



# UD05.A.POO.Clases Internas

## Apuntes

DAM1-Programación 2024-25

<b>Introducción</b>	<b>2</b>
<b>Clases e interfaces internas static</b>	<b>4</b>
<b>Clases internas miembro (no static)</b>	<b>7</b>
<b>Clases internas locales</b>	<b>11</b>
<b>Clases anónimas</b>	<b>13</b>

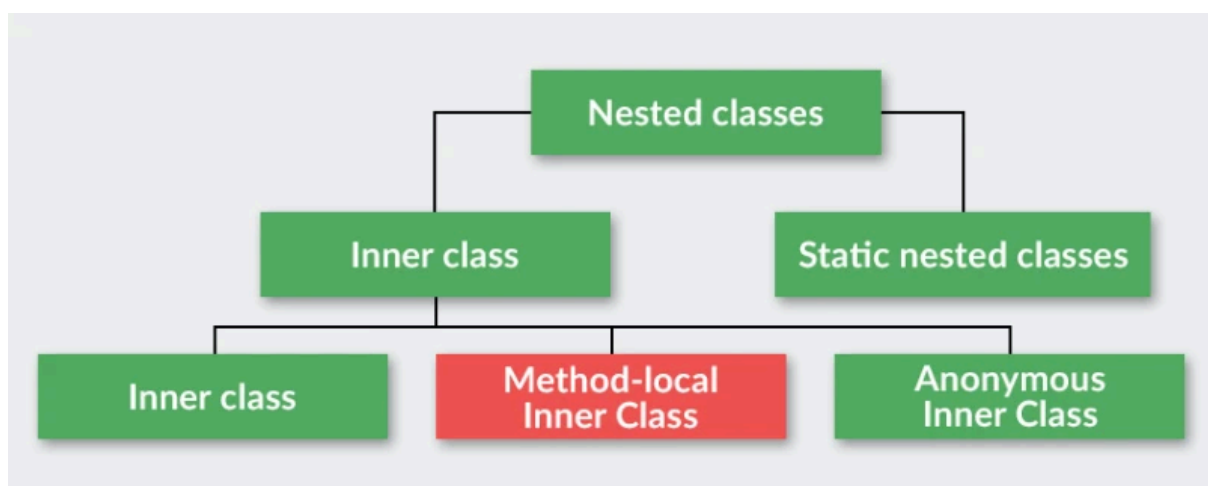
- [Píldoras informáticas](#)
  -  Curso Java. Interfaces y clases internas. Clases internas I. Vídeo 53
  -  Curso Java. Interfaces y clases internas. Clases internas II. Vídeo 54

# Introducción

Una clase interna es una clase definida dentro de otra clase, llamada clase contenedora, en alguna variante de la siguiente forma general:

```
class ClaseContenedora {  
    ...  
    class ClaseInterna {  
        ...  
    }  
    ...  
}
```

Las clases internas fueron introducidas en la versión Java 1.1. Además de su utilidad en sí, las clases internas se utilizan mucho en un nuevo modelo de eventos que se introdujo en dicha versión de Java.



Hay cuatro tipos de clases internas:

1. **Clases anidadas estáticas** (*static nested classes*). Sólo pueden ser creadas dentro de otra clase al máximo nivel, utilizando la palabra *static*.
2. **Clases internas miembro** (*inner nested classes*). Se definen en la clase contenedora sin la palabra *static*. Se utilizan para crear objetos internos a una clase (el motor de un coche por ejemplo), de modo que esos objetos pasan a ser atributos de la clase.
3. **Clases internas locales**. Se declaran dentro de un bloque de código, normalmente en un método, aunque también se pueden crear en un inicializador static o de objeto. Normalmente se usan cuando el método intenta solucionar un problema complicado y necesita apoyarse en una clase, pero no se necesita que esta clase esté disponible fuera. Son por tanto clases que quedan fuera del diseño. Las clases internas a un método sólo pueden ser instanciadas en el interior de dicho método
4. **Clases anónimas**. Similares a las clases internas locales, pero sin nombre. Sólo se puede crear un único objeto. Se utilizan con mucha frecuencia para definir clases y objetos que definan comparadores, que gestionen los eventos de los distintos componentes de la interface de usuario.

En lo sucesivo se utilizará la terminología clase contenedora o clase global para hacer referencia a la clase que contiene a la clase interna.

