

## **JAVA - Objetos**

1. Manejo de clases
2. Herencia
3. Abstracción
4. Encapsulamiento
5. Polimorfismo

## Definición de Paradigma de programación

Podemos definir a un paradigma de programación como un enfoque particular o criterio para la creación de software.

Hay diferentes paradigmas de programación, que determinan distintos estilos de programación y diferentes formas de resolver un problema.

Para mostrar más claro los diferentes enfoques de los paradigmas, te presentamos un mismo ejemplo abordado desde ambos paradigmas.

*SEUDOCÓDIGO A MODO DE EJEMPLO:*

<b>INICIO Principal()</b>	<b>// Definimos el inicio del programa</b>
AlumnoNuevo = nuevo Alumno()	<b>// Llamamos a la clase Alumno, generando</b>
	<b>// el objeto AlumnoNuevo</b>
AlumnoNuevo.obtener()	<b>// Utilizamos el método obtener() de Alumno</b>

**FIN Principal**

<b>CLASE Alumno</b>	<b>// Creamos la clase alumno</b>
<b>Atributos</b>	<b>// Definimos sus atributos o propiedades</b>
<b>NOMBRE : CADENA</b>	<b>// Son tres Nombre, Apellido y Edad</b>
<b>APELLIDO: CADENA</b>	
<b>EDAD : ENTERO</b>	
<b>METODO Obtener ()</b>	<b>// Dentro de la clase creamos</b>
Mostrar (“Ingrese nombre”)	<b>// el método Obtener</b>
Ingresar ( <b>NOMBRE</b> )	
Mostrar (“Ingrese apellido”)	
Ingresar ( <b>APELLIDO</b> )	
Mostrar (“Ingrese edad”)	
Ingresar ( <b>EDAD</b> )	
<b>Hasta que EDAD &gt; 0</b>	
<b>FIN METODO</b>	
<b>FIN CLASE</b>	

En cambio, en la versión de Objetos, habrás notado que hay algo nuevo.

Hay una clase (Alumno) que contiene atributos (Nombre, Apellido y Edad) y métodos (Obtener()).

Es decir que, en este mismo componente (en la clase (Alumno)), vamos a tener los datos y la forma en que se manipulan esos datos mediante el método (Obtener()).

Luego, en el programa principal, se crea un objeto (AlumnoNuevo) en base a una clase (Alumno).

También habrás notado que no hay llamadas a funciones, sino que hay referencias a los métodos definidos en la clase Alumno utilizando el objeto AlumnoNuevo.obtener(). Es decir que creamos un Objeto a partir de una clase y luego usamos el objeto para llamar a los métodos.

En el mismo objeto se encuentran los datos y sus comportamientos, de tal manera que, ante un cambio en los datos, no se deberá cambiar el programa principal, sino que los cambios deberán realizarse solamente en la clase.

En programación un paradigma, es un estilo, una forma, un modo, una práctica y hasta una visión del desarrollo de un sistema, cuando nos referimos a objetos y clases.

Aunque esta definición teórica no contribuya mucho a la realización de código, nunca olvides que un Paradigma no deja de marcar y ser una guía de los pasos a seguir en cualquier acción a realizar.

Entonces, la programación orientada a **objetos** es un enfoque de programación que tiene un estilo en el cual cada programa es pensado y escrito como un objeto y además, se forma por una serie o un conjunto de componentes, que también son **objetos** y que con sus datos (**atributos o propiedades**) y con la posibilidad de realizar acciones (**métodos**) cooperan y se relacionan para lograr el funcionamiento de la aplicación completa.

### ¿Cuáles son las diferencias de la POO con el paradigma estructurado?

La principal diferencia con la programación estructurada tradicional es que ahora deberás pensar simultáneamente en el conjunto de atributos que tendrá una clase y en los métodos para poder tratarlos. Es decir, que los datos son tan importantes como su comportamiento y deben ir íntimamente entrelazados.

En cambio, en el paradigma estructurado, los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida.

### Y entonces, ¿que son los Objetos ?

Antes de ver la definición de objeto, te pedimos que te tomes un momento para observar los objetos y seres vivos que hay a tu alrededor, por ejemplo, una mesa, una silla, tu perro, cualquier elemento.

Podrás notar que a cada uno de ellos se le puede asociar un conjunto de características, por ejemplo: color, peso, dimensiones, raza, etc.

En programación, el concepto de **objeto** agrupa cosas y seres vivos, para lo cual podemos indicar que cada objeto podrá tener un comportamiento propio generado por él mismo o inducidos por otros objetos: el perro ladra, camina, salta, etc. Así, cada objeto tendrá sus propias características y comportamiento.

