# UNIDAD 7: UTILIZACIÓN AVANZADA DE CLASES

## Módulo Profesional: Programación

# ÍNDICE

# RESUMEN INTRODUCTORIO

En esta unidad se va a tratar en primer lugar el concepto de composición de clases. Posteriormente se tratará el importante concepto de Herencia, en la que se verán los conceptos de superclases, clases padres, subclases y clases hijas. Más adelante se introduce el concepto de clases y métodos abstractos y clases y métodos finales. Se verá a continuación la sobreescritura de métodos. Finalizaremos revisando el tratamiento de los constructores en la herencia. Para cada uno de los apartados veremos ejemplos en el lenguaje de programación Java.

# INTRODUCCIÓN

Un conjunto de objetos aislados tiene escasa capacidad para resolver un problema. En una aplicación real los objetos colaboran e intercambian información, existiendo distintos tipos de relaciones entre ellos. La relación de herencia es un mecanismo que permite sobreescribir o extender las funcionalidades de una clase ya existente. De este modo se favorece la reutilización del código, se evitan redundancias, ya que no se repite de forma innecesaria la escritura de código ya implementado y se modela de forma más fidedigna la realidad.

# CASO INTRODUCTORIO

El equipo de trabajo que diriges en la empresa en la que trabajas, está desarrollando un proyecto en el que se han escrito todas las superclases necesarias. Se ha llegado al punto en el que hay que implementar las subclases necesarias para no tener que reescribir código que ya se ha escrito y codificar aquellas otras que están compuestas de clases ya existentes.

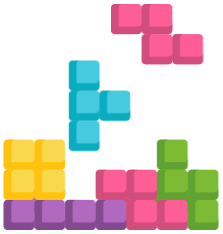
Al finalizar la unidad el alumnado:

- Conocerá los conceptos de herencia y composición de clases.
- Será capaz de implementar dichos conceptos en un lenguaje de programación orientado a objetos.
- Identificará cuándo utilizar clases y métodos abstractos y finales.
- Conocerá el concepto de método constructor y su aplicación en las relaciones entre clases.

# 1. COMPOSICIÓN DE CLASES

La composición es un tipo de relación **dependiente** en la que un objeto más complejo es conformado por objetos más pequeños. En esta situación, la frase "***Tiene un***", debe tener sentido:

**El coche** *tiene* **llantas**
**El portátil** *tiene* **un teclado**.



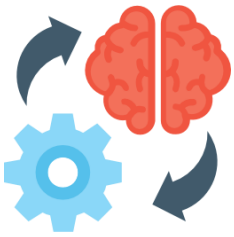
**EJEMPLO PRÁCTICO**

Se representa la relación: El **portátil** *tiene un* **teclado**.

```
public class Portatil {

    private String fabricante;
    private String modelo;
    private String servicioTecnico;
    private Teclado teclado = new Teclado();

      public Portatil() {
            // Lo que haga el constructor...
      }

}
```

La composición generalmente se usa cuando se quieren tener las características de una **clase existente** dentro de otra **clase**, pero no en el interfaz. Esto es, aloja un objeto para implementar características en su clase, pero el usuario de su clase ve el interfaz que se ha definido, en vez del interfaz de la clase original. Para hacer esto, se sigue el típico patrón de alojar objetos privados de clases existentes en su nueva clase.

En ocasiones, sin embargo, tiene sentido permitir que el usuario de la clase acceda a la composición de su clase, esto es, hacer públicos los miembros objeto. Los miembros objeto usan su control de accesos, entonces es seguro y cuando el usuario conoce que está formando un conjunto de piezas, hace que la interfaz sea más fácil de entender.



**RECUERDA**

Los objetos que **componen** a la clase contenedora deben **existir** desde el principio. (También pueden ser creados en el constructor, no sólo al momento de declarar las variables como se muestra en el ejemplo).

No hay momento en que la **clase contenedora** pueda existir sin alguno de sus objetos componentes. Por lo que la existencia de estos objetos no debe ser abiertamente manipulada desde el exterior de la clase.

