO DE CREACIÓN DEL ALMACÉN**

**1. Introducir producto alimenticio**

**2. Introducir producto textil**

**3. Introducir producto droguería**

**4. Terminar.**

**Elija opción:**

Con ese menú, se creará un arraylist con varios productos para poder simular los productos de un supermercado y poder hacer compras.

Entonces empezará el proceso de compras, mostrando para cada cliente el listado de productos y pidiendo al cliente que seleccione el producto que quiere comprar por nombre, hasta que ya no quiera más productos. En ese momento, se procede imprimir el ticket con los distintos productos y el precio total a pagar.

Para cada cliente nuevo se visualizará lo siguiente:

**¡¡¡¡Acaba de entrar en el supermercado virtual!!!!**

**Lista de productos**

**1 Nombre:peras Precio:2.0 fecha caducidad alimento:23/2/1989**

**2 Nombre:toalla Precio:8.0 Color:azul**

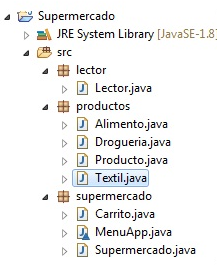
**3 Nombre:gel Precio:2.0 Marca:Legrain**

**Desea comprar producto?(S/N)**

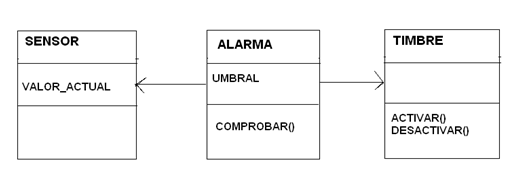
Cuando el cliente termine (responde N), se calcula el importe de su compra y se empieza con un nuevo cliente (previamente se pregunta si hay un nuevo cliente)

Cuando no haya más clientes, terminaremos la ejecución del programa.

Esquema del proyecto (lector lo puedes importar como paquete externo y no incluirlo en el proyecto):



**4.-** Crea una clase denominada *Alarma* cuyos objetos activen un objeto de tipo *Timbre* cuando el valor medido por un sensor supere un umbral preestablecido:



Implementa en Java todo el código necesario para el funcionamiento de la alarma, suponiendo que la alarma comprueba si debe activar o desactivar el timbre cuando se invoca el método *comprobar().*

Crea una subclase de Alarma denominada *Alarma\_Luminosa*, que además de activar el timbre, encienda una luz (que representaremos con un objeto de tipo *Bombilla*). Aseguraos de que cuando se activa la alarma luminosa se enciende la luz de alarma y también suena la señal sonora asociada al timbre.

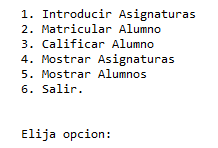
**5.** Tenemos que hacer un programa que gestione un centro en el que se dispone de 5 asignaturas en las que se podrán matricular los alumnos del centro. Cada alumno se matriculará de 3 asignaturas.

De dichos alumnos nos interesan las siguientes propiedades:

nombre, DNI, curso, grupo y asignaturas en las que se ha matriculado (nombre), así como la nota que ha obtenido en cada una de ellas.

De cada asignatura queremos conocer el nombre y el precio.

Crearás el siguiente menú:



En la opción 1 el programa pedirá las 5 asignaturas junto con sus precios de que dispone el centro.

En la opción 2 el programa pedirá los datos de un alumno a matricular y los nombres de 3 asignaturas de las 5 disponibles.

En la opción 3 el programa pedirá los datos de un alumno a calificar en una determinada asignatura, de la que pedirá el nombre y le dará una calificación de 1 a 10 al azar.

En la opción 4 el programa muestra las asignaturas de las que se puede matricular un alumno, así como sus precios.

En la opción 5 el programa muestra todos los alumnos junto con las asignaturas de las que se ha matriculado, además de su nota en cada asignatura. (Si todavía no hay nota indicará “no disponible”) y el precio total de la matrícula.

Muestra al usuario en cada opción los datos que ya tiene almacenado el programa y que le puedan ser de ayuda para realizar la tarea pedida y considera los mensajes a mostrar en caso de no poder realizar la operación, indicando al usuario el motivo por pantalla.

## 7.3. Clases abstractas

No hay Word de clases abstractas, ver ejemplos de Eclipse

## 7.4. Interfaces

**INTERFACES – (*No leer, Ver ejemplos para 2eva*)**

### **7.4.1 ¿Qué es y para qué sirve un interfaz?**

Un interfaz es un mecanismo que permite definir un conjunto de métodos y constantes, que proporcionarán una forma de acceso común a todas aquellas clases que implementen dicho interfaz. Puede verse un interfaz como un **mecanismo estándar de acceso a un conjunto de clases**.

Se trata de declarar métodos abstractos y constantes que, posteriormente puedan ser implementados de diferentes maneras según las necesidades de un programa.

Por ejemplo, el interfaz Comparable de la API de Java (puedes verlo en la documentación de Java <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html>) permite comparar objetos de forma estándar, sea cual sea la clase a la que pertenezcan, siempre y cuando dicha clase implemente **el interfaz Comparable**. En concreto, este interfaz solo contiene un método (*int compareTo(Object o)*). Ello nos asegura que, en cualquier objeto perteneciente a una de las clases que implementan el interfaz Comparable (por ejemplo, Integer, Date, Character, File, String, etc.), podemos invocar el método *compareTo* para compararlo con otro objeto.

### 7.4.2 ¿Cómo se programa un interfaz?

La API de Java proporciona bastantes interfaces ya programados. Si para tu aplicación necesitas definir un nuevo interfaz, puedes declararlo utilizando la palabra clave interface. En el interior de la declaración, debes declarar todas las constantes y métodos que deseas que tenga el interfaz. No puedes implementarlos: **en el interfaz solo se declaran los métodos**. Son las clases que implementan el interfaz las responsables de programar estos métodos.

public interface MiInterfaz {

int CONSTANTE = 100;  
 int metodoAbstracto( int parametro );

}

Se observa que las variables (atributos) adoptan la declaración en mayúsculas, pues en realidad actuarán como constantes *final*. En ningún caso estas variables actuarán como variables de instancia.

Por su parte, los métodos tras su declaración presentan un punto y coma, en lugar de su cuerpo entre llaves. Son métodos abstractos, por tanto, métodos sin implementación.**Ej:**

public interface Almacen {

// constantes del interfaz  
 public final static int CAPACIDAD\_MINIMA= 10;

// métodos del interfaz  
 public void almacenar(Object dato, int identificador);  
 public Object recuperar(int identificador);

}

final static no es necesario ponerlo, por defecto son **final static**.

En el ejemplo anterior se declara un interfaz para almacenar y recuperar objetos. El interfaz, como se puede ver, solo declara los métodos, y deben ser las clases que implementen el interfaz Almacen las que programen estos dos métodos. Además de los métodos, se declara la constante CAPACIDAD\_MINIMA, que puede ser accedida mediante la expresión Almacen.CAPACIDAD\_MINIMA.

### 7.4.3 ¿Cómo se implementa un interfaz?

Una clase, para implementar un interfaz, debe indicarlo en su declaración, utilizando la palabra clave implements. Pero no es suficiente con indicar que implementa el interfaz, debe declarar y programar todos los métodos que indica el interfaz.

class ImplementaInterfaz implements MiInterfaz{  
 int multiplicando=CONSTANTE;  
 int metodoAbstracto( int parametro ){  
 return ( parametro \* multiplicando );  
 }  
}

En este ejemplo se observa que han de codificarse todos los métodos que determina la interfaz (m*etodoAbstracto()*), y la validez de las constantes (*CONSTANTE*) que define la interfaz durante toda la declaración de la clase.

Una interfaz no puede implementar otra interfaz, aunque sí extenderla (*extends*) ampliándola.

Por ejemplo, a continuación, se presenta una clase que implementa el interfaz Almacen:

public class Armario implements Almacen {

// Declaración de atributos  
 // Otros métodos: constructores, etc.  
 // Otros métodos que no tienen nada que ver  
 // Con el interfaz

public void abrirPuertas() {  
 // Código del método  
 }

// Métodos del interfaz Almacen  
 public void almacenar(Object dato, int identificador) {  
 // Código necesario para almacenar el dato  
 }

public Object recuperar(int identificador) {  
 // Código necesario para recuperar el dato  
 }

}

Es muy importante recalcar que un interfaz obliga a aquellas clases (no abstractas) que indican que lo implementan a programar todos sus métodos. Sin embargo, la clase puede, libremente, programar cualquier otro método no especificado en el interfaz.

### 7.4.4 ¿Si una clase implementa un interfaz, sus subclases lo heredan?

Dada una clase que implementa un interfaz, toda subclase de ésta, implementa implícitamente el mismo interfaz, sin necesidad de indicarlo con la palabra clave implements, y sin necesidad de volver a programar o declarar los métodos del interfaz.

### 7.4.5 ¿Cómo se declara una referencia a un objeto que implementa un interfaz?

Se puede declarar una referencia a un objeto que implementa un interfaz, como en el caso general, del tipo de la clase a la cual pertenece el objeto. Por ejemplo, si Armario es una clase que implementa el interfaz Almacen:

Armario miArmario= new Armario();  
 miArmario.almacenar(new String("camisa blanca"), N\_ID);

Pero hay situaciones en que no resulta recomendable la solución anterior. Por ejemplo, si deseamos que una misma referencia pueda apuntar a, según sea el caso, objetos de distintas clases (pero todas ellas implementando el mismo interfaz), podemos utilizar directamente el nombre del interfaz en la declaración. Por ejemplo, si tenemos la clase Caja, que también implementa

Almacen:

Almacen miContenedor= new Armario();  
 miContenedor.almacenar(new String("camisa blanca"),56356);

...  
 // supongamos que ahora no se necesita más este objeto  
 // de la clase Armario, y se necesita uno de la clase  
 // Caja, utilizando la misma referencia:

miContenedor= new Caja();  
 miContenedor.almacenar(new Integer(34), 334233);

Otro caso en que resulta útil es cuando necesitamos un array de objetos que implementan un interfaz, pero cada uno de ellos puede ser de distinta clase:

Almacen[] contenedores= new Almacen[3];

contenedores[0]= new Armario();  
 contenedores[1]= new Caja();  
 contenedores[2]= new Armario();

### 7.4.6 ¿Cómo se declara un parámetro de un método para que este pueda ser un objeto de cualquier clase que implemente un determinado interfaz?

Si se declara el parámetro del método con el nombre del interfaz como tipo, entonces este parámetro podrá recibir un objeto de cualquier clase, siempre que esta implemente dicho interfaz. Por ejemplo, si las clases Armario y Caja implementan el interfaz Almacen:

public void guardarTodoEn(Almacen contenedor) {

contenedor.almacenar (objeto1, 23);  
 contenedor.almacenar (objeto2, 33);

}

...

// en otra parte de la clase

if (guardarEnCaja) guardarTodoEn(new Caja());  
 else guardarTodoEn(new Armario());

En el ejemplo, el método guardarTodoEn puede almacenar sus objetos en un objeto de cualquier clase que implemente el interfaz Almacen. Una restricción que tiene esto es que, de esta forma, solo se puede acceder directamente en el método guardarTodoEn, a aquellos métodos que declara el interfaz, pero no al resto de los métodos que posean las clases que lo implementan. Para poder acceder habría que realizar un casting al objeto apuntado.

Todo lo explicado anteriormente es igualmente aplicable al valor de retorno de un método.

Java es un lenguaje que incorpora **herencia simple** de implementación pero que puede aportar **herencia múltiple de interfaz**. Esto posibilita la herencia múltiple en el diseño de los programas Java.

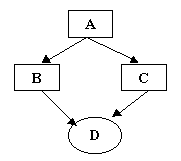
Una interfaz puede heredar de más de una interfaz antecesora.

interface InterfazMultiple extends Interfaz1, Interfaz2{ }

Una clase no puede tener más que una clase antecesora, pero puede implementar más de una interfaz:

class MiClase extends SuPadre implements Interfaz1, Interfaz2{ }

El ejemplo típico de herencia múltiple es el que se presenta con la herencia en diamante:

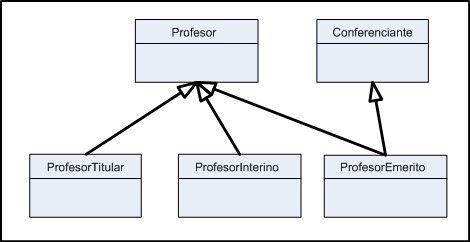


*Imagen 6: Ejemplo de herencia múltiple*

Para poder llevar a cabo un esquema como el anterior en Java es necesario que las clases A, B y C de la figura sean interfaces, y que la clase D sea una clase (que recibe la herencia múltiple):

interface A{ }  
interface B extends A{ }  
interface C extends A{ }  
class D implements B,C{ }

**Ejemplo**:



**Colisiones en la herencia múltiple**

En una herencia múltiple, los identificadores de algunos métodos o atributos pueden coincidir en la clase que implementa varios interfaces, si dos de las interfaces padres tienen algún método o atributo que coincida en nombre. A esto se le llama *colisión*.