Un Objeto es un elemento que posee características, denominadas atributos, y comportamientos, denominados métodos, que puede interactuar con otros objetos del sistema, enviando mensajes, y tiene asociado una identidad que lo distingue entre todos los objetos del sistema.

Más técnicamente hablando:

• **Objeto:** es la entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos).

#### EJEMPLO

“Mi perro es de color marrón” ← ese es el dato

“Mi perro puede ladrar” ← ese es el método

En muchos casos, corresponden a los objetos reales del mundo que nos rodea, y también a objetos internos del sistema (del programa). Esta entidad **objeto** tiene como “estructura” a una **clase**.

#### EJEMPLO

“Un caniche es de la clase perro” ← Caniche es el objeto, Perro es la clase.

El **objeto** inicia su participación activa en el sistema cuando es llamada la **clase** que contiene su “estructura” por medio de una instancia. Este proceso, en realidad, constituye técnicamente una “instancia de una clase” y explicaremos el significado de “instancia” en las siguientes páginas, después de conocer bien el concepto de clase.

#### EJEMPLO

Para crear un “Caniche” necesito la clase “Perro” ← genera la instancia de la clase Perro y crea al  
objeto Caniche

Podríamos dejar como resumen de estos párrafos que un objeto es un ejemplo, un derivado o una muestra de una clase, que tiene todas o algunas de las características de la clase que representa y que puede o no interactuar con otros objetos.

### Clases

¿Y cuál es la relación entre las clases y los objetos?

Las Clases son definiciones de las propiedades y comportamientos que podrá tener un tipo de objeto. Instanciar una clase, realiza la lectura de estas definiciones y la creación de un objeto a partir de ellas.

Sigamos con los casos de los objetos de la vida real y el ejemplo de un perro.

Nosotros sabemos qué es un perro porque conocemos la idea de un perro. Todos los perros presentan las características y tienen el mismo comportamiento. Ahora bien, cada perro en particular se diferencia porque tiene diferentes atributos, por ejemplo, son de distinta raza, de distinto tamaño, de distinto peso, etc. Por lo tanto, la idea de un perro es el equivalente a una Clase.

Podemos ver a **una clase** como un modelo o plantilla que representa entidades o conceptos.

En una clase se definen los datos y el comportamiento que tendrá la entidad que representa. Los datos se denominan atributos, mientras que el comportamiento está definido por funciones y procedimientos que se denominan métodos. Una clase se utiliza para crear objetos, es por eso que también se dice que un objeto es una instancia de una clase.

#### EJEMPLO

La clase perro, puede indicar como atributo: pelo, color, altura, peso etc. que son todas propiedades pertenecientes a los perros.

También puede indicar si ladra, corre, come, duerme etc y esos son los métodos que pertenecen a esa clase.

Una clase es un modelo abstracto que define cómo será un objeto real.

¿Y en programación JAVA, como se escribe una clase? Probemos ahora con el ejemplo de un Automóvil.

### *Ejemplo Práctico*

```
public class Auto {                                // este es un ejemplo de como escribir la clase Auto
    String marca = "Ford";                        // comienza con una lista de atributos o propiedades
    String modelo = "Falcón";
    int año = 1975;
    int puertas = 4;

    public Auto() {                                // posee un método constructor que lo hace
arrancar                                           arrancar();
    }
    public String arrancar(){                      // el método arrancar avisa cuando
        return "en marcha";                       // el auto está "en marcha"
    }
}
```

En el código, definimos la clase auto, con sus atributos marca, modelo, año etc y un método Auto() que invoca a otro método que lo hacer arrancar(). Por ahora, este detalle no es tan real como debería y no estamos haciendo un código funcional, solamente el ejemplo muestra como sería la forma de la escritura de la clase.

Tenemos la clase y el objeto ¿dónde está? En el ejemplo anterior no generamos un nuevo objeto y por ende el objeto no está codificado, eso lo veremos en las siguientes páginas.

### **Diseño de clases**

Como te contamos antes, una clase es un modelo, un esqueleto, una estructura para generar objetos. La clase está compuesta por atributos (datos, variables), también llamadas variables de instancia, que nos indican en qué estado se encuentra cada objeto, y métodos (funcionalidades) que indican cuales comportamientos posee el objeto.

Las clases habitualmente se denotan con nombres generales y globales como Animal, Árbol, Socio, Barco etc.