**ARTÍCULO DE INTERÉS**

Para ampliar información sobre la composición y ver más ejemplos se recomienda realizar la lectura:

- Composición

## 2. HERENCIA

**La herencia** es uno de los 4 pilares de la programación orientada a objetos (POO) junto con la **Abstracción**, **Encapsulación** y **Polimorfismo**. 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.

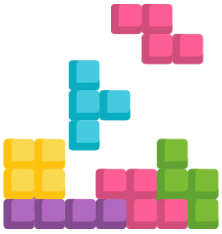
Java permite **heredar** a las clases **características** y **conductas** de una o varias clases denominadas base. Las clases que heredan de clases base se denominan **derivadas**, estas a su vez pueden ser clases bases para otras clases derivadas. Se establece así una **clasificación jerárquica**, similar a la existente en Biología con los animales y las plantas.

La herencia ofrece una **ventaja** importante, permite la **reutilización del código**. Una vez que una clase ha sido depurada y probada, el código fuente de dicha clase no necesita modificarse. Su funcionalidad se puede cambiar derivando una nueva clase que herede la funcionalidad de la clase base y le añada otros comportamientos. Reutilizando el código existente, el programador ahorra tiempo y dinero, ya que solamente tiene que verificar la nueva conducta que proporciona la clase derivada.

**ARTÍCULO DE INTERÉS**

En el siguiente archivo PDF encontrará más información sobre la herencia y cómo implementarla en Java:

- Herencia en Java



**EJEMPLO PRÁCTICO**

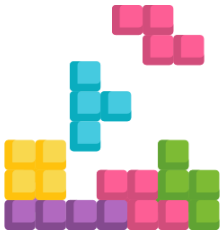
Se va a ver un ejemplo simple de la herencia. Para este caso se creará una clase que se heredará en la cual se encuentre el apellido, y otra clase donde se asignará el nombre de una persona. La clase que se heredará:

```
package herencia;

public class ClaseHeredada {

	String Apellido;

	public ClaseHeredada(String Dato) {
		this.Apellido = Dato;
	}

}
```

Una vez que se tiene la clase que se va a heredar, se crea la clase en la que se encontrará el nombre de la persona y se le asigna el siguiente código:

```
package herencia;

public class Herencia extends ClaseHeredada {
	String Nombre;

	public Herencia(String Texto) {
		super(Texto);
	}

	public void setPersona(String NombrePer) {
		this.Nombre = NombrePer + this.Apellido;
	}
```

```
        public String getPersona() {
              return Nombre;
        }

}
```



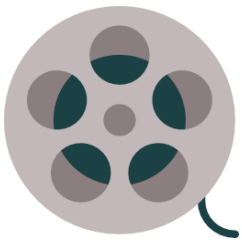
**EJEMPLO PRÁCTICO**

Se usa extends para indicar que se está heredando la clase "ClaseHeredada" en la que se encuentra el apellido. La palabra super se usa para indicar que estamos instanciando al constructor de la clase que se está heredando y los métodos setPersona y getPersona son los que se heredarán y se usarán para obtener el nombre de la persona.

Una vez realizado este proceso, se usará la clase instanciando en el método main de la siguiente forma:

```
public static void main(String[] args) {
  Herencia X=new Herencia(" Arias Figueroa");
  X.setPersona("Kevin Arnold");
  System.out.println(X.getPersona());
}
```

Se instancia la clase "Herencia" pasándole a su constructor el apellido de la persona, lo que hará que internamente se pase a la clase superior "ClaseHeredada", luego se le asignará el nombre haciendo uso del método setPersona y se imprimirá mediante el método getPersona.



**VIDEO DE INTERÉS**

Un tutorial muy interesante de visionar sobre herencia en Java se puede encontrar en YouTube en la siguiente URL:

- Herencia

# 3. SUPERCLASES Y SUBCLASES

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 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, por lo que se puede decir que Object es la superclase de todas las clases en Java.

Las subclases heredan el **estado** y el **comportamiento** en forma de las **variables** y los **métodos** de su superclase. La subclase puede utilizar los ítems heredados de su superclase tal y como son, o puede **modificarlos** o **sobreescribirlos**. Por eso, según se va bajando por el árbol de la herencia, las clases se convierten en más y **más especializadas**.