Esto se dará cuando las interfaces padre (en el ejemplo anterior *B* y *C*) tienen un atributo o método que se llame igual. Java resuelve el problema estableciendo una serie de reglas.

Para la colisión de nombres de atributos, se obliga a especificar a qué interfaz base pertenecen al utilizarlos.

Para la colisión de nombres en métodos:

* Si tienen el mismo nombre y diferentes parámetros: se produce sobrecarga de métodos permitiendo que existan varias maneras de llamar al mismo.
* Si solo cambia el valor devuelto: se da un error de compilación, indicando que no se pueden implementar los dos.
* Si coinciden en su declaración: se elimina uno de los dos, con lo que solo queda uno.

### 7.4.7 Diferencias entre un interface y una clase abstracta

Un interface es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros dato constantes y otros no constantes.

Un interface es parecido a una clase abstracta en Java, pero con las siguientes diferencias:

- Todo método es abstracto y público sin necesidad de declararlo. Por lo tanto, un interface en Java no implementa ninguno de los métodos que declara.   
- Un interface se implementa (implements) no se extiende (extends) por sus subclases.  
- Una clase puede implementar más de un interfaz en Java, pero solo puede extender una clase. Es lo más parecido que tiene Java a la herencia múltiple, que de clases normales está prohibida.  
- Podemos declarar variables del tipo de clase del interfaz, pero para inicializarlas tendremos que hacerlo de una clase que lo implemente.

Así, por ejemplo, podemos declarar el siguiente interfaz en Java:

public interface Figura{   
 int area();  
}  
  
Y una clase que lo implementa:

public class Cuadrado implements Figura {

int lado;  
public Cuadrado (int ladoParametro) {  
 lado = ladoParametro;  
}

public int area(){  
 return lado\*lado;  
}

}

Más adelante podemos:

public class PruebaInterfaz{

public static void main(String args[]){  
  
 Figura figura=new Cuadrado (5);  
 // Podemos crear una referencia de interface(variable   
 // figura) y que un objeto

// que pertenezca a una clase que la implementa le sea   
// asignada a la variable  
System.out.println(figura.area());

}   
}

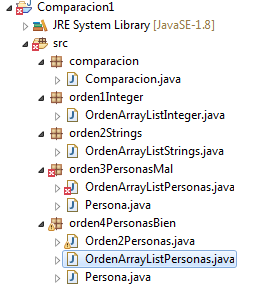
Una clase solamente puede derivar **extends** de una clase base, pero puede implementar varios interfaces. Los nombres de los interfaces se colocan separados por una coma después de la palabra reservada **implements**. El lenguaje Java no fuerza, por tanto, una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de su comportamiento similares.

## 7.5. Comparator - Comparable

|  |
| --- |
| **ORDENACIÓN UTILIZANDO LOS INTERFACES DE JAVA COMPARABLE Y COMPARATOR  – *este merece la pena leerlo 2ª eva*** |

### 7.5.1 Introducción

Vamos a ver algunos ejemplos para centrarnos en cuál es el problema que tenemos que resolver (Proyecto Comparacion1).



Recordemos cómo funciona el método compareTo, que hemos usado muchas veces (paquete **comparacion**):

|  |
| --- |
| public class Comparacion {  public static void main(String[] args){  Integer num = new Integer(6);  Int resul = num.compareTo(new Integer(10));  if (resul == 0)  System.out.println("Son iguales");  else if (resul > 0)  System.out.println("num es mayor");  else  System.out.println("num es menor");  resul = "adios".compareTo("Hola");    if (resul == 0)  System.out.println("Son iguales");  else if (resul > 0)  System.out.println("adios es mayor");  else  System.out.println("adios es menor");  }  } |

Salida:

num es menor

adios es mayor

Fíjate que la salida te dice si ponemos adiós que es mayor que Hola y si ponemos Adios nos saldría que es menor. Esto se debe a que en la tabla ASCII se colocan primero las mayúsculas y luego las minúsculas y ese el **orden natural** que usa compareTo.

Lo mismo que para los números, el 1 siempre será menor que el 2.

En el **ejemplo** del paquete **orden1Integer** vemos que la clase **Collections tiene un método estático llamado sort()**, al que pasándole por parámetro una colección, le decimos que la ordene. Si no especificamos el orden, el criterio es el mismo que el que utiliza el método compareTo() que ya conocemos (ya veremos cómo podemos especificar el orden).

En el **ejemplo** del paquete **orden2Strings** vemos cómo podemos ordenar también un ArrayList de Strings.

En estos dos ejemplos nos fijamos en que la forma de ordenar cuando queremos hacerlo al revés que con el orden natural es añadiendo al método sort un segundo parámetro:

**Collections.sort(arrayListInt, Collections.reverseOrder());**

Collections.reverseOrder() nos devuelve un objeto de tipo Comparator que nos permite ordenar la colección (o Array) de forma opuesta a la natural. Por lo que vemos que ahora el orden es descendente.

En caso de tener un Array haríamos lo siguiente para ordenar de las dos formas:

**Arrays.sort(a);**

**Arrays.sort(a, Collections.reverseOrder());**

En el **ejemplo** del paquete **orden3PersonasMal** vemos que tenemos un ArrayList de personas y que como tiene varios atributos en el objeto que queremos comparar, pues sort no sabe cómo tiene que ordenar (y no ordena, porque además da un error de compilación al intentar hacerlo).

### 7.5.2 [Uso del interface Comparable y método compareTo (API Java)](http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=587:ejercicio-ejemplo-resuelto-interface-comparable-y-metodo-compareto-api-java-comparar-objetos-cu00913c&catid=58:curso-lenguaje-programacion-java-nivel-avanzado-i&Itemid=180) para comparar y ordenar objetos y colecciones

Ya vimos con los algoritmos de ordenación cómo podemos ordenar Arrays o ArrayList (en general cualquier colección).

Veremos que no es necesario incluir en nuestro código esos métodos, ya que en Java tenemos la posibilidad de ordenar de una manera más sencilla. Es el propio Java quien ordenará por nosotros, tal y como vimos en los dos ejemplos de la Introducción. Eso sí, tenemos que entender en qué se basa esa ordenación para ser capaces de utilizar la de manera adecuada.

En el ejemplo del paquete **orden3PersonasMal,** ya no nos funciona porque no hemos indicado qué criterio usamos para decidir la comparación (se trata de una clase que tiene varios atributos ahora).

Vamos a ver cómo una clase debe hacer uso de la interfaz **Comparable**. Esto nos va a permitir que en dicha clase se pueda realizar la comparación de objetos, permitiendo hacer **ordenaciones** de los mismos. Seguimos con la clase Persona de antes:

|  |
| --- |
| public class **Persona** {  public int **nombre**, **edad**;  public Persona( int **nombre**, int edad) {  this.nombre = **nombre**;  this.edad = edad;  } } |

Lo primero que tenemos que pensar es que si queremos ordenar personas, tenemos que decidir cuál es el criterio para decidir que una persona es mayor, menor o igual a otra. En este caso podríamos decir que una persona es menor si tiene una edad menor, pero es una decisión que depende del caso en concreto. También podría ser válido si quisiéramos hacer un listado de personas ordenar por el nombre para hacer un listado alfabético.

Vemos ahora el paquete **orden4PersonasBien:**

En **Orden2Personas.java** vemos que lo primero es implementar el interfaz Comparable:

|  |
| --- |
| **public class Persona implements Comparable<Persona>** |

La base para realizar la ordenación va a ser la **interfaz Comparable**, en este primer ejemplo.

|  |
| --- |
| public interface Comparable<T>  **Que tiene el método:**  int compareTo([T](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html" \t "type parameter in Comparable) o) |

¿Qué significa la <T>?. Representa que se puede aplicar a cualquier Tipo de Objeto, y que hay que especificarlo como hemos visto antes.

Vamos a implementar la interfaz Comparable en la clase que representa los objetos que queremos ordenar. Por ello, estamos obligados a implementar (sobreescribir) el método **de la interfaz**:

|  |
| --- |
| **@Override public int compareTo (Persona o)** |

Este método debe devolver un número negativo, cero o un número positivo en función de que el objeto que comparemos con el objeto “o” de referencia sea menor, igual o mayor respectivamente que ese objeto de referencia.

Por tanto, vamos a incluir el método **public int compareTo(Persona o)** en nuestra clase **Persona**.

**@Override**

**public int compareTo(Persona o) {**

**return this.nombre.compareTo(o.nombre);**

**}**

En el main de **Orden2Personas.java** vemos:

**Persona p1 = new Persona("Pepe", 28);**

**Persona p2 = new Persona("Juan", 32);**

**Persona p3 = new Persona("Juan", 32);**

**System.out.println(p1.compareTo(p2));**

**System.out.println(p2.compareTo(p1));**

**System.out.println(p2.compareTo(p3));**

La salida será:

6

-6

0

Esa salida refleja por ejemplo, que Pepe es mayor que Juan, ya que está después en el alfabeto. En concreto, lo que refleja es que hay 6 posiciones de diferencia en el alfabeto.

Nos devuelve esto porque hemos programado en nuestro método **compareTo** que compare así el atributo nombre de las personas.

En el mismo paquete, tenemos **OrdenArrayListPersonas.java,** en el que ahora lo que tenemos es una colección de Personas. El método estático **sort** de la clase **Collections** nos va a ordenar dicha colección en base al método compareTo que hemos programado. Ejecútalo y observa la salida.

**1.-** ¿Cómo hacemos si queremos la ordenación alfabéticamente en orden descendente?

**2.-** ¿Cómo hacemos si queremos la ordenación por edad en orden ascendente?

**3.-** ¿Cómo hacemos si queremos la ordenación por edad en orden descendente?

|  |
| --- |
| **RESUMEN**  Si tenemos 2 Personas y queremos compararlas u ordenarlas, la pregunta a hacerse sería:  **“cuándo una persona es mayor que otra o cuándo es menor, o cuándo son iguales”,** atendiendo a algún tipo de atributo. Después sobreescribimos compareTo de acuerdo al criterio establecido. |

Cuando en lugar de comparar explícitamente 2 personas lo que queremos es que se ordene una colección, lo que hay que hacer es sobreescribir el método compareTo, que es el que aplica el método sort para ser capaz de ordenar una colección o un array. Esto es lo que hemos visto en **OrdenArrayListPersonas.java,** cuando hace:

**Collections.sort(personas);**

En el proyecto **Comparacion2**, podemos ver un ejemplo (paquete ordenArrayPersonas) de cómo se puede hacer una ordenación en un array:

**Arrays.sort(arrayPersonas);**

En el proyecto **Comparacion3**, vemos otro ejemplo (paquete ordenArrayLPersonas) de cómo se puede hacer una ordenación en un ArrayList:

En este caso la clase Persona quedaría de la siguiente manera:

public class Persona **implements Comparable<Persona>**{

    public int dni, edad;

    public Persona( int d, int e){

    this.dni = d;

    this.edad = e;

}

Como podemos ver resaltado hemos incluido también el código **implements Comparable<Persona>**, que nos indica que vamos a implementar en la clase la interfaz Comparable para el parámetro Persona.

También hemos incluido el código necesario para implementar el método **public** **int** **compareTo(Persona o), donde escribiremos el criterio para comparar personas. Si consideramos que el orden será ascendente por dni, tenemos que escribir:**

**@Override**

**public int compareTo(Persona o) {**

**return (new Integer(this.dni).compareTo(new Integer(o.dni)));**

**}**

O también:

**@Override**

**public int compareTo(Persona o) {**

**if (this.dni > o.dni)**

**return 1;**

**else if (this.dni < o.dni)**

**return -1;**

**else**

**return 0;**

**}**

Date cuenta que **lo que se compara en compareTo deben ser objetos**.

Ahora escribimos un programa que generará 2 Personas y las compara mostrando cuál es la relación de comparación entre ellas (en base a los dnis).

public class **Programa** {

  public static void main(String arg[]) {

    Persona **p1** = new **Persona**(74999999, 35);

    Persona **p2** = new **Persona**(72759474, 30);

if (p1.**compareTo**(p2) < 0 ) {   
 System.out.println("La persona p1: es menor.");   
 }

    else if ( p1.**compareTo**(p2) > 0 ) {  
 System.out.println("La persona p1: es mayor.");   
 }

     else {   
 System.out.println ("La persona p1 es igual   
 a la persona p2");   
 }

  }

}

**Comprueba con varios casos si funciona bien. Si queremos ordenar un ArrayList, vemos que funcionaría perfectamente:**

**ArrayList<Persona> personas = new ArrayList<Persona>();**

**personas.add(new Persona(74999999, 35));**

**personas.add(new Persona(72759474, 30));**

**personas.add(new Persona(11111111, 35));**

**personas.add(new Persona(22222222, 30));**

**personas.add(new Persona(33333333, 35));**

**personas.add(new Persona(44444444, 30));**

**personas.add(new Persona(55555555, 35));**

**personas.add(new Persona(66666666, 30));**

**// Ahora sí nos deja ordenar personas**

**Collections.sort(personas);**

Comprueba la salida.