Toda clase está compuesta por atributos y métodos. Los atributos son características de los objetos. Cuando defines un atributo tendrás que especificar su nombre y su tipo. En definitiva, la definición de un atributo no es más que la declaración de una variable.

## Atributos y Métodos

### Atributos

Cuando definimos atributos dentro de una clase, éstos toman el nombre técnico de atributos de instancia, porque cuando se crea un objeto en memoria de la computadora, éste objeto tendrá una copia tanto de los atributos como también de sus métodos de la clase inicial.

Repasemos:

En JAVA, un atributo o propiedad de una clase, se define igual que una variable y para ello necesita del tipo de dato que tendrá, por ejemplo:

EJEMPLO Código JAVA:

```
String nombre;  
String apellido;  
int    edad;
```

estos atributos se crearán dentro de una clase a la cual pertenecerán y al momento de crear una instancia de esta clase (llamarla para interactuar con ella), generará un objeto con los mismos atributos. Para ello los atributos se inicializan dentro de la clase:

EJEMPLO Código JAVA:

```
Personas Medicos = new Personas(); // instancia de la clase Personas para crear al objeto Medicos
```

```
public class personas {           // Clase personas que será instanciada para crear un objeto.  
String nombre = "Juan";           // Atributos nombre, apellido, edad que pasarán a ser parte  
String apellido = "Perez";        // del objeto  
int    edad = 23;  
}
```

¿No entiendo, y donde se escribe una clase? ¿Cuándo tengo que escribirla y como se relaciona con el objeto?

La clase que vimos en la página anterior, es un ejemplo sencillo y sin ninguna funcionalidad real, de cómo se escribe una clase Auto.

Como ese código representa a la clase Auto en lenguaje JAVA, la escritura debe realizarse en NetBeans y para ello debemos generar previamente un proyecto.

Te invitamos a ver el siguiente video que muestra paso a paso, como crear el proyecto, la clase y el objeto de esa clase para que puedas ir incorporando el concepto práctico de la escritura en la POO. El video tiene un lenguaje sencillo y algunos términos, no son los que estrictamente deban ser utilizados. Hemos dado mayor importancia a que se entienda lo explicado que a la técnica teórica.

## Comportamiento (métodos)

Con los métodos estamos definiendo el comportamiento que tendrá y/o podrá realizar un objeto.

En nuestro ejemplo siguiente, los métodos son 2: MiNombre y MiEdad

Cuando se crea un objeto (tomando como base la clase a la cual pertenece) se genera una copia de los métodos de la clase, en cada objeto creado. A esto se lo denomina método de instancia.

Es importante que sepas que junto con el nombre del método, la declaración lleva información del tipo de devolución de los datos que posee del método, el número y el tipo de los parámetros necesarios, y qué otras clases y objetos pueden llamar al método.

Los métodos pueden poseer argumentos (parámetros) o no. En caso de no tenerlos solo se escriben los paréntesis vacíos, en caso de tenerlos se define el conjunto de argumentos de cualquier método en una lista de declaraciones de variables delimitadas por comas donde cada declaración de variable se realiza indicando el tipo de dato y el nombre, por ejemplo: *String Nombre*.

EJEMPLO Código JAVA:

Veamos entonces como escribir la creación del objeto “Medicos” mediante la instancia a la clase “personas”:

### Archivo medicos.java

```
public class medicos {  
    public static void main(String[] args) {  
        personas Medicos = new personas();           // creo el objeto Medicos en base a personas  
        Medicos.MiNombre();                          // Llamo al método MiNombre en Medicos  
        Medicos.MiEdad();                            // Llamo al método MiEdad en Medicos  
    }  
}
```

Como podrás notar, los métodos los llamamos desde Medicos con la escritura de Medicos.MiNombre() o Medicos.MiEdad() ya que como anteriormente creamos el objetos Medicos con todos los atributos y métodos de la clase personas.

Ahora veremos como está conformada la clase “personas” para albergar los atributos (Nombre, Apellido y Edad) y los métodos MiNombre y MiEdad que emiten por pantalla los datos de los atributos.