Una **subclase** es una clase que desciende de otra clase. Una subclase hereda el estado y el comportamiento de todos sus ancestros.

El término **superclase** se refiere a la clase que es el ancestro más directo, así como a todas las clases ascendentes.

Se declara que una clase es una subclase de otra clase dentro de La declaración de la Clase.

Se supone que se quiere crear una subclase llamada `SubClase` de otra clase llamada `SuperClase`. Se escribiría:

```
  public class SubClase extends SuperClase {

}
```

Esto declara que `SubClase` es una **subclase** de `SuperClase`. Y también declara implícitamente que `SuperClase` es la **superclase** de `SubClase`. Una subclase también hereda variables y miembros de las superclases de su superclase, y así a lo largo del árbol de la herencia.

**ENLACE DE INTERÉS**

Se entenderá mejor el concepto de subclase y superclase visitando el siguiente enlace:

- Programación Orientada a Objetos con Java

# 4. CLASES Y MÉTODOS ABSTRACTOS Y FINALES

En este apartado veremos las clases, juntos a sus métodos abstractos y también finales.

## 4.1 Clases y métodos abstractos

A veces, una clase que se ha definido representa un concepto abstracto y como tal, no debe ser ejemplarizado. Por ejemplo, la comida en la vida real. ¿Ha visto algún ejemplar de comida? No. Lo que ha visto son ejemplares de manzanas, pan, y chocolate. Comida representa un **concepto abstracto** de cosas que son comestibles. **No tiene sentido** que exista un ejemplar de comida.

En la programación orientada a objetos se podrían modelar conceptos abstractos, y, a su vez, no querer que se creen ejemplares de ellos.

Por ejemplo, la clase `Number` del paquete `java.lang` representa el concepto abstracto de número. Tiene sentido modelar números en un programa, pero no tiene sentido crear un objeto genérico de números. En su lugar, la clase `Number` sólo tiene sentido como superclase de otras clases como `Integer` y `Float` que implementan números de tipos específicos. Las clases como `Number`, que implementan conceptos abstractos y no deben ser ejemplarizadas, son llamadas **clases abstractas**. Una clase abstracta es una clase que **sólo puede tener subclases**--no puede ser ejemplarizada.

Para declarar que una clase es una clase abstracta, se utiliza la palabra clave **`abstract`** en la declaración de la clase.

```
abstract class Number {
    . . .
}
```

Si se intenta ejemplarizar una clase abstracta, el compilador mostrará un **error** similar a este y no compilará el programa.

```
AbstractTest.java:6: class AbstractTest is an abstract class. It can't be
instantiated.
        new AbstractTest();
        ^
1 error
```

Una clase abstracta puede contener **métodos abstractos**, esto es, métodos que **no tienen implementación**. De esta forma, una clase abstracta puede definir un interface de programación completo, incluso proporciona a sus subclases la declaración de todos los métodos necesarios para implementar el interface de programación. Sin embargo, las clases abstractas pueden dejar algunos detalles o toda la implementación de aquellos métodos a sus subclases.

**ENLACE DE INTERÉS**

Lo mejor para aprender a utilizar las clases y métodos abstractos es mediante ejemplos. Es por esto que se recomienda visitar y realizar la lectura de la siguiente web:

- Clases y métodos abstractos en Java. Abstract class

Se va a detallar a continuación un ejemplo de cuándo sería necesario crear una clase abstracta con métodos abstractos. En una aplicación de dibujo orientada a objetos, se pueden dibujar círculos, rectángulos, líneas, etc… Cada uno de esos objetos gráficos comparten ciertos estados (posición, caja de dibujo) y comportamiento (movimiento, redimensionado, dibujo). Se pueden aprovechar esas similitudes y declararlos todos a partir de un mismo objeto padre - `ObjetoGrafico`.

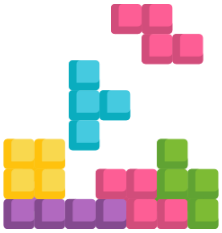
Sin embargo, los objetos gráficos también tienen diferencias substanciales: dibujar un círculo es bastante diferente a dibujar un rectángulo. Los objetos gráficos no pueden compartir estos tipos de estados o comportamientos.