Hay que señalar que la JVM (Java Virtual Machine) no sabe nada de la existencia de clases internas. Por ello, el compilador convierte estas clases en clases globales, contenidas en ficheros **.class** cuyo nombre es **ClaseContenedora\$ClaseInterna.class**. Esta conversión inserta variables ocultas, métodos y argumentos en los constructores. De todas formas, lo que más afecta al programador de todo esto es lo referente al nombre de los ficheros que aparecen en el directorio donde se realiza la compilación, que pueden resultar sorprendentes si no se conoce su origen.

# Clases e interfaces internas static

Se conocen también con el nombre de **clases anidadas** (*nested classes*). Las clases e interfaces internas static sólo pueden ser creadas dentro de otra clase al máximo nivel, es decir directamente en el bloque de definición de la clase contenedora y no en un bloque más interno.

También es posible definir clases e interfaces internas static dentro de una interfaz contenedora. Este tipo de clases internas **se definen utilizando la palabra static**. Todas las interfaces internas son implícitamente static.

En cierta forma, las clases internas static se comportan como clases normales en el mismo paquete (*package*).

Para utilizar su nombre desde fuera de la clase contenedora hay que precederlo por el **nombre de la clase contenedora y el operador punto (.)**. Este tipo de relación entre clases se puede utilizar para agrupar varias clases dentro de una clase más general. Lo mismo puede decirse de las interfaces internas. Las clases internas static pueden ver y utilizar los miembros static de la clase contenedora.

No se necesitan objetos de la clase contenedora para crear objetos de la clase interna static. Los métodos de la clase interna static no pueden acceder directamente a los objetos de la clase contenedora, en caso de que los haya: deben disponer de una referencia a dichos objetos, como cualquier otra clase.

La sentencia **import** puede utilizarse para importar una clase interna static, en la misma forma que si se tratara de importar una clase de un package (con el punto (.)). Por ejemplo, si la interface `Linkable` es interna a la clase `List`, para implementar dicha interface hay que escribir,

```
class ... implements List.Linkable

// y para importarla hay que usar,

import List.*;

// o bien

import List.Linkable;
```

Otras características importantes son las siguientes:

1. Pueden definirse clases e interfaces internas dentro de interface y clases contenedoras, con las cuatro combinaciones posibles.
2. Puede haber varios niveles, esto es una clase interna static puede ser clase contenedora de otra clase interna static, y así sucesivamente.
3. Las clases e interfaces internas static pertenecen al *package* de la clase contenedora.
4. Pueden utilizarse los calificadores `final`, `public`, `private` y `protected`. Ésta es una forma más de controlar el acceso a ciertas clases.

A continuación se presenta un ejemplo de clase interna static:

```
// fichero ClasesIntStatic.java
class A {
    int i = 1; // variable miembro de objeto
    static int is = -1; // variable miembro de clase

    public A(int i) {
        this.i = i;
    } // constructor
    // a los métodos de la clase contenedora hay que pasarles referencias
    // a los objetos de la clase interna static

    public void printA(Bs unBs) {
        System.out.println("i=" + i + " unBs.j=" + unBs.j);
    }

    // definición de una clase interna static
    static class Bs {
        int j = 2;

        public Bs(int j) {
            this.j = j;
        } // constructor
        // los métodos de la clase interna static no pueden acceder a la i
        // pues es una variable de objeto. Sí pueden acceder a is

        public void printBs() {
            System.out.println(" j=" + j + " is=" + is);
        }
    } // fin clase Bs

    public static void main(String[] arg) {
        A a1 = new A(11), a2 = new A(12);
        System.out.println("a1.i=" + a1.i + " a2.i=" + a2.i);
        // dos formas de crear objetos de la clase interna static
        A.Bs b1 = new A.Bs(-10); // necesario poner A.Bs
        A.Bs b2 = new Bs(-11); // b2 es independiente de a1

        // referencia directa a los objetos b1 y b2
        System.out.println("b1.j=" + b1.j + " b2.j=" + b2.j); // los métodos de la
        // clase interna acceden directamente a
        // las
        // variables
        // de la clase contenedora sólo si son static
    }
}
```

```
b1.printBs(); // escribe: j=-10 is=-1
b2.printBs(); // escribe: j=-20 is=-1
// a los métodos de la clase contenedora hay que pasarles referencias
// a los objetos de la clase interna, para que puedan identificarlos
a1.printA(b1); // escribe: i=11 unBs.j=-10
a1.printA(b2); // escribe: i=11 unBs.j=-11
} // fin de main()

} // fin clase contenedora Aclass ClasesIntStatic {
```

## Clases internas miembro (no static)

Las clases internas miembro o simplemente clases internas, son clases definidas al máximo nivel de la clase contenedora, **sin la palabra static**. Se suelen llamar clases internas miembro o simplemente clases internas. No existen interfaces internas de este tipo.