EJEMPLO Código JAVA:

### Archivo personas.java

```
public class personas {
```

```

String Nombre = "Juan";           // creamos los atributos de la clase
String Apellido = "Perez";        // en este caso Nombre, Apellido y Edad
int edad = 48;

public void MiNombre() {        // método que muestra por pantalla el Nombre

    System.out.println("Mi nombre es: " + Apellido + ", " + Nombre);

}

public void MiEdad(){          // método que muestra por pantalla la Edad

    System.out.println("Y tengo " + edad + " años");

}
}

```

Para ver más sobre creación de métodos te invitamos a ver el siguiente video que muestra paso a paso, como crear el proyecto, la clase y el objeto de esa clase para que puedas ir incorporando el concepto práctico de la escritura en la POO. El video tiene un lenguaje sencillo y algunos términos, no son los que estrictamente deban ser utilizados. Hemos dado mayor importancia a que se entienda lo explicado que a la técnica teórica.

### ¿Que es un método?

Un método es el código definido para realizar una acción que se incluye dentro de una clase y puede ser parte de un Objeto cuando ese Objeto haya sido creado mediante la instancia a la clase, en nuestro ejemplo veíamos esta instancia y creación del objeto en la línea de código:

EJEMPLO Código JAVA:

```
personas Medicos = new personas();
```

en donde el objeto Medicos se crea a partir de la instancia de la clase personas.

### Parámetros, Argumentos y Valor de Retorno en Métodos.


Aunque es bastante habitual encontrar confusiones con respecto a los conceptos de parámetro y argumento, hay que significar que son, conceptualmente hablando, los opuestos en la tarea del traslado de la información de un método a otro.

EJEMPLO Código JAVA:

En el caso del método que calcula los números primos, sería :



```
public static boolean primo(int n){  
    for (i = n; i>1; i--){  
        if ( n%i == 0)  
            return false;  
    }  
    return true;  
}
```

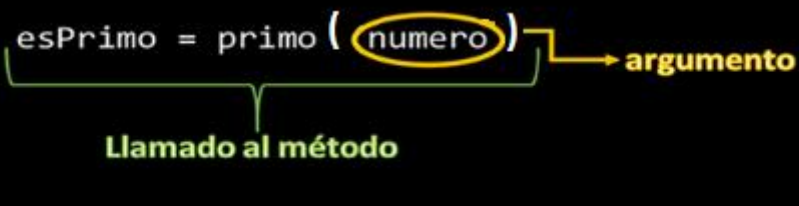


Podemos definir entonces que el **parámetro** es el **valor** que **recibe** un **método** para su procesamiento.

En cambio, un argumento:

EJEMPLO Código JAVA:

```
public static void main(String[] args){  
    int numero;  
    boolean esPrimo;  
    // Se lee el numero del usuario  
  
    esPrimo = primo ( numero )  
}  
  
Llamado al método
```



Sería el “opuesto” a la recepción de un dato como es el parámetro, ya que el **argumento** es el **valor** que se **envía** a un **método** para ser procesado.

En muchos casos, un objeto se “comunica” con sus métodos o con los métodos de otros objetos mediante el envío de argumentos que el método recibe como parámetro, esta comunicación que es “unilateral”, ya que se envía un dato que es recibido por un método, tiene también su opción “bidireccional”, cuando el método puede retornar valores y modificar el estado (valor de sus atributos) de otro objeto o simplemente de una variable.

Veamos un ejemplo, en donde

- 1 crearemos un objeto calculadora,
- 2 en base a una clase calculo,
- 3 que tendrá un método que suma dos valores
- 4 que le enviaremos como argumentos
- 5 y recibirá como parámetros.
- 6 El resultado, nos llegará como un retorno de valor del método
- 7 y los guardaremos en una variable.

EJEMPLO Código JAVA:

#### Archivo sumas.java

```
public static void main(String[] args) {  
    calculo Calculadora = new calculo();           //1. Crearemos un objeto calculadora,  
                                                    //2,en base a una clase calculo,  
  
    double resultado;                             //creamos la variable para guardar el resultado  
  
    resultado = Calculadora.sumar(50.38 , 80.56); //3. método que suma dos valores  
  
    System.out.println(resultado);                // mostramos el resultado  
}
```

EJEMPLO Código JAVA:

#### Archivo calculo.java

```
public class calculo {                               // clase calculo,  
  
    public double sumar(double valor1,double valor2) { //3. un método que suma dos valores  
                                                    //y recibe como parámetros.  
  
        return(valor1+valor2);                    //6. retorno de valor del  
método  
                                                    }  
    }  
}
```

Repasemos el código, nos encontramos primero con la creación de un objeto de la clase calculo que llama al método sumar pasándole como argumentos los valores a sumar, en este caso los números a sumar son: 50.38 y 80.56 . El separador decimal es el punto.

Como podrás observar los argumentos se separan con una coma en el caso que sean varios.