Por otro lado, todos los objetos `ObjetoGrafico` deben saber cómo dibujarse a sí mismos; se diferencian en cómo se dibujan unos y otros. Esta es la situación perfecta para una clase abstracta.

Primero se debe declarar una clase abstracta, `ObjetoGrafico`, para proporcionar las variables "miembro" y los métodos que van a ser compartidos por todas las subclases, como la posición actual y el método **`moverA()`**.

También se deberían declarar métodos abstractos como **`dibujar()`**, que necesita ser implementado por todas las subclases, pero de manera completamente diferente (no tiene sentido crear una implementación por defecto en la superclase).

Una clase abstracta **no necesita** contener un método abstracto. Pero **todas las clases que contengan un método abstracto** o no proporcionen implementación para cualquier método abstracto declarado en sus superclases, deben ser declaradas como una **clase abstracta**.



**EJEMPLO PRÁCTICO**

La clase `ObjetoGrafico` se parecería a:

```
public abstract class ObjetoGrafico {
     int x, y;...

       void moverA(int nuevaX, int nuevaY) {
            ...
           }

       abstract void dibujar();

}
```

Todas las subclases no abstractas de `ObjetoGrafico` como son `Circulo` o `Rectangulo` deberán proporcionar una implementación para el método **`dibujar()`**.

Clase `Circulo`:

```
public class Circulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
```

Clase `Rectangulo`:

```
public class Rectangulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
```

## 4.2 Clases y métodos finales

Se puede declarar que una clase sea **final**; esto es, que la clase **no pueda tener subclases**. Existen (al menos) dos razones por las que se querría hacer esto:

- **Seguridad**: Un mecanismo que los hackers utilizan para atacar sistemas es crear subclases de una clase y luego sustituirla por el original. Las subclases parecen y sienten como la clase original pero hacen cosas bastante diferentes, probablemente causando daños u obteniendo información privada. Para prevenir esta clase de subversión, se puede declarar que la clase sea final y así prevenir que se cree cualquier subclase.
  La clase `String` del paquete `java.lang` es una clase final sólo por esta razón. La clase `String` es tan vital para la operación del compilador y del intérprete que el sistema Java debe garantizar que siempre que un método o un objeto utilicen un `String`, obtenga un objeto `java.lang.String` y no algún otro string. Esto asegura que ningún string tendrá propiedades extrañas, inconsistentes o indeseables.
  Si se intenta compilar una subclase de una clase final, el compilador mostrará un **mensaje de error** y **no compilará** el programa.
  Además, los bytecodes verifican que no está teniendo lugar una subversión, al nivel de byte comprobando que una clase no es una subclase de una clase final.
- **Diseño**: Otra razón por la que se podría querer declarar una clase final son razones de diseño orientado a objetos. Se podría pensar que una clase es "perfecta" o que, conceptualmente hablando, la clase no debería tener subclases.

Para especificar que una clase es una clase final, se utiliza la palabra clave **`final`** antes de la palabra clave **`class`** en la declaración de la clase. Por ejemplo, si quisiéramos declarar `AlgoritmodeAjedrez` como una clase final (perfecta), la declaración se parecería a esto.

```
final class AlgoritmodeAjedrez {
    . . .
}
```

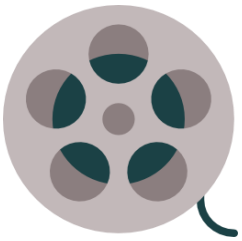
Cualquier intento posterior de crear una subclase de `AlgoritmodeAjedrez` resultará en el siguiente error del compilador.

```
Chess.java:6:    Can't    subclass    final    classes:    class
AlgoritmodeAjedrez
class MejorAlgoritmodeAjedrez extends AlgoritmodeAjedrez {
      ^
1 error
```

Para los métodos finales, si realmente lo que se quiere es proteger algunos métodos de una clase para que no sean sobreescritos, se puede utilizar la palabra clave **final** en la declaración del método para indicar al compilador que este método **no puede ser sobreescrito** por las subclases.