Las clases internas no pueden tener variables miembro static. Tienen una nueva sintaxis para las palabras `this`, `new` y `super`, que se verá un poco más adelante.

La característica principal de estas clases internas es que cada objeto de la clase interna existe siempre dentro de un y sólo un objeto de la clase contenedora. Un objeto de la clase contenedora puede estar relacionado con uno o más objetos de la clase interna. Tener esto presente es muy importante para entender las características que se explican a continuación.

Relación entre las clases interna y contenedora respecto al acceso a las variables miembro:

1. Debido a la relación uno a uno, los métodos de la clase interna ven directamente las variables miembro del objeto de la clase contenedora, sin necesidad de cualificarlos.
2. Sin embargo, los métodos de la clase contenedora no ven directamente las variables miembro de los objetos de la clase interna: necesitan cualificarlos con una referencia a los correspondientes objetos. Esto es consecuencia de la relación uno a varios que existe entre los objetos de la clase contenedora y los de la clase interna.
3. Otras clases diferentes de las clases contenedora e interna pueden utilizar directamente los objetos de la clase interna, sin cualificarlos con el objeto o el nombre de la clase contenedora. De hecho, se puede seguir accediendo a los objetos de la clase interna aunque se pierda la referencia al objeto de la clase contenedora con el que están asociados.

Respecto a los permisos de acceso:

1. Las **clases** internas pueden también ser **private** y **protected** (las clases normales sólo pueden ser `public` y `package`). Esto permite nuevas posibilidades de encapsulación.
2. Los métodos de las clases internas acceden directamente a todos los miembros, incluso `private`, de la clase contenedora.
3. También la clase contenedora puede acceder –si dispone de una referencia- a todas las variables miembro (incluso `private`) de sus clases internas.
4. Una clase interna puede acceder también a los miembros (incluso `private`) de otras clases internas definidas en la misma clase contenedora.

Otras características de las clases internas son las siguientes:

1. Una clase interna miembro puede contener otra clase interna miembro, hasta el nivel que se desee (aunque no se considera buena técnica de programación utilizar muchos niveles).
2. En la clase interna, la palabra **this** se refiere al objeto de la propia clase interna. Para acceder al objeto de la clase contenedora se utiliza **ClaseContenedora.this**.
3. Para crear un nuevo objeto de la clase interna se puede utilizar **new**, precedido por la referencia al objeto de la clase contenedora que contendrá el nuevo objeto: **unObjCC.new()**. El tipo del objeto es el nombre de la clase contenedora seguido del

nombre de la clase interna, como por ejemplo:

```
ClaseCont.ClaseInt unObjClInt = unObjClaCont.new ClaseInt(...);
```

Supóngase como ejemplo adicional que B es una clase interna de A y que C es una clase interna de B. La creación de objetos de las tres clases se puede hacer del siguiente modo:

```
A a = new A(); // se crea un objeto de la clase A
A.B b = a.new B(); // b es un objeto de la clase interna B dentro de a
A.B.C c = b.new C(); // c es un objeto de la clase interna C dentro de b
```

4. Nunca se puede crear un objeto de la clase interna sin una referencia a un objeto de la clase contenedora. Los constructores de la clase interna tienen como argumento oculto una referencia al objeto de la clase contenedora.
5. El nuevo significado de la palabra **super** es un poco complicado: Si una clase deriva de una clase interna, su constructor no puede llamar a **super()** directamente. Ello hace que el compilador no pueda crear un constructor por defecto. Al constructor hay que pasarle una referencia a la clase contenedora de la clase interna super-clase, y con esa referencia, `ref`, llamar a `ref.super()`.

Las clases internas pueden derivar (heredar) de otras clases diferentes de la clase contenedora. En este caso, conviene tener en cuenta las siguientes reglas:

1. Las clases internas constituyen como una segunda jerarquía de clases en Java: por una parte están en la clase contenedora y ven sus variables; por otra parte pueden derivar de otra clase que no tenga nada que ver con la clase contenedora. Es muy importante evitar **conflictos con los nombres**. En caso de conflicto entre un nombre heredado y un nombre en la clase contenedora, el nombre heredado debe tener prioridad.
2. En caso de conflicto de nombres, Java obliga a utilizar la referencia **this** con un nuevo significado: para referirse a la variable o método miembro heredado se utiliza **this.name**, mientras que se utiliza **NombreClaseCont.this.name** para el miembro de la clase contenedora..
3. Si una clase contenedora deriva de una super-clase que tiene una clase interna, la clase interna de la sub-clase puede a su vez derivar de la clase interna de la super-clase y redefinir todos los métodos que necesite. La casuística se puede complicar todo lo que se desee, pero siempre hay que recomendar hacer las cosas lo más sencillas que sea posible.