Si queremos ordenar descendentemente por dni:

**Collections.sort(personas, Collections.reverseOrder());**

El reverseOrder() lo hace según lo que hayamos escrito en el método sobreescrito compareTo(),

Otra forma de ordenar descendentemente, sin usar reverseOrder() sería dar la vuelta a nuestra comparación:

**@Override**

**public int compareTo(Persona o) {**

**return (new Integer(o.dni).compareTo(new Integer(this.dni)));**

**// hemos intercambiado el orden de comparación**

**}**

Esta **implementación de la interfaz Comparable** es muy habitual ya que en general muchas veces vamos a desear que nuestras clases sean comparables no solo por el hecho de poder comparar objetos de dicha clase, sino por el uso de poder ordenar objetos de esa clase. La **ordenación** es una de las funciones más importantes y necesarias en el ámbito de la informática, ya que tener la información ordenada hace mucho más rápidas las consultas o modificaciones de datos.

¿Por qué crees que se usa la implementación de **interfaces** en Java?

Hay varios motivos, pero uno principal es que mediante la implementación de interfaces todos los programadores usan el mismo nombre de método y estructura formal para comparar objetos (o clonar, u otras operaciones).

Imagínate que estás trabajando en un equipo de programadores y tienes necesidad de utilizar una clase que ha codificado otro programador. Gracias a que se implementan interfaces Java, no necesitarás estudiar el código para ver qué método hay que invocar para comparar objetos de esa clase. Dado que todos los programadores usan la implementación de interfaces, sabemos que la comparación de objetos (o clonación, etc.) debe hacerse invocando determinados métodos de determinada forma. Esto facilita el desarrollo de programas y la comprensión de los mismos, especialmente cuando hablamos de programas o desarrollos donde pueden intervenir cientos o miles de clases diferentes.

En el mismo caso de la clase Persona, vamos a cambiar el criterio de ordenación:

Ahora vamos a considerar que ordenamos a las personas por su edad, pero en el caso de que dos personas tengan la misma edad, vamos a considerar que es menor la que tenga un número de DNI más bajo. Para este caso debes introducir más personas que cumplan estas condiciones, con la misma edad.

@Override

**public int compareTo(Persona o) {**

**if (this.edad > o.edad)**

**return 1;**

**else if (this.edad < o.edad)**

**return -1;**

**else if (this.dni > o.dni)**

**return 1;**

**else if (this.dni< o.dni)**

**return -1;**

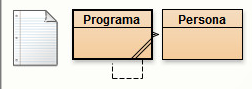
**else**

**return 0;**

**}**

Si cambiamos el criterio para que se ordene el ArrayList descendentemente por edad y ascendentemente por DNI ¿Cómo se haría?

El diagrama de clases de este ejemplo sería:



### 7.5.3 [Interface Comparator (API java). Diferencias con Comparable. Clase Collections. Código ejemplo](http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=599:interface-comparator-api-java-diferencias-con-comparable-clase-collections-codigo-ejemplo-cu00918c&catid=58:curso-lenguaje-programacion-java-nivel-avanzado-i&Itemid=180)

**INTERFACE COMPARATOR**

A continuación procederemos a describir de manera detallada el uso y el comportamiento de la **interface Comparator** del api de Java.

Además veremos un ejemplo de uso de esta interfaz y cómo se utiliza. Señalaremos también cuáles son las **diferencias entre la interface Comparator y la interface Comparable**, dos interfaces que tienen ciertas similitudes y ciertas diferencias.

**COMPARATOR**

Esta interfaz es la encargada de permitirnos el poder comparar 2 elementos en una colección. Por tanto pudiera parecer que es igual a la interfaz **Comparable** (recordemos que esta interfaz nos obligaba a implementar el método **compareTo** (Object o)) que hemos visto anteriormente en el paquete java.lang.

Su diferencia es:

|  |
| --- |
| Mientras que **Comparable** nos obliga a implementar el método:  **compareTo** (Object o)  la interfaz **Comparator** nos obliga a implementar el método:  **compare** (Object o1, Object o2) |

Si, por ejemplo para la clase Persona quisiéramos ordenar a las Personas por su altura, o por su nombre en orden alfabético podríamos recurrir a la interfaz **Comparator** para implementar el método **compare** (Object o1, Object o2) definiendo el método deseado.

**EJERCICIO RESUELTO**

Ahora vamos a desarrollar un ejercicio para comprender mejor su funcionamiento. Vamos a suponer una clase **Persona** similar a la vista en anteriores ocasiones sobre la cual vamos a definir un orden para poder ordenar colecciones de ese elemento.

Al igual que antes, usaremos la clase ArrayList para contener una colección de objetos.

Lo primero será definir mediante código nuestra clase Persona que será la siguiente:

/\* Ejemplo Interface Comparable\*/  
  
public class Persona **implements Comparable<Persona>** {  
    private int idPersona;  
    private String nombre;  
    private int altura;  
  
    public Persona (int idPersona, String nombre, int altura) {  
        this.idPersona = idPersona;  
        this.nombre = nombre;  
        this.altura = altura;}  
  
    @Override  
    public String toString() {  
        return "Persona-> ID: " +idPersona   
 +" Nombre: " +nombre  
 +"Altura: "+altura;  
 }  
  
    @Override  
    public int compareTo(Persona o) {  
        return this.nombre.compareTo(o.nombre);  
 }

public int getIdPersona() {return idPersona;}  
    public String getNombre() {return nombre;}  
    public int getAltura() {return altura;}  
}

En ella podemos ver que cada objeto de la clase Persona tendrá como campos un idPersona, nombre y altura. No hemos detallado los métodos set correspondientes a cada propiedad porque trataremos de centrarnos en el código que resulta de interés para este caso concreto.

Observamos también que queremos que esta clase sea ordenada naturalmente como ya hemos visto anteriormente al implementar Comparable. En este caso la ordenación se basa en el campo nombre de las Personas como queda reflejado en el método compareTo. Es decir, cuando se trate de realizar una ordenación “natural”, esta se hará por orden alfabético de nombres.

Las colecciones de Persona se ordenarán por este método siempre que se invoque una forma de ordenación que use la ordenación natural. Como ejemplo tenemos el siguiente código:

import java.util.ArrayList;

import java.util.Collections;

public class Programa {

    public static void main(String arg[]) {

        ArrayList<Persona> listaPersonas = new ArrayList<Persona>();

        listaPersonas.add(new Persona(1,"Maria", 185));

        listaPersonas.add(new Persona(2,"Carla", 190));

        listaPersonas.add(new Persona(3,"Yovana", 170));

Collections.sort(listaPersonas);  // Ejemplo uso ordenación natural

        System.out.println("Personas Ordenadas por orden natural: "+listaPersonas);

      }

}

La lista de Personas ha sido ordenada usando la invocación Collections.sort pasando como parámetro la colección y el orden utilizado en esta invocación es el orden natural definido en su clase, es decir, por el nombre de cada Persona. En este caso: Carla, María y Yovana de acuerdo al orden alfabético C, M, Y.

La clase **Collections** que hemos utilizado es una clase similar a la clase **Arrays** del api de Java. Puede ser invocada simplemente importándola y nos aporta distintos métodos estáticos, siendo únicamente necesario pasarle determinados parámetros para obtener un resultado.

Ahora bien, imaginemos que queremos ordenar a las Personas por un orden distinto a lo que hemos denominado orden natural, en este caso hablaremos de orden total. Como orden total podríamos elegir la altura. Una forma sería modificar el método compareTo, como hacíamos en el apartado anterior.

Y otra forma consiste (y a partir de aquí es lo nuevo) en hacer uso de la **interfaz Comparator** de la siguiente manera:

|  |
| --- |
| Creamos una clase diferente para **implementar la interfaz Comparator**, donde **sobreescribimos el método compare()** y en la clase Persona ya no implementaremos la interfaz Comparable. |

/\* Ejemplo Interface Comparator \*/  
  
import java.util.Comparator;  
  
public class OrdenarPersonaPorAltura **implements Comparator<Persona>** {  
    @Override  
    public int compare(Persona o1, Persona o2) {  
    return o1.getAltura() - o2.getAltura();

// Devuelve un entero positivo si la altura de o1 es mayor que la de o2  
    }  
}

Hemos definido el orden así, si o1 < o2 en este caso queremos que o1 vaya delante de o2 (decimos que es “más pequeño) por ordenarlo de menor altura a mayor altura. Entonces debemos devolver un número negativo, 0 si son iguales y un número positivo si o2 es de menor altura.

El siguiente paso es utilizar la clase OrdenarPersonaPorAltura de la siguiente manera en el programa principal:

public class Programa {

    public static void main(String arg[]) {

        ArrayList<Persona> listaPersonas = new ArrayList<>();

        listaPersonas.add(new Persona(1,"Maria",185));

        listaPersonas.add(new Persona(2,"Carla",190));

        listaPersonas.add(new Persona(3,"Yovana",170));

        Collections.sort(listaPersonas,   
 new **OrdenarPersonaPorAltura()**);

System.out.println("Personas Ordenadas por orden total: "

+ listaPersonas);

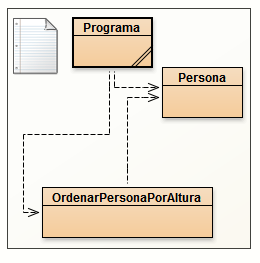
      }

}

La única observación con respecto al anterior programa es que ahora la sentencia de ordenación Collections.sort lleva añadido el parámetro **new OrdenarPersonaPorAltura().**

**Esta es otra forma del método sort de la clase Collections, en este caso pasándole como parámetros un objeto colección y un objeto que implemente Comparator y defina un orden a utilizar.** Al pasarle un objeto de tipo OrdenarPersonaPorAltura indicamos el orden total a utilizar, que en este caso es la altura.

El diagrama de clases de este ejemplo es:



**CONCLUSIONES**

La **primera conclusión** puede ser que aunque **Comparable** y **Comparator** parecen iguales, en realidad no lo son, y mientras una se define para el orden natural, la otra se define para un orden total.

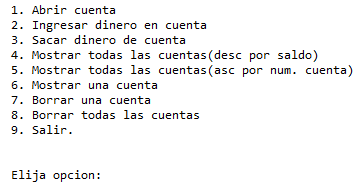
El orden natural es utilizado por diversos métodos del api de Java, pero no siempre hemos de usar el orden natural. En determinados casos podremos querer ordenar objetos por un orden distinto al orden natural, y para ello nos será útil implementar la interface Comparator. Aunque vimos que dando la vuelta al método compareTo, somos capaces también de ordenar sin usar la interfaz Comparator.

La **segunda conclusión** es que gracias a la interfaz **Comparator**, podemos ordenar muy fácilmente colecciones utilizando clases que implementen el método compare **por cada tipo de ordenación que deseemos. Cosa que no podíamos hacer con la interfaz Comparable, ordenamos en el mismo programa solo de una forma, obviamente no es posible sobreescribir el método compareTo en la misma clase más de una vez.**

Prueba detenidamente los proyectos Jugadores1..Jugadores4 para terminar de comprender todo lo explicado.

En el apartado del Aula Virtual de **Excepciones, hicimos un ejercicio con Cuentas de bancos, en el que no desarrollasteis la pregunta:**

“Crea una **segunda versión** en la que las cuentas se puedan ver de dos formas distintas, ascendentemente por número de cuenta y descendentemente por saldo:”



Lo que os pido ahora es que completéis el ejercicio con esas opciones y lo subáis al Aula virtual antes del jueves.

## 7.6. Interfaces en Java. Ejemplos de Interfaces.

Antes de Java 8 una interface era considerada como una **clase abstracta pura:** todos sus métodos son abstractos y si tiene atributos son todos constantes.

A partir de Java 8 el concepto de Interface ha cambiado. Podemos considerar una interface como una **clase abstracta**que sólo puede contener constantes, métodos abstractos, métodos por defecto, métodos estáticos y tipos anidados.

Se crean utilizando la palabra clave **interface** en lugar de class.

[public] **interface** NombreInterface [extends Interface1, Interface2, …]{

      [métodos abstractos]

      [métodos default]

      [métodos static]

      [tipos anidados]

      [atributos constantes]

}

La interface puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases. Si tiene el modificador public el archivo .java que la contiene debe tener el mismo nombre que la interfaz.

Igual que las clases, al compilar el archivo .java de la interface se genera un archivo .class

Las interfaces no pueden ser instanciadas, solo pueden ser implementadas por clases o extendidas por otras interfaces.

A diferencia de las clases que pueden heredar de una sola clase, las interfaces en Java pueden heredar de varias interfaces.

**Todos los métodos que aparecen en la interfaz son públicos** por lo que se puede omitir el modificador de acceso *public*.

**En los métodos abstractos**no es necesario escribir *abstract.*

**Los métodos por defecto** se especifican mediante el modificador *default*.

**Los métodos estáticos** se especifican mediante la palabra reservada *static*.

**Todos los atributos son constates públicos y estáticos**. Por lo tanto, se pueden omitir los modificadores *public static final* cuando se declara el atributo. Se deben inicializar en la misma instrucción de declaración.

Los nombres de las interface suelen acabar en *able* aunque no es necesario: configurable, arrancable, dibujable, comparable, clonable, etc.