Se podría desear hacer que un método fuera final si el método tiene una implementación que no debe ser cambiada y que es crítica para el estado consistente del objeto. Por ejemplo, en lugar de hacer AlgoritmodeAjedrez como una clase final, podríamos hacer **`siguienteMovimiento()`** como un método final.

```
class AlgoritmodeAjedrez {
    . . .
    final void siguienteMovimiento(Pieza piezaMovida,
                   PosicionenTablero nuevaPosicion) {
    }
    . . .
}
```



**VIDEO DE INTERÉS**

Repase el uso de los modificadores static y final con el siguiente vídeo:

- Static y Final

# 5. SOBREESCRITURA DE MÉTODOS

Una subclase puede:

- Sobreescribir completamente la implementación de un método de una superclase
- Añadir implementación a un método de la superclase.
- Métodos que una subclase no puede sobreescribir.

## 5.1 Reemplazar la implementación de un método de una superclase

Algunas veces, una subclase querría reemplazar completamente la implementación de un método de su superclase. De hecho, muchas superclases proporcionan implementaciones de métodos vacías con la esperanza de que la mayoría, si no todas, de sus subclases reemplacen completamente la implementación de ese método.

Un ejemplo de esto es el método **run()** de la clase `Thread`. La clase `Thread` proporciona una implementación vacía (el método no hace nada) para el método **run()**, porque, por definición, este método depende de la subclase. La clase `Thread` posiblemente no puede proporcionar una implementación medianamente razonable del método **run()**.

Para reemplazar completamente la implementación de un método de la superclase, simplemente se llama a un método con el mismo nombre que el del método de la superclase y se sobreescribe el método con la misma firma que la del método sobreescrito.

```
class ThreadSegundoPlano extends Thread {
    void run() {
        . . .
    }
}
```

La clase `ThreadSegundoPlano` sobreescribe completamente el método **run()** de su superclase y reemplaza completamente su implementación.

**ENLACE DE INTERÉS**

Para entender en su totalidad el código fuente escrito en Java, es imprescindible tener muy claro el concepto de sobreescritura de métodos. Amplíe información realizando la siguiente lectura:

- Sobreescritura de métodos

## 5.2 Añadir implementación a un método de la superclase

Otras veces una subclase querrá mantener la implementación del método de su superclase y posteriormente **ampliar** algún comportamiento específico de la subclase.

Por ejemplo, los métodos constructores de una subclase lo hacen normalmente: la subclase quiere preservar la inicialización realizada por la superclase, pero proporciona inicialización adicional específica de la subclase.

Se supone que se quiere crear una subclase de la clase `Windows` del paquete `java.awt`. La clase `Window` tiene un constructor que requiere un argumento del tipo `Frame` que es el padre de la ventana.

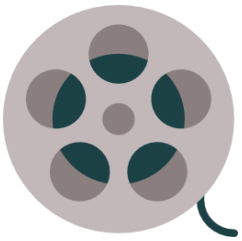
```
public Window(Frame parent)
```

Este constructor realiza alguna inicialización en la ventana para que trabaje dentro del sistema de ventanas. Para asegurarse de que una subclase de `Window` también trabaja dentro del sistema de ventanas, se deberá proporcionar un constructor que realice la misma inicialización.

Mucho mejor que intentar recrear el proceso de inicialización que ocurre dentro del constructor de `Window` se podría utilizar lo que la clase `Window` ya hace. Se puede utilizar el código del constructor de `Window` llamándolo desde dentro del constructor de la subclase `Window`.

```
class Ventana extends Window {

    public Ventana(Frame parent) {

        super(parent);

        . . .

            // Ventana especifica su inicialización aquí

        . . .

    }

}
```

El constructor de **Ventana** llama primero al constructor de su superclase, y no hace nada más. Típicamente, este es el comportamiento deseado de los constructores. Las superclases deben tener la oportunidad de realizar sus tareas de inicialización antes que las de su subclase. Otros tipos de métodos podrían llamar al constructor de la superclase al final del método o en el medio.



