**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), ya que ayuda a evitar el mal uso de un objeto.

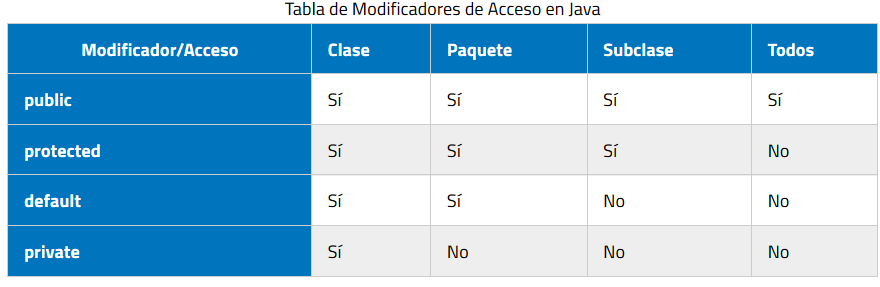
Al permitir el acceso a datos privados solo a través de un conjunto de métodos bien definidos, **podemos evitar que se asignen valores incorrectos a esos datos**, por ejemplo, realizando una verificación de rango.

* 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



## 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

## 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

## 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 sub-clases 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();

}

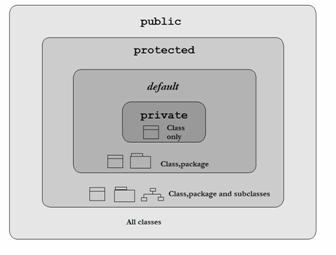
}

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.



## 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();

}

}

## 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) (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++;

}

## 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>