Por otro lado, nos encontramos en la clase con el método sumar que recibe ambos parámetros, define con double después del public, que retornará un valor del tipo doble.:

```
public double sumar(double valor1,double valor2){
```

y entre los paréntesis, indica que recibirá dos parámetros, también del tipo double con los nombres valor1 y valor2, los que serán procesados y devueltos al objeto que los llamó con la orden:

```
return(valor1+valor2);
```

Y como obtengo el valor de la suma?

En la primer porción de código, cuando se llama al método sumar con:

```
resultado = Calculadora.sumar(50.38 , 80.56);
```

se está solicitando que el valor retornado por el cálculo sea guardado en la variable resultado.

Cuando un método no enviará ninguna respuesta, debe contener la palabra void indicando esta situación:

EJEMPLO Código JAVA:

```
public void MiNombre() {
```

```
        System.out.println("Mi nombre es: " + Apellido + ", " + Nombre);  
    }
```

## **Identidad de un Objeto**

### **Identidad**

La identificación o identidad es la propiedad que permite diferenciar a un objeto y distinguirse de otros. Generalmente esta propiedad es tal, que da nombre al objeto. Tomemos por ejemplo el "verde" como un objeto concreto de una clase color; la propiedad que da identidad única a este objeto es precisamente su "color" verde. Tanto es así que para nosotros no tiene sentido usar otro nombre para el objeto que no sea el valor de la propiedad que lo identifica.

EJEMPLO Código JAVA:

```
color verde = new color();
```

En programación la identidad de todos los objetos sirve para comparar si dos objetos son iguales o no.

La identidad tiene su fundamento en el soporte que da un tipo de clase a su utilización para la creación de varios objetos, con lo cual, así como hemos creado el objeto verde, utilizando la misma clase "color" podemos crear otro objeto con la identidad "rojo", y aunque ambos objetos tendrán inicialmente los mismos atributos y métodos, la identidad es lo que separa los datos y ejecución de métodos en memoria:

EJEMPLO Código JAVA:

```
color rojo = new color();
```

## **Constructores**

Un Constructor es un método de las clases, el cual es llamado automáticamente cuando se crea un objeto a partir de esa clase.

Por ser métodos, los constructores también aceptan parámetros. Cuando en una clase no especificamos ningún tipo de constructor, el compilador añade uno de uso público por omisión sin parámetros, el cual NO hace nada.

EJEMPLO Código JAVA:

```
public class Auto {  
    String marca = "Ford";  
    String modelo = "Falcón";  
}
```

```
int año = 1975;
```

```
int puertas = 4;
```

```
public Auto() {      // Podemos ver en esta línea la creación del constructor  
                  // de la clase Auto.
```

```
}
```

```
}
```

## Características de los Constructores

- 1 Un constructor, tiene el mismo nombre de la clase a la cual pertenece.
- 2 No retorna ningún valor, por lo cual no debe especificarse ningún tipo de dato.
- 3 Debe declararse como public, aunque en algunos casos puede ser de otro tipo.
- 4 Si la clase tiene algún constructor, el constructor por defecto deja de existir. En ese caso, si queremos que haya un constructor sin parámetros tendremos que declararlo explícitamente.

Su función es inicializar el objeto y sirve para asegurarnos que los objetos siempre contengan valores válidos.

## Uso del This

Hay ocasiones en las que resulta útil asegurarnos que nos estamos refiriendo al objeto desde el que se está ejecutando un método.

Para ello se implementó en JAVA el uso de la palabra reservada “this”, que en estas ocasiones se puede usar la referencia especial desde el objeto actual.

Esta referencia se suele usar para pasar una referencia al objeto actual como un parámetro que fue recibido para nuestros métodos.

Es así como podemos realizar una nueva versión de la clase calculo, que utilizamos en la sección de parámetros y argumentos:

EJEMPLO Código JAVA:

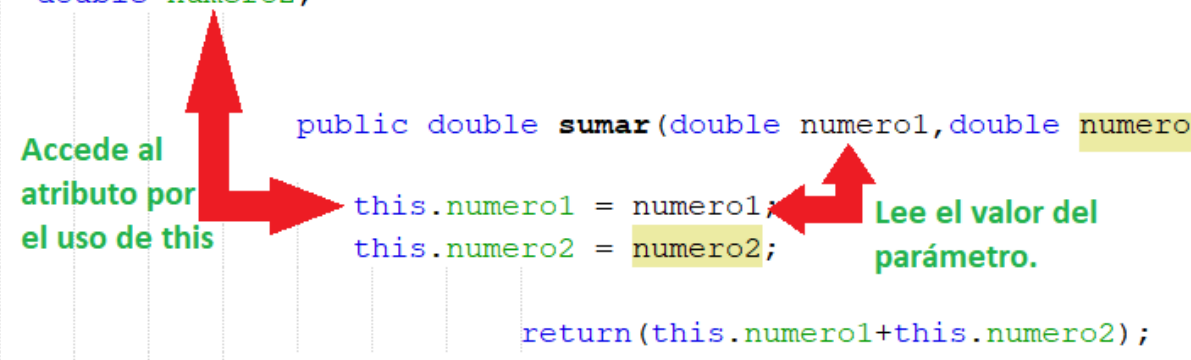
```
public class calculo {  
    double numero1;  
    double numero2;  
    public double sumar(double numero1,double numero2) {  
        this.numero1 = numero1;    // uso de this refrenciando a numero1  
        this.numero2 = numero2;    // uso de this refrenciando a numero1  
        return(this.numero1+this.numero2); // respuesta con this.  
    }  
}
```

Como podemos notar en el código anterior se crearon dos nuevos atributos (numero1 y numero2) que se utilizan dentro del método sumar para hacer una copia del valor que traen los parámetros al método y después con ese mismo this retornamos el valor de la suma.

Sin la utilización de este modificador this, el acceso a los atributos del método actual puede generar inconvenientes si tenemos asignados el mismo nombre a un parámetro del método.

En el ejemplo, tanto los parámetros como los atributos se llaman igual, se diferencian en el método por el uso del this, que obliga al método a tomar el valor creado por él y no el que proviene de los parámetros:

```
public class calculo {  
    double numero1;  
    double numero2;  
  
    public double sumar(double numero1, double numero2) {  
        this.numero1 = numero1;  
        this.numero2 = numero2;  
        return (this.numero1+this.numero2);  
    }  
}
```



Accede al atributo por el uso de this

Lee el valor del parámetro.

Netbeans nos ayuda a leer el código, asignando el mismo color a **verde** a los atributos que se referencian con el this y el color **negro** a los parámetros que provienen del método.

## Creación de Objetos

En el ejemplo anterior de métodos, realizábamos la creación de un objeto con la palabra clave New, la sentencia era:

EJEMPLO Código JAVA:

```
personas Medicos = new personas();
```

Podemos leer la sentencia del ejemplo como:

*“Voy a crear un objeto Medicos del tipo o de la clase personas instanciando su método constructor personas()”*

Esta creación de un objeto, se logra instanciando la clase personas con la llamada a su constructor en conjunto con new, lo que hace que la orden de creación sea **new personas()**.

Medicos está representando una nueva identidad para el objeto, o sea, le asignamos un nombre y también indicamos que será del tipo “personas” con la sentencia personas Medicos.

Cuando se crea un objeto de la clase Personas, su variable de instancia llamada Medicos se inicializa de manera predeterminada con null, si es que la clase ya no tiene asignación de datos a los atributos.

Pero ¿qué pasa si queremos proporcionar un nombre a la hora de crear un objeto Medicos?

En cada clase que declaramos, se puede proporcionar de manera opcional un constructor con parámetros que pueden utilizarse para inicializar un objeto de una clase al momento de crear ese objeto.

El siguiente ejemplo mejora la clase Persona con un constructor que puede recibir un nombre y usarlo para inicializar las variables de instancia nombre, apellido y edad al momento de crear un objeto Medico:

EJEMPLO Código JAVA:

Veamos entonces como escribir la creación del objeto “Medicos” mediante la instancia a la clase “personas” con un constructor que recibe tres parámetros:

#### **Archivo medicos.java**

```
public class medicos {  
    public static void main(String[] args) {  
        personas Medicos = new personas("Juan", "Perez", 38); // Clase con argumentos.  
        Medicos.MiNombre();  
        Medicos.MiEdad();  
    }  
}
```

EJEMPLO Código JAVA:

#### **Archivo personas.java**

```
public class personas {  
    String Nombre;  
    String Apellido;  
    int edad;  
  
    public personas(String nombre, String apellido, int edad){ // Constructor con parámetros  
        this.Nombre = nombre;  
        this.Apellido = apellido;  
        this.edad = edad;  
    }  
  
    public void MiNombre() { // Método que muestra el nombre por pantalla  
        System.out.println("Mi nombre es: " + Apellido + ", " + Nombre);  
    }  
}
```

```
public void MiEdad(){           // Método que muestra la edad por pantalla  
    System.out.println("Y tengo " + edad + " años");  
}  
}
```

Java requiere una llamada al constructor para cada objeto que se crea, por lo que éste es el punto ideal para inicializar las variables de instancia de un objeto.