Las interfaces juegan un papel fundamental en la creación de aplicaciones Java:

- Las interfaces definen un protocolo de comportamiento y proporcionan un formato común para implementarlo en las clases.

- Utilizando interfaces es posible que clases no relacionadas, situadas en distintas jerarquías de clases sin relaciones de herencia, tengan comportamientos comunes.

**Ejemplo**:

//Interfaz que define relaciones de orden entre objetos.

public interface Relacionable {

     boolean esMayorQue(Relacionable a);

     boolean esMenorQue(Relacionable a);

     boolean esIgualQue(Relacionable a);

}

**IMPLEMENTACIÓN**

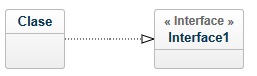
Para indicar que una clase implementa una interface se utiliza la palabra clave **implements**.

public class *UnaClase* implements *Interface1*{

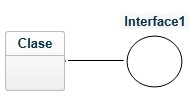
……

}

En **UML** una clase que implementa una interface se representa mediante una flecha con línea discontinua apuntando a la interface:

[](https://4.bp.blogspot.com/-xwxYgV2ucZ0/WOKA1mh8UCI/AAAAAAAAAhg/HUNJVOnr8-AlHXkj1aTP7h4-AYyoyNiTgCLcB/s1600/Clase_implements_Interface.jpg)

O también se puede representar de forma abreviada:

[](https://1.bp.blogspot.com/-uwsSeHOEFEI/WOKBA_Ou6OI/AAAAAAAAAhk/mAGnEHppL04NCfHNJKRdXFxGHQdcWlDDACLcB/s1600/implementsAbreviado.jpg)

Las clases que implementan una interfaz deben **implementar** **todos los métodos abstractos**. De lo contrario serán clases abstractas y deberán declararse como tal.

**Ejemplo:** La clase Fraccion implementa la interfaz Relacionable. En ese caso se dice que la clase **Fraccion es Relacionable**

[](https://1.bp.blogspot.com/-dyOm1uyGpZw/WOKBQSCgV-I/AAAAAAAAAho/l9FBY4sL7YoS3ovlbE1m9ochACNyPsyYACLcB/s1600/ClaseRelacionable.jpg)

public class Fraccion implements Relacionable {

    private int num;

    private int den;

    public Fraccion() {

        this.num = 0;

        this.den = 1;

    }

    public Fraccion(int num, int den) {

        this.num = num;

        this.den = den;

        simplificar();

    }

    public Fraccion(int num) {

        this.num = num;

        this.den = 1;

    }

    public void setDen(int den) {

        this.den = den;

        this.simplificar();

    }

    public void setNum(int num) {

        this.num = num;

        this.simplificar();

    }

    public int getDen() {

        return den;

    }

    public int getNum() {

        return num;

    }

    //sumar fracciones

    public Fraccion sumar(Fraccion f) {

        Fraccion aux = new Fraccion();

        aux.num = num \* f.den + den \* f.num;

        aux.den = den \* f.den;

        aux.simplificar();

        return aux;

    }

    //restar fracciones

    public Fraccion restar(Fraccion f) {

        Fraccion aux = new Fraccion();

        aux.num = num \* f.den - den \* f.num;

        aux.den = den \* f.den;

        aux.simplificar();

        return aux;

    }

    //multiplicar fracciones

    public Fraccion multiplicar(Fraccion f) {

        Fraccion aux = new Fraccion();

        aux.num = num \* f.num;

        aux.den = den \* f.den;

        aux.simplificar();

        return aux;

    }

    //dividir fracciones

    public Fraccion dividir(Fraccion f) {

        Fraccion aux = new Fraccion();

        aux.num = num \* f.den;

        aux.den = den \* f.num;

        aux.simplificar();

        return aux;

    }

    //Cálculo del máximo común divisor por el algoritmo de Euclides

    private int mcd() {

        int u = Math.abs(num); //valor absoluto del numerador

        int v = Math.abs(den); //valor absoluto del denominador

        if (v == 0) {

            return u;

        }

        int r;

        while (v != 0) {

            r = u % v;

            u = v;

            v = r;

        }

        return u;

    }

    private void simplificar() {

        int n = mcd(); //se calcula el mcd de la fracción

        num = num / n;

        den = den / n;

    }

    @Override

    public String toString() {   
 //Sobreescritura del método toString heredado de Object

        simplificar();

        return num + "/" + den;

    }

    //Implementación del método abstracto de la interface

    @Override

    public boolean esMayorQue(Relacionable a) {

        if (a == null) {

            return false;

        }

        if (!(a instanceof Fraccion)) {

            return false;

        }

        Fraccion f = (Fraccion) a;

        this.simplificar();

        f.simplificar();

        if ((num / (double) den) <= (f.num / (double) f.den)) {

            return false;

        }

        return true;

    }

    //Implementación del método abstracto de la interface

    @Override

    public boolean esMenorQue(Relacionable a) {

        if (a == null) {

            return false;

        }

        if (!(a instanceof Fraccion)) {

            return false;

        }

        Fraccion f = (Fraccion) a;

        this.simplificar();

        f.simplificar();

        if ((num / (double) den) >= (f.num / (double) f.den)) {

            return false;

        }

        return true;

    }

    //Implementación del método abstracto de la interface

    @Override

    public boolean esIgualQue(Relacionable a) {

        if (a == null) {

            return false;

        }

        if (!(a instanceof Fraccion)) {

            return false;

        }

        Fraccion f = (Fraccion) a;

        this.simplificar();

        f.simplificar();

        if (num != f.num) {

            return false;

        }

        if (den != f.den) {

            return false;

        }

        return true;

    }

}

Ejemplo de utilización de la clase Fraccion:

public static void main(String[] args) {

        //Creamos dos fracciones y mostramos cuál es la mayor y //cuál menor.

        Fraccion f1 = new Fraccion(3, 5);

        Fraccion f2 = new Fraccion(2, 8);

        if (f1.esMayorQue(f2)) {

            System.out.println(f1 + " > " + f2);

        } else {

            System.out.println(f1 + " <= " + f2);

        }

        //Creamos un ArrayList de fracciones y las mostramos //ordenadas de menor a mayor

        ArrayList<Fraccion> fracciones = new ArrayList();

        fracciones.add(new Fraccion(10, 7));

        fracciones.add(new Fraccion(-2, 3));

        fracciones.add(new Fraccion(1, 9));

        fracciones.add(new Fraccion(6, 25));

        fracciones.add(new Fraccion(3, 8));

        fracciones.add(new Fraccion(8, 3));

         Collections.sort(fracciones, new Comparator<Fraccion>(){

            @Override

            public int compare(Fraccion o1, Fraccion o2) {

                if(o1.esMayorQue(o2)){

                    return 1;

                }else if(o1.esMenorQue(o2)){

                    return -1;

                }else{

                    return 0;

                }

            }

        });

        System.out.println("Fracciones ordenadas de menor a mayor");

        for(Fraccion f: fracciones){

            System.out.print(f + " ");

        }

}

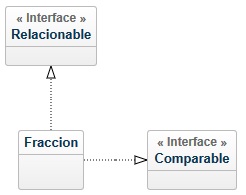
Una clase puede implementar más de una interface. Los nombres de las interfaces se escriben a continuación de *implements* y separadas por comas:

public class *UnaClase* implements *Interface1*, *Interface2*, Inte*r*face3{

……….

}

En el ejemplo, para ordenar las fracciones hemos utilizado un Comparator como parámetro de Collections.sort. También podríamos ordenarlas haciendo que la clase Fraccion implemente además la interfaz Comparable:

[](https://3.bp.blogspot.com/-_hGC1ZG6ouc/WOKCXdW2TaI/AAAAAAAAAhs/jC7TlQkR5LgNkjcja-m6g_CmQ8uJFtkLACLcB/s1600/Clase_Java_implemanta_varias_interfaces.jpg)

public class Fraccion implements Relacionable, Comparable<Fraccion> {

         ……

         //Código de la clase Fraccion

         ……

         //Añadimos a la clase el método compareTo

       @Override

       public int compareTo(Fraccion o) {

          if(this.esMenorQue(o)){

              return -1;

          }else if(this.esMayorQue(o)){

              return 1;

          }else{

              return 0;

          }

    }

} //fin de la clase Fraccion

De este modo para ordenar escribiríamos: Collections.sort(fracciones)

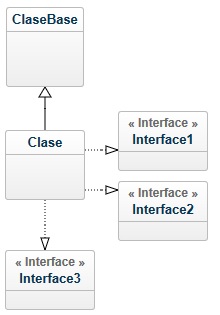
Una clase solo puede tener una clase base pero puede implementar múltiples interfaces. El lenguaje Java no permite herencia múltiple entre clases, pero las interfaces proporcionan una alternativa para implementar algo parecido a la herencia múltiple de otros lenguajes.

Una clase que además de implementar interfaces herede de otra se declarará de esta forma:

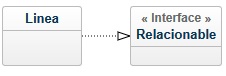
public class *UnaClase* extends *ClaseBase* implements *Interface1*, *Interface2*, Inte*r*face3{

……….

}

[](https://1.bp.blogspot.com/-mpJ2wCmANi8/WOKCzi7uHXI/AAAAAAAAAhw/MA2nOKq7KdMEJoq5pii_uhWheVWiLBkzgCLcB/s1600/Interface_Java_Multiple_Implements.jpg)

Una interfaz la puede implementar cualquier clase. Por ejemplo, podemos tener una clase Linea que también implementa la interfaz Relacionable:

[](https://1.bp.blogspot.com/-raxs1F8JUJY/WOKDBYQapoI/AAAAAAAAAh0/IahbKSEzNJEBoSQs_mm3DG5s5WCdz5uEwCLcB/s1600/LineaRelacionable.jpg)

public class Linea implements Relacionable {

    private double x1;

    private double y1;

    private double x2;

    private double y2;

    public Linea(double x1, double y1, double x2, double y2) {

        this.x1 = x1;

        this.y1 = y1;

        this.x2 = x2;

        this.y2 = y2;

    }

    public double longitud() {

        double l = Math.sqrt((x2 - x1) \* (x2 - x1) +   
(y2 - y1) \* (y2 - y1));

        return l;

    }

    //Implementación del método abstracto de la interface

    @Override

    public boolean esMayorQue(Relacionable a) {

        if (a == null) {

            return false;

        }

        if (!(a instanceof Linea)) {

            return false;

        }

        Linea linea = (Linea) a;

        return this.longitud() > linea.longitud();

    }

    //Implementación del método abstracto de la interface

    @Override

    public boolean esMenorQue(Relacionable a) {

        if (a == null) {

            return false;

        }

        if (!(a instanceof Linea)) {

            return false;

        }

        Linea linea = (Linea) a;

        return this.longitud() < linea.longitud();

    }

    //Implementación del método abstracto de la interface

    @Override

    public boolean esIgualQue(Relacionable a) {

        if (a == null) {

            return false;

        }

        if (!(a instanceof Linea)) {

            return false;

        }

        Linea linea = (Linea) a;

        return this.longitud() == linea.longitud();

    }

    //Sobreescritura del método toString heredado de Object

    @Override

    public String toString() {

        StringBuilder sb = new StringBuilder();

        sb.append("Coordenadas inicio linea: ");

        sb.append(x1);

        sb.append(" , ");

        sb.append(y1);

        sb.append("\nCoordenadas final linea: ");

        sb.append(x2);

        sb.append(" , ");

        sb.append(y2);

        sb.append("\nLongitud: ");

        sb.append(longitud());

        return sb.toString();

    }

}

Podemos hacer un ejemplo de utilización de la clase Linea similar al que hemos hecho para la clase Fracción:

public static void main(String[] args) {

        Linea l1 = new Linea(2, 2, 4, 1);

        Linea l2 = new Linea(5, 2, 10, 8);

        if (l1.esMayorQue(l2)) {

            System.out.println(l1 + "\nes mayor que" + l2);

        } else {

            System.out.println(l1 + "\nes menor o igual que" + l2);

        }

        ArrayList<Linea> lineas = new ArrayList();

        lineas.add(new Linea(0, 7, 1, 4));

        lineas.add(new Linea(2, -1, 3, 5));

        lineas.add(new Linea(1, 9, 0, -3));

        lineas.add(new Linea(15, 3, 9, 5));

        Collections.sort(lineas, new Comparator<Linea>(){

            @Override

            public int compare(Linea o1, Linea o2) {

                if(o1.esMayorQue(o2)){

                    return 1;

                }else if(o1.esMenorQue(o2)){

                    return -1;

                }else{

                    return 0;

                }

            }

        });

        System.out.println("\nLineas ordenadas por longitud de menor a mayor");

        for (Linea l : lineas) {

            System.out.println(l);

        }

}

**INTERFACES Y POLIMORFISMO**

La definición de una interface implica una definición de un nuevo tipo de referencia y por ello **se puede usar el nombre de la interface como nombre de tipo**.

El nombre de una interfaz se puede utilizar en cualquier lugar donde pueda aparecer el nombre de un tipo de datos.

Si se define una variable cuyo tipo es una interface, se le puede asignar un objeto instancia de una clase que implementa la interface.