El uso de las clases internas miembro tiene las siguientes restricciones:

1. Las clases internas no pueden tener el mismo **nombre** que la clase contenedora o package.
2. Tampoco pueden tener **miembros static**: variables, métodos o clases.

A continuación se presenta un ejemplo completo de utilización de clases internas miembro:



```

/ fichero ClasesInternas.java
// clase contenedora
class A {
    int i = 1; // variable miembro

    public A(int i) {
        this.i = i;
    } // constructor
    // los métodos de la clase contenedora necesitan una
    // referencia a los objetos de la clase interna

    public void printA(B unB) {
        System.out.println("i=" + i + " unB.j=" + unB.j); // sí acepta unB.j
    }

    // la clase interna puede tener cualquier visibilidad. Con private da error
    // porque main() no puede acceder a la clase interna
    protected class B {
        int j = 2;

        public B(int j) {
            this.j = j;
        } // constructor

        public void printB() {
            System.out.println("i=" + i + " j=" + j); // sí sabe qué es j
        }
    } // fin clase B
} // fin clase contenedora A

class ClasesInternas {
    public static void main(String[] arg) {
        A a1 = new A(11);
        A a2 = new A(12);
        println("a1.i=" + a1.i + " a2.i=" + a2.i);
        // forma de crear objetos de la clase interna
        // asociados a un objeto de la clase contenedora
        A.B b1 = a1.new B(-10), b2 = a1.new B(-20);
        // referencia directa a los objetos b1 y b2 (sin cualificar).
        println("b1.j=" + b1.j + " b2.j=" + b2.j);
        // los métodos de la clase interna pueden acceder directamente a
        // las variables miembro del objeto de la clase contenedora
        b1.printB(); // escribe: i=11 j=-10
        b2.printB(); // escribe: i=11 j=-20
        // los métodos de la clase contenedora deben recibir referencias
    }
}

```

```
// a los objetos de la clase interna, para que puedan identificarlos
a1.printA(b1);
a1.printA(b2);
A a3 = new A(13);
A.B b3 = a3.new B(-30);
println("b3.j=" + b3.j);
a3 = null; // se destruye la referencia al objeto de la clase contenedora
b3.printB(); // escribe: i=13 j=-30
a3 = new A(14); // se crea un nuevo objeto asociado a la referencia a3
// b3 sigue asociado al anterior objeto de la clase contenedora
b3.printB(); // escribe: i=13 j=-30
} // fin de main()

public static void println(String str) {
    System.out.println(str);
}
} // fin clase ClasesInternas
```

# Clases internas locales

Las clases internas locales o simplemente clases locales no se declaran dentro de otra clase al máximo nivel, sino **dentro de un bloque de código, normalmente en un método**, aunque también se pueden crear en un **inicializador static** o de objeto.

Las principales características de las clases locales son las siguientes:

1. Como las variables locales, las clases locales **sólo son visibles y utilizables en el bloque de código en el que están definidas**. Los objetos de la clase local deben ser creados en el mismo bloque en que dicha clase ha sido definida. De esta forma se puede acercar la definición al uso de la clase.
2. Las clases internas locales tienen **acceso a todas las variables miembro y métodos de la clase contenedora**. Pueden ver también los miembros heredados, tanto por la clase interna local como por la clase contenedora.
3. Las clases locales pueden utilizar las **variables locales y argumentos** de métodos visibles en ese bloque de código, pero sólo si son **final** (en realidad la clase local trabaja con sus copias de las variables locales y por eso se exige que sean final y no puedan cambiar).
4. Un objeto de una clase interna local sólo puede existir en relación con un objeto de la clase contenedora, que debe existir previamente.
5. La palabra **this** se puede utilizar en la misma forma que en las clases internas miembro, pero no las palabras **new** y **super**.

Restricciones en el uso de las clases internas locales:

1. No pueden tener el mismo **nombre** que ninguna de sus clases contenedoras.
2. No pueden definir variables, métodos y clases **static**.
3. No pueden ser declaradas public, protected, private o package, pues su visibilidad es siempre la de las variables locales, es decir, la del bloque en que han sido definidas.