Para reforzar y ampliar los conceptos de Objetos y del paradigma de programación orientado a objetos te recomendamos que veas el video donde el profesor Jorge Fumarola, docente del plan codo a codo, explica estos temas en el siguiente link: <https://youtu.be/dFOLxyRRiq4>

## ENCAPSULAMIENTO

---

En un lenguaje orientado a objetos como JAVA, hablamos de encapsular al referirnos al acceso que brinda una clase a diferentes objetos sobre sus datos.

Pensemos el hecho de encapsular como guardar u ocultar información para que ésta no sea modificada o tenga un tratamiento privado.

Hemos visto en las unidades anteriores que muchas de las clases, métodos y propiedades se definen con la palabra “public” delante de la sentencia de su creación y con eso determinábamos que el acceso de esa información sería público para cualquier otra clase que desee obtener información y/o comunicarse con ella.

El encapsulamiento, nace y se utiliza exactamente para lo contrario. Hacemos que la información que antes era pública, ahora sea privada.

Encapsulamos información (métodos, atributos etc.) para que no pueda ser accedida desde otros métodos y solamente pueda ser utilizada dentro del método que creó esa información

Cuando elegimos definir que la información no será compartida, la llamaremos privada y se define con “private” en JAVA.

El encapsulamiento de variables y métodos en un programa es una simple y poderosa herramienta que provee dos principales beneficios a los desarrolladores de software:

- ❑ **Modularidad**, esto es, el código de una clase puede ser escrito y modificado, independientemente del código de otras clases. Así mismo, un objeto puede ser utilizado en un sistema sin alterar su estado y conducta.

- ❑ **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 (si éste lo permite) en cualquier momento sin afectar a los otros objetos que dependan de ello.

### **Modificadores de acceso: public y private.**

Así como en el apartado anterior mencionamos cual es el objetivo de encapsular, tendremos que tener en cuenta las herramientas necesarias para un correcto encapsulamiento, o no, de la información.

Como primer contacto con encapsulamiento veremos el siguiente código:

### **EJEMPLO Código JAVA:**

Supongamos que estamos definiendo la clase personal, y por motivos de requerimiento de la empresa, no debe compartirse con otros módulos la información del sueldo abonado a cada empleado, aunque



sí compartiremos datos (edad, apellido y nombre). Para situaciones como esta podremos encapsular el atributo a mantener privado:

```
public class personal{                                //declaración de la clase personal
    //Atributos
    private double sueldo = 3000.00; //declaro como privado el sueldo
    public int edad = 25;                // declaro como público la edad
    public String apellido = "Perez";    // declaro como público el Apellido
    public String nombre = "José";      // declaro como público el nombre
}
```

En este sencillo ejemplo, podemos ver que aunque la clase personal se definió como pública, uno de sus atributos (el sueldo), fue encapsulado para proteger la información, los demás atributos son definidos como públicos y pueden ser accedidos desde otras clases.

SIGUIENTE→

### Conceptos de Getters y Setters.

En el apartado anterior, definimos datos de acceso público y privado, unos comparten la información y otros no. Ahora, y ¿que sucede si deseamos cambiar el sueldo de una persona?. El atributo es privado y normalmente tendríamos denegado el acceso a él. Para poder acceder es que existen los métodos normalmente llamados Getters y Setters, aunque se utilizan con las palabras Get y Set seguida del nombre del atributo.

Antes de profundizar sobre los Getters y Setters debemos recordar todo lo referente a la clase personal(), es decir sabemos que al crear la clase definimos una parte pública y una parte privada (el sueldo), por lo que estos métodos Get y Set, serán métodos de acceso, lo que significa que generalmente son una forma pública para cambiar miembros de las clases privadas.

Los métodos getter y setter se utilizan para definir una propiedad (el valor) de un atributo, y a éstos atributos se accede como propiedades situadas fuera de la clase, aunque las defina dentro de la clase como métodos.

Veamos un ejemplo.