Volviendo al ejemplo, las clases Linea y Fraccion implementan la interfaz Relacionable. Podemos escribir las instrucciones:

Relacionable r1 = new Linea(2,2,4,1);

Relacionable r2 = new Fraccion(4,7);

En Java, utilizando interfaces como tipo se puede aplicar el polimorfismo para clases que no están relacionadas por herencia.

Por ejemplo, podemos escribir:

System.out.println(r1);  //ejecuta toString de Linea

System.out.println(r2);  //ejecuta toString de Fraccion

También podemos crear un array de tipo Relacionable y guardar objetos de clases que  implementan la interfaz.

        ArrayList<Relacionable> array = new ArrayList();

        array.add(new Linea(15, 3, 9, 5));

        array.add(new Fraccion(10, 7));

        array.add(new Fraccion(6, 25));

        array.add(new Linea(3, 4, 10, 15));

        array.add(new Fraccion(8, 3));

        array.add(new Linea(0, 7, 1, 4));

        array.add(new Linea(2, -1, 3, 5));

        array.add(new Fraccion(1, 9));

        array.add(new Linea(1, 9, 0, -3));

        array.add(new Fraccion(3, 8));

        array.add(new Fraccion(-2, 3));

        for (Relacionable r : array) {

            System.out.println(r);

        }

En este caso dos clases no relacionadas, *Linea* y *Fraccion*, por implementar la misma interfaz *Relacionable* podemos manejarlas a través de referencias a la interfaz y aplicar polimorfismo.

**MÉTODOS DEFAULT**

A partir de Java 8 las interfaces además de métodos abstractos pueden contener métodos por defecto o métodos default.

En la interfaz se escribe el código del método. Este método estará disponible para todas las clases que la implementen, no estando obligadas a escribir su código. Solo lo incluirán en el caso de querer modificarlo.

**De este modo, si se modifica una interfaz añadiéndole una nueva funcionalidad, se evita tener que modificar el código en todas las clases que la implementan.**

Ejemplo: Vamos a añadir un nuevo método a la interfaz Relacionable que devuelva el nombre de la clase (String) del objeto que la está utilizando. Si lo añadimos como abstracto tendremos que modificar las clases Linea y Fraccion y añadir en cada una el nuevo método. En lugar de esto vamos a crear el método como default y de este modo las clases Linea y Fraccion lo pueden usar sin necesidad de escribirlo.

public interface Relacionable {

    boolean esMayorQue(Relacionable a);

    boolean esMenorQue(Relacionable a);

    boolean esIgualQue(Relacionable a);

**default** String nombreClase(){ //método por defecto

        String clase = getClass().toString();

        int posicion = clase.lastIndexOf(".");

        return clase.substring(posicion+1);

    }

}

Ejemplo de uso:

       ArrayList<Relacionable> array = new ArrayList();

        array.add(new Linea(15, 3, 9, 5));

        array.add(new Fraccion(10, 7));

        array.add(new Fraccion(6, 25));

        array.add(new Linea(3, 4, 10, 15));

        array.add(new Fraccion(8, 3));

        array.add(new Linea(0, 7, 1, 4));

        array.add(new Linea(2, -1, 3, 5));

        array.add(new Fraccion(1, 9));

        array.add(new Linea(1, 9, 0, -3));

        array.add(new Fraccion(3, 8));

        array.add(new Fraccion(-2, 3));

        for (Relacionable r : array) {

            System.out.println(r.nombreClase());

**//usamos el método por defecto**

            System.out.println(r);

            System.out.println();

        }

Puede darse el caso de una clase que implemente varias interfaces y las interfaces contengan métodos default iguales (mismo nombre y lista de parámetros).

Ejemplo: La interface *Modificable* contiene dos métodos abstractos y un método default *nombreClase* que es igual que el contenido en la interface Relacionable:

public interface Modificable{

    void aumentar(int n);

    void disminuir(int n);

    default String nombreClase(){

        String clase = getClass().toString();

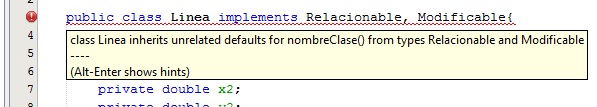
        int posicion = clase.lastIndexOf(".");

        return clase.substring(posicion+1);

    }

}

Supongamos que la clase Linea implementa las dos interfaces. En la clase Linea se debe escribir el código de los dos métodos abstractos, pero el método default provoca un error de compilación:

[](https://4.bp.blogspot.com/-AGNYP5oYjYY/WOKEcYjkKzI/AAAAAAAAAh4/dR5ghVuDY58DslAe7aei1MseD2vh2V9LACEw/s1600/Java_Interfaces_Class_inherits_Unrelated_Default.jpg)

La clase Linea *hereda* dos métodos default iguales. Esto provoca un *conflicto* que tenemos que resolver. Lo podemos hacer de dos formas. La primera forma es redefinir el método nombreClase() en la clase Linea:

public class Linea implements Relacionable, Modificable{

    ……

    //Código de la clase Linea

    ……

    @Override

    public void aumentar(int n) {

        this.x1+=n;

        this.y1+=n;

        this.x2+=n;

        this.y2+=n;

    }

    @Override

    public void disminuir(int n) {

        this.x1-=n;

        this.y1-=n;

        this.x2-=n;

        this.y2-=n;

    }

    //Redefinición del método default

    @Override

    public String nombreClase(){

        String clase = getClass().toString();

        int posicion = clase.lastIndexOf(".");

        return clase.substring(posicion+1);

    }

}//Final de la clase Linea

En este caso no se ha modificado el código del método aunque podríamos haberlo hecho. Nos hemos limitado a copiar y pegar el código cambiando el modificador *default* por *public*.

Para estos casos en los que no es necesario cambiar el método podemos usar la segunda forma de resolver el conflicto: indicar cuál de los dos métodos se tiene que ejecutar.

public class Linea implements Relacionable, Modificable{

    ……

    //Código de la clase Linea

    ……

    //Redefinición del método default

    //Se ejecutará el método de la interface Relacionable

    @Override

    public String nombreClase(){

        return Relacionable.super.nombreClase();

    }

}

La forma de indicar el método de la interface que se ejecuta es:

*nombreInterface.super.metodo*

**MÉTODOS STATIC**

A partir de Java 8 las interfaces también pueden contener métodos static. En la interfaz se escribe el código del método. Los métodos static NO pueden ser redefinidos en las clases que la implementan.

Para utilizarlos se escribe:  
 *nombreInterface.metodoStatic*

Ejemplo: Añadimos a la interfaz Relacionable un método estático *esNull*. El método comprueba si la variable *a* contiene o no la dirección de un objeto.

interface Relacionable {

    boolean esMayorQue(Relacionable a);

    boolean esMenorQue(Relacionable a);

    boolean esIgualQue(Relacionable a);

    default String nombreClase(){

        String clase = getClass().toString();

        int posicion = clase.lastIndexOf(".");

        return clase.substring(posicion+1);

    }

**static** boolean esNull(Relacionable a){

        return a == null;

    }

}

Ejemplo de uso de un método static en una interfaz Java: supongamos que disponemos de un Array en el que algunos elementos pueden ser null, es decir, no se les ha asignado un objeto. Utilizaremos el método *esNull* para comprobarlo y evitar que se produzca un error al intentar acceder a uno de estos elementos.

        Relacionable[] array = new Relacionable[20];

        array[1] = new Linea(15, 3, 9, 5);

        array[5] = new Fraccion(10, 7);

        array[9] = new Fraccion(6, 25);

        array[11] = new Linea(3, 4, 10, 15);

        array[14] = new Fraccion(8, 3);

        array[15] = new Linea(0, 7, 1, 4);

        array[18] = new Linea(2, -1, 3, 5);

        for (Relacionable r : array) {

            if (!Relacionable.esNull(r)) {   
 **//usamos el método static**

                System.out.println(r.nombreClase());

                System.out.println(r);

                System.out.println();

            }

        }

**ATRIBUTOS CONSTANTES**

Una interfaz también puede contener atributos constantes.

Ejemplo:

//Interfaz que contiene días relevantes en una aplicación de banca.

public interface IDiasOperaciones {

    int DIA\_PAGO\_INTERESES = 5;

    int DIA\_COBRO\_HIPOTECA = 30;

    int DIA\_COBRO\_TARJETA = 28;

}

Una clase que la implemente puede usar las constantes como si fuesen propias:

public class Banco implements IDiasOperaciones{

    …………..

    public void mostrarInformacionIntereses(){

        System.out.println("El día " + DIA\_PAGO\_INTERESES

                         + " de cada mes se realiza el pago de intereses");

    }

    …………..

}

Una clase que no implemente la interfaz puede usar las constantes escribiendo antes el nombre de la interfaz.

public static void main(String[] args) {

        …………………

        System.out.println("Los intereses se pagan el día "

                           +IDiasOperaciones.DIA\_PAGO\_INTERESES);

        System.out.println("La hipoteca se paga el día "

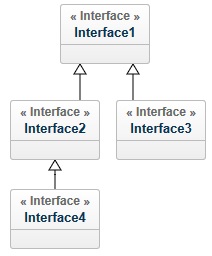
                           +IDiasOperaciones.DIA\_COBRO\_HIPOTECA);

        …………………

}

**HERENCIA ENTRE INTERFACES**

Se puede establecer una jerarquía de herencia entre interfaces igual que con las clases.

[](https://2.bp.blogspot.com/-j2Q0ar7MrFo/WOKGLF6mN_I/AAAAAAAAAh8/8s0T-gQFW_EJTHm4w0pIXx6XeZOgGuuSQCLcB/s1600/Herencia_Interface_Java.jpg)

public interface Interface2 extends Interface1{

…….

}

public interface Interface3 extends Interface1{

…….

}

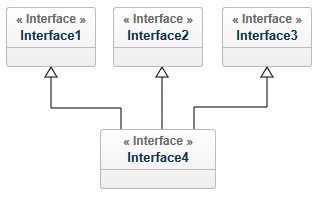
public interface Interface4 extends Interface2{

…….

}

Cada interface hereda el contenido de las interfaces que están por encima de ella en la jerarquía y puede añadir nuevo contenido o modificar lo que ha heredado siempre que sea posible. Los métodos static no se pueden redefinir. Los métodos abstract heredados se pueden convetir en métodos default. Los métodos default se pueden redefinir o convertir en abstract.

En las interfaces sí se permite herencia múltiple:

[](https://4.bp.blogspot.com/-G5m1ELGH7VY/WOKGizBxdZI/AAAAAAAAAiA/ObNGJuUibnEzU5jkk484XbNrI_cMqWOCgCLcB/s1600/Inteface_Java_Herencia_Multiple.jpg)

public interface Interface4 extends Interface1, Interface2, Interface3{

…….

}

Interface4 hereda todo el contenido de las tres interfaces.

Extraído de: <https://www.instintoprogramador.com.mx/2018/12/interfaces-en-java-ejemplos-de.html>

## 7.7. Ejercicio interfaz (para 11-03-2021)

Construye una interfaz llamada **Relacionable** que incluya los siguientes métodos:

// Devuelve verdadero si a es mayor que b

boolean **esMayor**(Object b) ;

// Devuelve verdadero si a es menor que b

boolean **esMenor**(Object b) ;

// Devuelve verdadero si a es igual que b

boolean **esIgual**(Object b) ;

Crea 3 clases:

Fracción

Linea //comparas longitud

Rectángulo //comparas el area

que implementen el interfaz Relacionable.

Crea un programa que instancie cada una de las clases y aplique sus métodos para probarlos.

((x1-x2)^2 + (y1-y2)^2)^0.5

## 7.7. Enumerados 1: Enumerados en Java

**1. Enum en Java**

**[2. Cómo funcionan los Enum](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "2_Como_funcionan_los_Enum)**

**[3. Declaración de enum en Java](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "3_Declaracion_de_enum_en_Java)**

**[4. Ejemplo con enum en Java](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "4_Ejemplo_con_enum_en_Java)**

**[4.1. Código de Ejemplo](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "41_Codigo_de_Ejemplo)**

**[5. Puntos importantes de enum](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "5_Puntos_importantes_de_enum)**

**[6. Métodos values(), ordinal() y valueOf()](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "6_Metodos_values_ordinal_y_valueOf)**

**[7. enum: Constructores, métodos, variables de instancia](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "7_enum_Constructores_metodos_variables_de_instancia)**

**[7.1. Explicación del Ejemplo](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "71_Explicacion_del_Ejemplo)**

**[8. Enum y Herencia](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "8_Enum_y_Herencia)**

**[8.1. Uso de ordinal( ) y compareTo()](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "81_Uso_de_ordinal_y_compareTo)**

**[9. enum y Métodos](https://javadesdecero.es/avanzado/enumerados-enum-ejemplos/" \l "9_enum_y_Metodos)**

### 7.7.1 Enum en Java

¿Qué es un ENUM? En su forma más simple, **una enumeración es una lista de constantes con nombre que definen un nuevo tipo de datos**. Un objeto de un tipo de enumeración solo puede contener los valores definidos por la lista. Por lo tanto, una enumeración proporciona una manera de definir con precisión un nuevo tipo de datos que tiene **un número fijo de valores válidos.**

Por ejemplo, los 4 palos en un mazo de cartas pueden ser 4 valores llamados espadas, bastos, oros y copas, que pertenecen a un tipo enumerado llamado Baraja. Otros ejemplos incluyen tipos de enumerados naturales (como los planetas, días de la semana, meses del año, colores, direcciones, etc.).