Las clases internas locales se utilizan para definir [clases Adapter en AWT](#). A continuación se presenta un ejemplo de definición de clases internas locales:

```
// fichero ClasesIntLocales.java
// Este fichero demuestra cómo se crean clases locales
class A {
    int i = -1; // variable miembro
    // constructor

    public A(int i) {
        this.i = i;
    }

    // definición de un método de la clase A
    public void getAi(final long k) { // argumento final
        final double f = 3.14; // variable local final
    }
}
```

```

        // definición de una clase interna local
        class BL {
            int j = 2;

            public BL(int j) {
                this.j = j;
            } // constructor

            public void printBL() {
                System.out.println(" j=" + j + " i=" + i + " f=" + f + " k=" + k);
            }
        } // fin clase BL
        // se crea un objeto de BL
        BL bl = new BL(2 * i);
        // se imprimen los datos de ese objeto
        bl.printBL();
    } // fin getAi
} // fin clase contenedora A

class ClasesIntLocales {
    public static void main(String[] arg) {
        // se crea dos objetos de la clase contenedora
        A a1 = new A(-10);
        A a2 = new A(-11);
        a1.getAi(1000); // se crea y accede a un objeto de la clase local
        a2.getAi(2000);
    } // fin de main()

    public static void println(String str) {
        System.out.println(str);
    }
} // fin clase ClasesIntLocales

```

# Clases anónimas

Las clases anónimas son muy similares a las clases internas locales, pero sin nombre. En las clases internas locales primero se define la clase y luego se crean uno o más objetos. En las clases anónimas se unen estos dos pasos: Como la clase no tiene nombre **sólo se puede crear un único objeto**, ya que las clases anónimas no pueden definir constructores. Las clases anónimas se utilizan con mucha frecuencia en comparadores, interfaces funcionales y en AWT para definir clases y objetos que gestionen los eventos de los distintos componentes de la interfaz de usuario. No hay interfaces anónimas.

Formas de definir una clase anónima:

1. Las clases anónimas requieren una extensión de la palabra clave **new**. **Se definen en una expresión de Java, incluida en una asignación o en la llamada a un método**. Se incluye la palabra **new** seguida de la definición de la clase anónima, entre llaves {...}.
2. Otra forma de definir las es mediante la palabra **new** seguida del nombre de la clase de la que hereda (sin **extends**) y la definición de la clase anónima entre llaves {...}. El nombre de la superclase puede ir seguido de argumentos para su constructor (entre paréntesis, que con mucha frecuencia estarán vacíos pues se utilizará un constructor por defecto).
3. Una tercera forma de definir las es con la palabra **new** seguida del nombre de la **interfaz** que implementa (sin **implements**) y la definición de la clase anónima entre llaves {...}. **En este caso la clase anónima deriva de Object**. El nombre de la interfaz va seguido por paréntesis vacíos, pues el constructor de Object no tiene argumentos. Para las clases anónimas compiladas el compilador produce ficheros con un nombre del tipo **ClaseContenedora\$1.class**, asignando un número correlativo a cada una de las clases anónimas.

Conviene ser muy cuidadoso respecto a los aspectos tipográficos de la definición de clases anónimas, pues al no tener nombre dichas clases suelen resultar difíciles de leer e interpretar. Se aconseja utilizar las siguientes **normas tipográficas**:

1. Se aconseja que la palabra **new** esté en la misma línea que el resto de la expresión.
2. Las **llaves** se abren en la misma línea que **new**, después del cierre del paréntesis de los argumentos del constructor.
3. El cuerpo de la clase anónima se debe **sangrar** o indentar respecto a las líneas anteriores de código para que resulte claramente distinguible.
4. El **cierre de las llaves** va seguido por el resto de la expresión en la que se ha definido la clase anónima. Esto puede servir como indicación tipográfica del cierre. Puede ser algo así como `};` o `});`

A continuación se presenta un ejemplo de definición de clase anónima en relación con AWT, donde **la clase anónima**, que se pasa como parámetro al método `addActionListener()`, **deriva de Object e implementa la interface ActionListener**.

```
unObjeto.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent e) {
```

```
...  
}  
});
```

Las clases anónimas se utilizan en lugar de clases locales para clases con muy poco código, de las que sólo hace falta un objeto. No pueden tener constructores, pero sí inicializadores static o de objeto. Además de las restricciones citadas, tienen restricciones similares a las clases locales.