**VIDEO DE INTERÉS**

Curso muy interesante sobre los métodos **constructores** y la **sobreescritura** de métodos:

- Métodos Constructor y sobreescritura de Métodos

## 5.3 Métodos que una subclase no puede sobreescribir

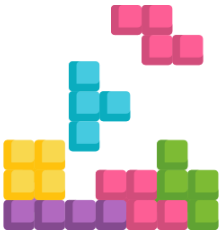
Los métodos que una subclase no puede sobreescribir son los que se explican a continuación:

- Una subclase no puede sobreescribir métodos que hayan sido declarados como **final** en la superclase (por definición, los métodos finales no pueden ser sobreescritos). Si intentamos sobreescribir un método final, el compilador mostrará un mensaje y no compilará el programa.
- Una subclase tampoco puede sobreescribir métodos que se hayan declarado como **static** en la superclase. En otras palabras, una subclase no puede sobreescribir un método de clase.

# 6. CONSTRUCTORES Y HERENCIA

A diferencia de lo que ocurre con los métodos y atributos no privados, **los constructores no se heredan**. Además de esta característica, deben tenerse en cuenta algunos aspectos sobre el comportamiento de los constructores dentro del contexto de la herencia, ya que dicho comportamiento es sensiblemente distinto al del resto de métodos.

Cuando existe una relación de herencia entre diversas clases y se crea un objeto de una subclase S, se ejecuta no sólo el constructor de S sino también el de todas las superclases de S. Para ello se ejecuta en primer lugar el constructor de la clase que ocupa el nivel más alto en la jerarquía de herencia y se continúa de forma ordenada con el resto de las subclases.

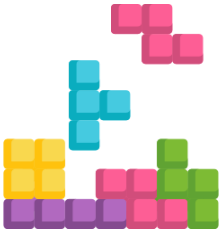


**EJEMPLO PRÁCTICO**

El siguiente ejemplo ilustra este comportamiento:

```
class A {

    A() {
        System.out.println("En A");
    }
}

class B extends A {

    B() {
        System.out.println("En B");
    }
}
```

```java
class C extends B {

    C() {
        System.out.println("En C");
    }
}

class Constructores_y_Herencia {

    public static void main(String[] args) {
        C obj = new C();
    }
}
```



**EJEMPLO PRÁCTICO**

Es posible que una misma clase tenga más de un constructor (sobrecarga del constructor), tal y como se muestra en el siguiente ejemplo:

```java
class A {

    A() {
        System.out.println("En A");
    }

    A(int i) {
        System.out.println("En A(i)");
    }
}

class B extends A {

    B() {
        System.out.println("En B");
    }

    B(int j) {
        System.out.println("En B(j)");
    }
}
```

**ENLACE DE INTERÉS**

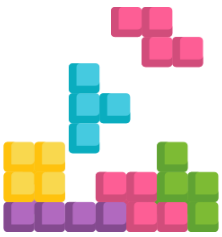
El siguiente enlace nos lleva a un tutorial muy interesante sobre herencia y constructores en Java:

- Subclases y Herencia

La cuestión que se plantea es: ¿qué constructores se invocarán cuando se ejecuta la sentencia **`B obj = new B(5);`**? Puesto que hemos creado un objeto de tipo B al que le pasamos un entero como parámetro, parece claro que en la clase B se ejecutará el constructor **`B(int j)`**. Sin embargo, puede haber confusión acerca de qué constructor se ejecutará en A. La regla en este sentido es clara: mientras no se diga explícitamente lo contrario, en la superclase se ejecutará siempre el constructor sin parámetros. Por tanto, ante la sentencia **`B obj = new B(5);`** se mostraría:

```
En A
En B(j)
```

Hay, sin embargo, un modo de cambiar este comportamiento por defecto para permitir ejecutar en la superclase un constructor diferente. Para ello debemos hacer uso de la sentencia **`super()`**. Esta sentencia invoca uno de los constructores de la superclase, el cual se escogerá en función de los parámetros que contenga **`super()`**.