Desde una perspectiva de programación, las enumeraciones son útiles siempre que necesites definir un **conjunto de valores que represente una colección de elementos.** Por ejemplo, se puede usar una enumeración para representar un conjunto de códigos de estado, como *éxito*, *espera*, *error* y *reintentos*, que indican el progreso de alguna acción.

En el pasado, dichos valores se definían como **variables finales**, pero las enumeraciones ofrecen un enfoque más estructurado. Todos los enum implícitamente extienden de la clase java.lang.Enum.

 public abstract class Enum<E extends Enum<E>>

      extends Object implements Comparable<E>, Serializable

### 7.7.2 Cómo funcionan los Enum

Los enum en Java se usan cuando conocemos todos los valores posibles en tiempo de compilación.

En Java (desde 1.5), las enumeraciones se representan utilizando el tipo de datos **enum**. Las enumeraciones Java son más potentes que las enumeraciones C/C++. En Java, también podemos agregarle atributos/variables, métodos y constructores. El objetivo principal de enum es definir nuestros propios tipos de datos (**tipos de datos enumerados**).

### 7.7.3 Declaración de enum en Java

La declaración de Enum puede hacerse fuera de**una clase**, o dentro de una clase (class), pero **NO** dentro de un método.

**EJEMPLO**

// ejemplo donde se declara enum

// fuera de cualquier clase (Nota la palabra enum en lugar

// de la palabra class)

enum Color

{

ROJO, VERDE, AZUL;

}

public class Test

{

// El método

public static void main(String[] args)

{

Color c1 = Color.ROJO;

System.out.println(c1);

}

}

**Salida:**

**ROJO**

Una enumeración se crea usando la palabra clave **enum**.

La primera línea dentro de enum debe ser una lista de constantes y luego otras cosas como métodos, variables y constructores.

De acuerdo con las convenciones de nomenclatura de Java, se recomienda que nombremos las constantes con **mayúsculas**.

### 7.7.4 Ejemplo con enum en Java

**EJEMPLO**

Enumeración simple que enumera varias formas de transporte:

//Una enumeración de transporte

enum Transporte{

COCHE, CAMION, AVION, TREN, BARCO;

}

Los identificadores *COCHE*, *CAMION*, etc. se denominan **constantes de enumeración**.

Cada uno se declara **implícitamente** como un miembro público (**public**) y estático (**static**) de *Transporte*. Además, el tipo de las constantes de enumeración es el tipo de enumeración en el que se declaran las constantes, que es *Transporte* en este caso. Por lo tanto, en el lenguaje de Java, estas constantes se llaman **auto-tipado**.

Una vez que hayas definido una enumeración, puedes crear una variable de ese tipo. Sin embargo, aunque las enumeraciones definen un tipo de clase, no creamos una instancia de una enumeración usando *new,* declaramos variables de su tipo. Por ejemplo, así declaramos *tp* como una variable del tipo de enumeración *Transporte*:

Transporte tp;

Como tp es de tipo Transporte, los únicos valores que se le pueden asignar son los definidos por la enumeración. Por ejemplo, esto asigna tp el valor AVION:

tp = Transporte.AVION;

Se pueden comparar dos constantes de enumeración utilizando el operador relacional **==**. Por ejemplo, esta declaración compara el valor en tp con la constante TREN:

If (tp == Transporte.TREN) // ...

En nuestro caso no habría igualdad.

Un valor de enumeración también se puede usar para controlar una sentencia [switch](https://javadesdecero.es/basico/sentencia-switch-java-ejemplo/" \t "_blank). Por supuesto, todas las declaraciones de **case** deben usar constantes de la misma enumeración que la utilizada por la expresión de switch. Por ejemplo, este *switch* es perfectamente válido:

//Uso de enum para controlar una sentencia switch

switch(tp){

case COCHE:

//

case CAMION:

//

}

Observa que en las sentencias case, los nombres de las constantes de enumeración se usan sin estar calificados por el nombre de tipo de enumeración. Es decir, se utiliza CAMION, no Transporte.CAMION. Esto se debe a que el tipo de enumeración en la expresión de switch ya ha especificado implícitamente el tipo de enumeración de las constantes de case.

No es necesario calificar las constantes en las declaraciones de **case** con su nombre de tipo enum. De hecho, intentar hacerlo provocará un error de compilación.

#### 7.7.4.1 Código de Ejemplo

Cuando se muestra una constante de enumeración, como en una instrucción println(), se genera su nombre. Por ejemplo, dada esta declaración:

System.out.println(Transporte.BARCO);  
*//Se muestra el nombre BARCO.*

Prueba el siguiente programa que usa la enumeración Transporte:

enum Transporte{

COCHE, CAMION, AVION, TREN, BARCO;

}

class Enumerados {

public static void main(String[] args) {

Transporte tp;

tp=Transporte.AVION;

System.out.println("Valor de tp: "+tp);

System.out.println();

tp=Transporte.TREN;

//Comparación de 2 valores enum

if (tp==Transporte.TREN)

System.out.println("tp tiene el valor de TREN\n");

//enum para controlar sentencia switch

switch(tp){

case COCHE:

System.out.println("Un auto lleva personas.");

break;

case CAMION:

System.out.println("Un camión lleva carga.");

break;

case AVION:

System.out.println("Un avión vuela.");

break;

case TREN:

System.out.println("Un tren corre sobre rieles.");

break;

case BARCO:

System.out.println("Un barco navega en el agua.");

break;

}

}

}

Salida:

Valor de tp: AVION

tp tiene el valor de TREN

Un tren corre sobre rieles.

Las constantes en Transporte las pondremos en mayúsculas. (Por lo tanto, se usa COCHE, no coche.) Sin embargo, **no es obligatorio el uso de mayúsculas**. Debido a que las enumeraciones a menudo reemplazan las variables finales, que tradicionalmente se han usado en mayúsculas, se ha llegado al acuerdo de que las **constantes de enumeración las escribamos en mayúsculas**.

### 7.7.5 Puntos importantes de enum

Esta explicación es útil para que entendáis bien los enumerados:

* Cada enum es implementado **internamente** mediante el uso de *class*.

Internamente enum Color que definimos antes se convierte en:

class Color {

public static final Color ROJO = new Color();

public static final Color AZUL = new Color();

public static final Color VERDE = new Color();

}

* Cada constante enum representa un objeto de tipo enum.
* El tipo enum se puede pasar como un argumento para *switch*.

Vemos un ejemplo más:

// Un programa Java para demostrar el trabajo de enum

// en case de switch (Archivo Test.Java)

import java.util.Scanner;

// Una clase enum

enum Dia

{

LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;

}

// Controlador de clase que contiene un objeto de "Dia" y

// main().

public class Test

{

Dia dia;

// Constructor

public Test(Dia dia)

{

this.dia = dia;

}

// Imprime una línea sobre el DIA usando switch

public void diaEs()

{

switch (dia)

{

case LUNES:

System.out.println("Los lunes son feos.");

break;

case VIERNES:

System.out.println("Los viernes son mejores.");

break;

case SABADO:

case DOMINGO:

System.out.println("Los fines de semana son mejores.");

break;

default:

System.out.println("Los días entre semana son regulares.");

break;

}

}

// Metodo

public static void main(String[] args)

{

String str = "LUNES";

Test t1 = new Test(Dia.valueOf(str));

t1.diaEs();

}

}

Salida:

Los lunes son feos.

* Cada constante enum siempre es implícitamente public static final. Entonces, como es static, podemos acceder utilizando el nombre del *enum*. Y como es final, no podemos crear enumeraciones “*hijas*“.
* Podemos declarar el método main() dentro de enum.

Por ejemplo:

// Un programa Java para demostrar que podemos tener

// main() dentro de enum

enum Color

{

ROJO, VERDE, AZUL;

// Método

public static void main(String[] args)

{

Color c1 = Color.ROJO;

System.out.println(c1);

}

}

El hecho de que enum define una clase permite que la enumeración de Java tenga poderes que las enumeraciones en otros lenguajes no tienen. Por ejemplo, pueden **crearse constructores**, **agregar variables** y **métodos de instancia**, e incluso **implementar interfaces**.

A diferencia de la forma en que se implementan las enumeraciones en algunos otros lenguajes, *Java implementa enumeraciones como tipos de clases*. Aunque no creemos una instancia de una enumeración usando new, actúa de forma muy similar a otras clases.

### 7.7.6 Métodos values(), ordinal() y valueOf()

Todas las enumeraciones tienen automáticamente dos métodos predefinidos: **values()** y **valueOf()**. Sus formas generales son:

public static tipo-enum[ ] values( )

public static tipo-enum valueOf(String str)

* Estos métodos están presentes dentro de **java.lang.Enum**.
* El método **values​​()** se puede usar para devolver todos los valores presentes dentro de enum.
* El **orden** es importante en las enumeraciones. Al usar el método **ordinal()**, se puede encontrar cada índice de la constante enum, al igual que el índice de matriz.
* El método **valueOf()** devuelve la constante enum del valor de cadena especificado, si existe.

El compilador agrega automáticamente el método **values** cuando se crea un enum. Este método devuelve un array conteniendo todos los valores del enumerado en el orden en que son declarados y se usa comúnmente en combinación con el ciclo for-each para iterar sobre los valores de un tipo enum.

**Ejemplo** de código. Compílalo y comprueba el resultado de ejecución:

// Programa Java para demostrar el funcionamiento de values(),

// ordinal() y valueOf()

enum Color

{

ROJO, VERDE, AZUL;

}

public class Test

{

public static void main(String[] args)

{

// Llamando a values()

Color arr[] = Color.values();

// enum con bucle

for (Color col: arr)

{

// Llamando a ordinal() para encontrar el índice

// de color.

System.out.println(col + " en el índice "

+ col.ordinal());

}

// Usando valueOf(). Devuelve un objeto de

// Color con la constante dada.

// La segunda línea comentada causa la excepción

// IllegalArgumentException

System.out.println(Color.valueOf("ROJO"));

// System.out.println(Color.valueOf("BLANCO"));

}

}

Salida:

ROJO en el índice 0

VERDE en el índice 1

AZUL en el índice 2

ROJO

### 7.7.7 enum: Constructores, métodos, variables de instancia

Es importante comprender que cada constante de enumeración es un objeto de su tipo de enumeración (como si fuera un objeto de una clase). Por lo tanto, **una enumeración puede definir constructores, agregar métodos y tener variables de instancia**.

La primera vez que llamamos a una instancia del enum se **inicializan todas** las instancias del mismo, se ejecutan los constructores para cada instancia respetando el orden definido. **Cada constante** de enumeración puede llamar a **cualquier método** definido por la enumeración. Cada constante de enumeración tiene su propia copia de cualquier variable de instancia definida por la enumeración.

* enum puede contener un constructor y se ejecuta por separado para cada constante enum en el momento en que se utiliza la primera.
* No podemos crear objetos enum explícitamente y, por lo tanto, no podemos invocar el constructor enum directamente.

**Ejemplo**:

enum Color{

// Enums

ROJO("Rojo", 3), AZUL("Azul", 5);

// Atributos

private String nombreColor;

private int numColor;

public int contador=0;

// Constructor

// Rojo - 3 - 1

// Azul - 5 - 1

Color(String nom, int col) {

this.nombreColor = nom;

this.numColor = col;

this.contador ++;

System.out.println(this.nombreColor + " - "

+this.numColor + " - " + this.contador)

}

}

public class C1{

public static void main(String[] args){

//System.out.println("Inicializando: " + Color.ROJO);

// Se inicializan todas las instancias la primera vez que se

// usa el enum.

Color c1;

c1 = Color.AZUL;

System.out.println("Inicializando: " + c1);

Color c2;

c2 = Color.ROJO;

System.out.println("Inicializando: " + c2);

}

}

La siguiente versión de Transporte ilustra el uso de un constructor, una variable de instancia y un método. Da a cada tipo de transporte una velocidad típica:

//Uso de un constructor, una variable de instancia y un método.

enum Transporte{

COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);

private int velocidad; //velocidad típica de cada transporte

//Añadir un constructor

Transporte(int s){velocidad=s;}

//Añadir un método

int getVelocidad(){return velocidad;}

}

class Enumerados {

public static void main(String[] args) {

Transporte tp;

//Mostrar la velocidad de un avión

System.out.println("La velocidad típica para un avión es: "+

Transporte.AVION.getVelocidad()+ " millas por hora.\n");

//Mostrar todas las velocidades y transportes

System.out.println("Todas las velocidades de transporte: ");

for (Transporte t:Transporte.values())

System.out.println(t + ": velocidad típica es "

+ t.getVelocidad() + " millas por hora.");

}

}

Salida:

La velocidad típica para un avión es: 600 millas por hora.

Todas las velocidades de transporte:

COCHE: velocidad típica es 60 millas por hora.

CAMION: velocidad típica es 50 millas por hora.

AVION: velocidad típica es 600 millas por hora.

TREN: velocidad típica es 70 millas por hora.

BARCO: velocidad típica es 20 millas por hora.

#### 7.7.7.1 Explicación del Ejemplo

Esta versión de Transporte agrega tres cosas:

La **primera** es la variable de instancia **velocidad**, que se usa para mantener la velocidad de cada tipo de transporte.

La **segunda** es el constructor **Transporte**, que pasa la velocidad de un transporte.

La tercera es el método **getVelocidad()**, que devuelve el valor de velocidad. Solo admite geters, que es lo que tiene sentido.

|  |
| --- |
| Cuando una variable de tipo *Transporte* (variable **tp** en el ejemplo)se declara en ***main() y se inicializa a un valor de la enumeración***, el constructor de Transporte se llama una vez para cada constante que se especifica. Observa cómo se especifican los argumentos para el constructor, poniéndolos entre paréntesis, después de cada constante, como se muestra aquí:  **COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);** |

Estos valores se pasan al parámetro de *Transporte()*, que luego asigna este valor a la *velocidad*. Hay algo más que notar sobre la lista de constantes de enumeración: **termina con un punto y coma**. Es decir, la última constante, *BARCO*, va seguida de un punto y coma. Cuando una enumeración contiene otros miembros, la lista de enumeración debe terminar en punto y coma.

Como cada **constante de enumeración tiene su propia copia de velocidad**, puede obtener la velocidad de un tipo de transporte especificado llamando a *getVelocidad()*. Por ejemplo, en *main()* la velocidad de un avión se obtiene mediante la siguiente llamada:

**Transporte.AVION.getVelocidad()**

La velocidad de cada transporte se obtiene al recorrer la enumeración mediante un [bucle for](https://javadesdecero.es/intermedio/bucle-for-each/" \t "_blank). Como hay una copia de velocidad para cada constante de enumeración, el valor asociado con una constante es separado y distinto del valor asociado con otra constante. Este es un concepto poderoso, que está disponible solo cuando las enumeraciones se implementan como clases, como lo hace Java.

Aunque el ejemplo anterior contiene solo un constructor, una enumeración puede ofrecer dos o más formas de sobrecargas, al igual que cualquier otra clase.

### 7.7.8 Enum y Herencia

Hay dos **restricciones** que se aplican a las enumeraciones.

* Una enumeración no puede heredar de otra clase (ya hereda implícitamente de java.lang.Enum).
* Una enumeración no puede ser una superclase (no se puede heredar de ella).

Esto significa que una enumeración no se puede extender. De lo contrario, enum actúa como cualquier otro tipo de clase. **La clave es recordar que cada una de las constantes de enumeración es un objeto de la clase en la que está definida**.

* Aunque no puede heredar de una superclase al declarar una enumeración, todas las enumeraciones heredan automáticamente una: **java.lang.Enum**. Esta clase define varios métodos que están disponibles para el uso de todas las enumeraciones.
* Hay dos métodos heredados que podemos usar: **ordinal()** y **compareTo()**.
* El método **toString()** devuelve el nombre de la constante enum.
* enum puede implementar interfaces.

#### 7.7.8.1 Uso de ordinal( ) y compareTo()

El método ordinal() se muestra aquí:

**final int ordinal()**

Devuelve el valor ordinal de la constante invocadora. Los valores ordinales comienzan en cero. Por lo tanto, en la enumeración Transporte, COCHE tiene un valor ordinal de cero, CAMION tiene un valor ordinal de 1, AVION tiene un valor ordinal de 2, y así sucesivamente.

Se puede comparar el valor ordinal de dos constantes de la misma enumeración utilizando el método compareTo(). Tiene esta forma general:

**final int compareTo(tipo-enum e)**

Aquí, *tipo-enum* es el tipo de enumeración y *e* es la constante que se compara con la constante de invocación. Tanto la constante de invocación como *e* deben ser de la misma enumeración.

* Si la constante de invocación tiene un valor ordinal menor que *e,* entonces *compareTo()* devuelve un valor **negativo**.
* Si los dos valores ordinales son iguales, se devuelve **cero**.
* Si la constante de invocación tiene un valor ordinal mayor que *e*, se devuelve un valor **positivo**.

El siguiente programa muestra **ordinal()** y **compareTo()**:

package enumTransportes4;

//Demostración de ordinal() y compareTo()

enum Transporte {

COCHE, CAMION, AVION, TREN, BARCO;

}

class Enumerados {

public static void main(String[] args) {

Transporte tp1, tp2, tp3;

// Obtiene todos los valores ordinales usando ordinal().

System.out.println("Aquí están todas las constantes de Transporte y sus valores ordinales: ");

// Acuérdate de que Transporte.values() devuelve un array

// con las constantes de la enumeración

for (Transporte t: Transporte.values())

System.out.println(t + " " + t.ordinal());

tp1 = Transporte.AVION;

tp2 = Transporte.TREN;

tp3 = Transporte.AVION;

System.out.println();

// Uso de CompareTo()

if (tp1.compareTo(tp2) < 0)

System.out.println(tp1 + " esta antes que " + tp2);

else if (tp1.compareTo(tp2) > 0)

System.out.println(tp1 + " esta despues que " + tp2);

else

System.out.println(tp1 + " es igual que " + tp3);

tp1 = Transporte.CAMION;

tp2 = Transporte.COCHE;

tp3 = Transporte.CAMION;

// Uso de CompareTo()

if (tp1.compareTo(tp2) < 0)

System.out.println(tp1 + " esta antes que " + tp2);

if (tp1.compareTo(tp2) > 0)

System.out.println(tp1 + " esta despues que " + tp2);

if (tp1.compareTo(tp3) == 0)

System.out.println(tp1 + " es igual que " + tp3);

}

}

Salida:

Aquí están todas las constantes de Transporte y sus valores ordinales:

COCHE 0

CAMION 1

AVION 2

TREN 3

BARCO 4

AVION está antes que TREN

AVION es igual que AVION

### 7.7.9 enum y Métodos

* enum puede contener métodos concretos, es decir, que no tengan ningún método abstracto (**abstract**).

Por ejemplo (fijaos bien en la ejecución):

// Programa Java para demostrar que los enum pueden tener

// Constructores y métodos concretos.

// Una enumeración (Note la palabra enum en lugar de class)

enum Color

{

ROJO, VERDE, AZUL;

// enum constructor llamado por separado para cada constante

private Color()

{

System.out.println("Constructor llamado para : " + this);

}

// Solo métodos concretos (no abstractos) permitidos

public void colorInfo()

{

System.out.println(this + “ desde colorInfo()”);

}

}

public class Test

{

// Metodo

public static void main(String[] args)

{

Color c1 = Color.ROJO;

// Míralo en el Proyecto, está separado y se ve más claro

System.out.println(c1);

c1.colorInfo();

}

}

Salida:

Constructor llamado para: ROJO

Constructor llamado para: VERDE

Constructor llamado para: AZUL

ROJO

ROJO desde colorInfo()

## 7.8. Enumerados 2: Enumerados en Java

Desde java 5 tenemos la posibilidad de definir enumerados *enum*.

***Un enumerado (o Enum) es una clase "especial"*** (tanto en Java como en otros lenguajes) ***que limita la creación de objetos a los especificados explícitamente en la implementación de la clase.*** La única restricción que tienen los enumerados respecto a una clase normal es que si tiene constructor, este debe ser privado para que no se puedan crear nuevos objetos.

***Ejemplo*** sobre una clase Enum. Los futbolistas están caracterizados por una demarcación a la hora de jugar un partido de fútbol, por tanto las demarcaciones en las que puede jugar un futbolista son finitas y por tanto se pueden enumerar en: Portero, Defensa, Centrocampista y Delantero. Con esta especificación podemos crearnos la siguiente clase "Enum" llamada "Demarcación":

public enum Demarcacion

{

PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO

}

***Por convenio los nombres de los enumerados se escriben en mayúsculas.***

Es muy importante entender que un "***Enum***" en java ***es realmente una clase*** (cuyos objetos solo pueden ser los definidos en esta clase: PORTERO…, DELANTERO) ***que hereda*** de la clase "***Enum(java.lang.Enum)***" y por tanto los enumerados tienen una serie de métodos heredados de esa clase padre. A continuación, vamos a mostrar algunos de los métodos más utilizados de los enumerados:

public enum Demarcacion{PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO}

Demarcacion delantero = Demarcacion.DELANTERO;

// Instancia de un enum de la clase Demarcación

delantero.name();

// Devuelve un String con el nombre de la constante (DELANTERO)

delantero.toString();

// Devuelve un String con el nombre de la constante (DELANTERO)

delantero.ordinal();

// Devuelve un entero con la posición del enum según está declarada (3 en este caso).

delantero.compareTo(Enum otro);

// Compara el enum con el parámetro según el orden en el que están declarados los enum

Demarcacion.values();

// Devuelve un array que contiene todos los enum

Ejemplo para probar:

Demarcacion delantero = Demarcacion.DELANTERO;

Demarcacion defensa = Demarcacion.DEFENSA;

// Devuelve un String con el nombre de la constante

System.out.println("delantero.name()= "+delantero.name());

System.out.println("defensa.toString()= "+defensa.toString());

// Devuelve un entero con la posición de la constante según está declarada.

System.out.println("delantero.ordinal()= "+delantero.ordinal());

// Compara el enum con el parámetro según el orden en el que están // declaradas las constantes.

System.out.println("delantero.compareTo(portero)= " + delantero.compareTo(defensa));

System.out.println("delantero.compareTo(delantero)= " + delantero.compareTo(delantero));

// Recorre todas las constantes de la enumeración

for(Demarcacion d: Demarcacion.values()){

System.out.println(d.toString()+" - ");

}

Tenemos como salida los siguientes resultados:

delantero.name()= DELANTERO

defensa.toString()= DEFENSA

delantero.ordinal()= 3

delantero.compareTo(defensa)= 2

delantero.compareTo(delantero)= 0

PORTERO - DEFENSA - CENTROCAMPISTA - DELANTERO

Como ya se ha dicho, un **enum** es una clase especial que limita la creación de objetos a los especificados en su clase (por eso, su constructor es privado, como se ve en el siguiente fragmento de código); pero estos objetos pueden tener atributos como cualquier otra clase. En la siguiente declaración de la clase, vemos un ejemplo en el que definimos un enumerado "Equipo" que va a tener dos atributos; el nombre y el puesto en el que quedaron en la liga del año 2009/2010.

public enum Equipo

{

BARÇA("FC Barcelona",1), REAL\_MADRID("Real Madrid",2),

SEVILLA("Sevilla FC",4), VILLAREAL("Villareal",7);

private String nombreClub;

private int puestoLiga;

private Equipo (String nombreClub, int puestoLiga){

this.nombreClub = nombreClub;

this.puestoLiga = puestoLiga;

}

public String getNombreClub() {

return nombreClub;

}

public int getPuestoLiga() {

return puestoLiga;

}

}

Como se ve BARCA, REAL\_MADRID, etc. son nombres del enumerado (u objetos de la clase Equipo) que tendrán como atributos el "nombreClub" y "puestoLiga". Como se ve en la clase ***definimos un constructor que es privado*** (es decir que solo es visible dentro de la clase Equipo) y solo definimos los métodos "get". Para trabajar con los atributos de estos enumerados se hace de la misma manera que con cualquier otro objeto; se instancia un objeto y se accede a los atributos con los métodos get. En el siguiente fragmento de código vamos a ver cómo trabajar con enumerados que tienen atributos:

// Instanciamos el enumerado

Equipo villareal = Equipo.VILLAREAL;

// Devuelve un String con el nombre de la constante

System.out.println("villareal.name()= "+villareal.name());

// Devuelve el contenido de los atributos

System.out.println("villareal.getNombreClub()="

+villareal.getNombreClub());

System.out.println("villareal.getPuestoLiga()=" +villareal.getPuestoLiga());

Como salida de este fragmento de código tenemos lo siguiente:

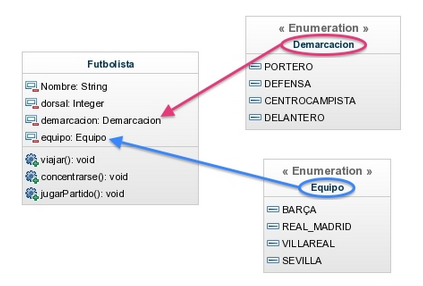
villareal.name()= VILLAREAL

villareal.getNombreClub()= Villareal

villareal.getPuestoLiga()= 7

Es muy importante tener claro que los enumerados no son Strings (aunque pueden serlo), sino que son objetos de una clase que solo son instanciables desde la clase que se implementa y que no se puede crear un objeto de esa clase desde cualquier otro lado que no sea dentro de esa clase. Es muy común (sobre todo cuando se está aprendiendo qué son los enumerados) que se interprete que un enumerado es una lista finita de Strings y en realidad es una lista finita de **objetos** de una determinada clase con sus atributos, constructor y métodos getter aunque estos sean privados.