**EJEMPLO PRÁCTICO**

En el siguiente ejemplo se especifica de forma explícita el constructor que deseamos ejecutar en A:

```
class A {

    A() {
        System.out.println("En A");
    }

    A(int i) {
        System.out.println("En A(i)");
    }
}
```

```
class B extends A {

    B() {
        System.out.println("En B");
    }

    B(int j) {
        super(j);   // Ejecutar en la superclase un constructor
que acepte un entero
        System.out.println("En B(j)");
    }
}
```

Ha de tenerse en cuenta que en el caso de usar la sentencia **super()**, ésta deberá ser obligatoriamente **la primera sentencia** del constructor. Esto es así porque se debe respetar el orden de ejecución de los constructores comentado anteriormente.

En resumen, cuando se crea una instancia de una clase, para determinar el constructor que debe ejecutarse en cada una de las superclases, en primer lugar, se exploran los constructores en **orden jerárquico ascendente** (desde la subclase hacia las superclases). Con este proceso se decide el constructor que debe ejecutarse en cada una de las clases que componen la jerarquía. Si en algún constructor no aparece explícitamente una llamada a **super(*lista_de_argumentos*)** se entiende que de forma implícita se está invocando a **super()** (constructor sin parámetros de la superclase). Finalmente, una vez que se conoce el constructor que debe ejecutarse en cada una de las clases que componen la jerarquía, éstos se ejecutan en orden jerárquico descendente (desde la superclase hacia las subclases).

Podemos considerar que todas las clases en Java tienen de forma implícita un **constructor por defecto** sin parámetros y sin código. Ello permite crear objetos de dicha clase sin necesidad de incluir explícitamente un constructor. Por ejemplo, dada la clase:

```
class A {
    int i;
}
```

No hay ningún problema en crear un objeto:

```
A obj = new A();
```

Ya que, aunque no lo escribamos, la clase A lleva implícito un constructor por defecto:

```
class A {

    int i;

    A() {
        // Constructor por defecto
    }
}
```

Sin embargo, es importante saber que dicho constructor por defecto **se pierde** si escribimos cualquier otro constructor. Por ejemplo, dada la clase:

```
class A {

    int i;

    A(int valor) {
        i = valor;
    }
}
```

La sentencia **`A obj = new A();`** generaría un **error de compilación**, ya que en este caso no existe ningún constructor en A sin parámetros. Hemos perdido el constructor por defecto. Lo correcto sería, por ejemplo, **`A obj = new A(5);`** Esta situación debe tenerse en cuenta igualmente cuando exista un esquema de herencia.

Se tiene la siguiente jerarquía de clases:

```
class A {

    int i;

    A(int valor) {
        i = valor;
    }
}

class B extends A {

    int j;

    B(int valor) {
        j = valor;
    }
}
```

En este caso la sentencia **`B obj = new B(5)`**; generará igualmente un error ya que, puesto que no hemos especificado un comportamiento distinto mediante **`super()`**, en A debería ejecutarse el constructor sin parámetros, sin embargo, tal constructor no existe puesto que se ha perdido el constructor por defecto. La solución pasaría por sobrecargar el constructor de A añadiendo un constructor sin parámetros.

```
class A {

    int i;

    A() {
        i = 0;
    }

    A(int valor) {
        i = valor;
    }
}

class B extends A {

    int j;

    B(int valor) {
        j = valor;
    }
}
```

O bien indicar explícitamente en el constructor de B que se desea ejecutar en A un constructor que recibe un entero como parámetro, tal y como se muestra a continuación:

```
class A {

    int i;

    A(int valor) {
        i = valor;
    }
}

class B extends A {

    int j;

    B(int valor) {
        super(0);
        j = valor;
    }
}
```

# RESUMEN FINAL

Al desarrollar una aplicación en Java, podemos reutilizar código utilizando la herencia de clases. Es muy importante no volver a programar lo que ya existe y poder reutilizarlo.

En esta unidad se ha comenzado analizando los conceptos de composición y herencia de clases. Posteriormente se ha visto la jerarquía de clases definiendo superclases y subclases. A continuación, se han visto los conceptos de clases y métodos abstractos y clases y métodos finales.

Para finalizar esta unidad nos hemos adentrado en la sobreescritura de métodos y el funcionamiento de los constructores en la herencia.