A continuación, vamos a poner un sencillo ejemplo en el que vamos a mezclar los dos enumerados anteriores (Demarcación y Equipo). En este ejemplo vamos a crear unos objetos de la clase **Futbolista**, que representarán a los jugadores de la selección española de fútbol que ganaron el mundial de fútbol de Sudáfrica en el año 2010. Esta clase va a caracterizar a los futbolistas por su nombre, su dorsal, la demarcación en la que juegan y el club de fútbol al que pertenecen; tal y como vemos en el siguiente diagrama de clases.



Como vemos, los atributos de demarcación y equipo son de la clase **Demarcacion** y **Equipo** respectivamente y son los enumerados vistos anteriormente; por tanto, un futbolista solo podrá pertenecer a uno de los cuatro equipos que forman el enumerado "Equipo" y podrá jugar en alguna de las cuatro demarcaciones que forman el enumerado "Demarcación". A continuación, mostramos la implementación de la clase "Futbolista":

package Main;

public class Futbolista {

private int dorsal;

private String Nombre;

private Demarcacion demarcacion;

private Equipo equipo;

public Futbolista() {

}

public Futbolista(String nombre, int dorsal, Demarcacion demarcacion, Equipo equipo) {

this.dorsal = dorsal;

Nombre = nombre;

this.demarcacion = demarcacion;

this.equipo = equipo;

}

// Metodos getter y setter

................................

@Override

public String toString() {

return this.dorsal + " - " + this.Nombre + " - "

+ this.demarcacion.name() + " - "

+ this.equipo.getNombreClub();

}

}

Dada esta clase podemos crearnos ya objetos de la clase futbolista, como mostramos a continuación:

Futbolista casillas = new Futbolista("Casillas", 1, Demarcacion.PORTERO, Equipo.REAL\_MADRID);

Futbolista capdevila = new Futbolista("Capdevila", 11, Demarcacion.DEFENSA, Equipo.VILLAREAL);

Futbolista iniesta = new Futbolista("Iniesta", 6, Demarcacion.CENTROCAMPISTA, Equipo.BARCA);

Futbolista navas = new Futbolista("Navas", 22, Demarcacion.DELANTERO, Equipo.SEVILLA);

Como vemos la demarcación y el equipo al que pertenecen solo pueden ser los declarados en la clase enumerado. Si llamamos al método "toString()" declarado en la clase futbolista, podemos imprimir por pantalla los datos de los futbolistas. Dado el siguiente código:

System.out.println(casillas.toString());

System.out.println(capdevila.toString());

System.out.println(iniesta.toString());

System.out.println(navas.toString());

Y dado el siguiente método "toString()"

@Override

public String toString() {

return this.dorsal + " - " + this.Nombre + " - "

+ this.demarcacion.name() + " - "

+ this.equipo.getNombreClub();

}

Tenemos como salida lo siguiente:

1 - Casillas - PORTERO - Real Madrid

11 - Capdevila - DEFENSA - Villareal

6 - Iniesta - CENTROCAMPISTA - FC Barcelona

22 - Navas - DELANTERO - Sevilla FC

En resumen: Un enumerado está formado por objetos definidos en la misma clase con constructor privado y si tiene atributos, estos solo tienen que tener métodos "getter" para obtener el valor del atributo.

Mirar: <http://www.aprenderaprogramar.es/index.php?option=com_content&view=article&id=647:tipos-enumerados-enum-java-ejemplos-de-codigo-error-enum-types-must-not-be-local-ejercicio-cu00681b&catid=68:curso-aprender-programacion-java-desde-cero&Itemid=188>

<http://chuwiki.chuidiang.org/index.php?title=Enumerados_en_java>

## 7.9. Enumerados en Java: Ejemplos a implementar

***Ejercicio 1*** que nos va a dar un error. Se propone que compruebes y expliques el error y encuentres una solución basada en la excepción que se produce.

public class PruebaValueOf {

    enum Animal {

        PERRO, GATO

    };

    public static void main(String[] args) {

        Animal animal = Animal.valueOf(args[0]);

        switch (animal) {

        case PERRO:

            System.out.println("El perro ladra.");

            break;

        case GATO:

            System.out.println("El gato maulla.");

            break;

        default:

            System.out.println("No es un tipo aceptado.");

         }

     }

}

***Ejercicio 2*** sobre una clase Enum. Los futbolistas están caracterizados por una demarcación a la hora de jugar un partido de fútbol, por tanto las demarcaciones en las que puede jugar un futbolista son finitas y por tanto se pueden enumerar en: Portero, Defensa, Centrocampista y Delantero. Con esta especificación podemos crearnos la siguiente clase "Enum" llamada "Demarcación":

public enum Demarcacion

{

PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO

}

***Por convenio los nombres de los enumerados se escriben en mayúsculas.***

Es muy importante entender que un "***Enum***" en java ***es realmente una clase*** (cuyos objetos solo pueden ser los definidos en esta clase: PORTERO,…, DELANTERO) ***que hereda*** de la clase "***Enum (java.lang.Enum)***" y por tanto los enumerados tienen una serie de métodos heredados de esa clase padre. A continuación, vamos a mostrar algunos de los métodos más utilizados de los enumerados:

public enum Demarcacion{PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO}

// Instancia de un enum de la clase Demarcación.

Demarcacion delantero = Demarcacion.DELANTERO;

// Devuelve un String con el nombre del enumerado (DELANTERO).

delantero.name();

// Devuelve un String con el nombre del enumerado (DELANTERO).

delantero.toString();

// Devuelve un entero con la posición del enum según está declarada (3).

delantero.ordinal();

// Compara el enum que invoca el método con el enum parámetro, según el orden en el que están declarados los enum.

delantero.compareTo(Enum otro);

// Devuelve un array que contiene todos los enum.

Demarcacion.values();

// Devuelve el enumerado que corresponde con la cadena pasada por

// parámetro

System.out.println(Demarcacion.valueOf("DELANTERO"));

Ejemplo para probar:

Demarcacion delantero = Demarcacion.DELANTERO;

Demarcacion defensa = Demarcacion.DEFENSA;

// Devuelve un String con el nombre de la constante

System.out.println("delantero.name()= " + delantero.name());

System.out.println("defensa.toString()= " + defensa.toString());

// Devuelve un entero con la posición de la   
// constante según está declarada.

System.out.println("delantero.ordinal()= " + delantero.ordinal());

// Compara el enum con el parámetro según el orden en el que están   
// declaradas las constantes.

System.out.println("delantero.compareTo(portero)= "   
 +delantero.compareTo(defensa));

System.out.println("delantero.compareTo(delantero)= "   
 +delantero.compareTo(delantero));

// Recorre todas las constantes de la enumeración

for(Demarcacion d: Demarcacion.values()){

System.out.println(d.toString()+" - ");

}

Tenemos como salida los siguientes resultados:

delantero.name()= DELANTERO

defensa.toString()= DEFENSA

delantero.ordinal()= 3

delantero.compareTo(defensa)= 2

delantero.compareTo(delantero)= 0

PORTERO - DEFENSA - CENTROCAMPISTA - DELANTERO

Como ya se ha dicho, un **enum** es una clase especial que **limita la creación de objetos a los especificados en su clase** (por eso, su constructor es privado, como se ve en el siguiente fragmento de código); pero estos objetos pueden tener atributos como cualquier otra clase. En la siguiente declaración de la clase, vemos un ejemplo en el que definimos un enumerado "Equipo" que va a tener dos atributos; el nombre y el puesto en el que quedaron en la liga del año 2009/2010.

public enum Equipo

{

BARÇA("FC Barcelona",1), REAL\_MADRID("Real Madrid",2),

SEVILLA("Sevilla FC",4), VILLAREAL("Villareal",7);

private String nombreClub;

private int puestoLiga;

private Equipo (String nombreClub, int puestoLiga){

this.nombreClub = nombreClub;

this.puestoLiga = puestoLiga;

}

public String getNombreClub() {

return nombreClub;

}

public int getPuestoLiga() {

return puestoLiga;

}

}

Como se ve BARCA, REAL\_MADRID, etc. son el nombre del enumerado (u objetos de la clase Equipo) que tendrán como atributos el "nombreClub" y "puestoLiga". Como se ve en la clase ***definimos un constructor que es privado*** (es decir que solo es visible dentro de la clase Equipo) y solo definimos los métodos "get". Para trabajar con los atributos de estos enumerados se hace de la misma manera que con cualquier otro objeto; se instancia un objeto y se accede a los atributos con los métodos get. En el siguiente fragmento de código vamos a ver cómo trabajar con enumerados que tienen atributos:

// Instanciamos el enumerado

Equipo villareal = Equipo.VILLAREAL;

// Devuelve un String con el nombre de la constante

System.out.println("villareal.name()= "+villareal.name());

// Devuelve el contenido de los atributos

System.out.println("villareal.getNombreClub()="

+villareal.getNombreClub());

System.out.println("villareal.getPuestoLiga()="  
 +villareal.getPuestoLiga());

Como salida de este fragmento de código tenemos lo siguiente:

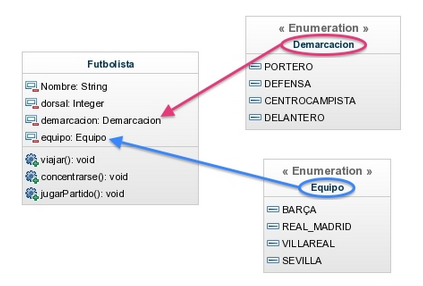
villareal.name()= VILLAREAL

villareal.getNombreClub()= Villareal

villareal.getPuestoLiga()= 7

|  |
| --- |
| Es muy importante tener claro que los enumerados no son Strings (aunque pueden serlo), sino que son objetos de una clase que solo son instanciables desde la clase que se implementa y que no se puede crear un objeto de esa clase desde cualquier otro lado que no sea dentro de esa clase. Es muy común (sobre todo cuando se está aprendiendo que son los enumerados) que se interprete que un enumerado es una lista finita de Strings y en realidad es una lista finita de **objetos** de una determinada clase con sus atributos, constructor y métodos getter aunque estos sean privados. |

A continuación, vamos a poner un sencillo ejemplo en el que vamos a mezclar los dos enumerados anteriores (Demarcación y Equipo). En este ejemplo, vamos a crear unos objetos de la clase **Futbolista**, que representarán a los jugadores de la selección española de fútbol que ganaron el mundial de fútbol de Sudáfrica en el año 2010. Esta clase va a caracterizar a los futbolistas por su nombre, su dorsal, la demarcación en la que juegan y el club de fútbol al que pertenecen; tal y como vemos en el siguiente diagrama de clases.



Como vemos los atributos de demarcación y equipo son de la clase **Demarcacion** y **Equipo** respectivamente y son los enumerados vistos anteriormente; por tanto un futbolista solo podrá pertenecer a uno de los cuatro equipos que forman el enumerado "Equipo" y podrá jugar en alguna de las cuatro demarcaciones que forman el enumerado "Demarcación". A continuación, se muestra la implementación de la clase "Futbolista":

package Main;

public class Futbolista {

private int dorsal;

private String Nombre;

private Demarcacion demarcacion;

private Equipo equipo;

public Futbolista() {

}

public Futbolista(String nombre, int dorsal,

Demarcacion demarcacion, Equipo equipo) {

this.dorsal = dorsal;

Nombre = nombre;

this.demarcacion = demarcacion;

this.equipo = equipo;

}

// Metodos getter y setter

................................

@Override

public String toString() {

return this.dorsal + " - " + this.Nombre + " - "

+ this.demarcacion.name()

+ " - " + this.equipo.getNombreClub();

}

}

Dada esta clase podemos crearnos ya objetos de la clase futbolista, como mostramos a continuación:

Futbolista casillas = new Futbolista("Casillas", 1, Demarcacion.PORTERO, Equipo.REAL\_MADRID);

Futbolista capdevila = new Futbolista("Capdevila", 11, Demarcacion.DEFENSA, Equipo.VILLAREAL);

Futbolista iniesta = new Futbolista("Iniesta", 6,

Demarcacion.CENTROCAMPISTA, Equipo.BARCA);

Futbolista navas = new Futbolista("Navas", 22, Demarcacion.DELANTERO, Equipo.SEVILLA);

Como vemos la demarcación y el equipo al que pertenecen solo pueden ser los declarados en la clase enumerado. Si llamamos al método "toString()" declarado en la clase futbolista, podemos imprimir por pantalla los datos de los futbolistas. Dado el siguiente código:

System.out.println(casillas.toString());

System.out.println(capdevila.toString());

System.out.println(iniesta.toString());

System.out.println(navas.toString());

Y dado el siguiente método "toString()":

@Override

public String toString() {

return this.dorsal + " - " + this.Nombre + " - "

+ this.demarcacion.name() + " - " +

this.equipo.getNombreClub();

}

Tenemos como salida lo siguiente:

1 - Casillas - PORTERO - Real Madrid

11 - Capdevila - DEFENSA - Villareal

6 - Iniesta - CENTROCAMPISTA - FC Barcelona

22 - Navas - DELANTERO - Sevilla FC

En **resumen**: Un enumerado está formado por objetos definidos en la misma clase con constructor privado y si tiene atributos, estos solo tienen que tener métodos "getter" para obtener el valor del atributo.