EJEMPLO Código JAVA:

Continuaremos con la clase personal(). En este caso, aunque el atributo sueldo continúe siendo privado, habilitaremos un método Set para enviarle un nuevo valor y cambiar el sueldo y un método Get para obtener en cualquier momento cuanto es el sueldo actual de esa persona, para ello el código sería el siguiente:

```

public class personal{                                //declaración de la clase personal
    //Atributos
    private double sueldo = 3000.00;    //declaro como privado el sueldo
    public int edad = 25;                // declaro como público la edad
    public String apellido = "Perez";    // declaro como público el Apellido
    public String nombre = "José";       // declaro como público el nombre
    public double getSueldo() {          // Incorporo el metodo getSueldo
        return sueldo;                  // el método nos devuelve el valor sueldo
    }
    public void setSueldo(double nuevosueldo) {      // Incorporo el método setSueldo
        this.sueldo = nuevosueldo;                  // el método le asigna un valor nuevo a sueldo
    }
}

```

En este ejemplo, podemos ver como aplicamos un get y un set. El setSueldo recibe como parámetro un valor (nuevosueldo) que le enviaremos desde otra clase, para asignar una nueva propiedad (en este caso importe) al sueldo de la persona. El método getSueldo, nos devuelve cual es el valor actual del atributo sueldo, independientemente de que haya sido modificado previamente o no.

Ya dicho esto podemos decir que los Setters & Getters nos sirven para dos cosas:

- **Setters:** Del Inglés **Set**, que significa establecer, pues nos sirve para asignar un valor a un atributo, pero de forma explícita, y solo nos permite dar acceso público a los atributos que deseamos que el usuario pueda modificar. Es decir permiten cambiar el valor de los atributos.
- **Getters:** Del Inglés **Get**, que significa obtener, pues nos sirve para obtener (recuperar o acceder) el valor ya asignado a un atributo y utilizarlo para cierto método. Por lo que devuelven el valor o propiedad de los atributos por él alcanzados.

¿Y si un atributo fue creado como privado, porque no se deseaba que se acceda a él, para que crear métodos set y get que pueden modificarlo.?

En muchas ocasiones, el uso del Set y Get no apunta a modificar la propiedad de un atributo privado, sino que además de esa opción tenemos la posibilidad de ejecutar un filtro o control del dato que llega al método Set, y eso no lo podríamos solucionar en un atributo que sea público solamente.

## Miembros Estáticos.

La definición formal de los elementos estáticos (o miembros de clase) nos dice que son aquellos que pertenecen a la clase, en lugar de pertenecer a un objeto en particular. Recuperando conceptos básicos de orientación a objetos, sabemos que tenemos:

**Clases:** definiciones de elementos de un tipo homogéneo.

**Objetos:** concreción de un ejemplar de una clase.

En las clases defines que tal objeto tendrá algunos atributos y métodos, sin embargo, para acceder a ellos o darles valores necesitarás construir objetos de esa clase. Una casa tendrá un número de puertas para entrar, en la clase tendrás definida que una de las características de la casa es el número de puertas, pero solo concretarás ese número cuando construyas objetos de la clase casa.

En la clase cuadrado definirás que el cálculo del área es el "lado elevado a dos", pero para calcular el área de un cuadrado necesitas tener un objeto de esa clase y pedirle que te devuelva su área.

Ese es el comportamiento normal de los miembros de clase. Sin embargo, los elementos estáticos o miembros de clase son un poco distintos. Son elementos que existen dentro de la propia clase y para acceder los cuales no necesitamos haber creado ningún objeto de esa clase. Osea, en vez de acceder a través de un objeto, accedemos a través del nombre de la clase.

## EJEMPLO Código JAVA:

¿Como definir entonces un atributo static?

Se define anteponiendo la palabra "static" en la creación del atributo.

En las prácticas veremos como utilizar este tipo de modificador en ejemplos sencillos.

```
static String apellido = "Perez";    // atributo static que puede
                                     // incorporarse en una clase
```

## Atributos y métodos de clase

### Atributos de clase

Hasta el momento, cuando hemos definido atributos dentro de la clase, hemos dicho que se denominan atributos de instancia, porque cuando se crea un objeto en memoria ese objeto tendrá una copia tanto de los atributos como también de sus métodos.

Existe un modificador que aplicado a las variables hace que el atributo de instancia se convierta en atributo de clase. La diferencia que existe con el anterior es que al momento de crear un objeto ese atributo de clase no se crea en el objeto, por lo que si se llegaran a crear más objetos ninguno de ellos tendría una copia del original sino que todos compartirían el mismo atributo. Esto implica que si un objeto modifica el valor de ese atributo, se verá reflejado el cambio en todos los objetos.

Para poder definir un atributo de clase tendrás que utilizar la palabra `static`, anteponiéndola al nombre del atributo.

## Métodos de clase

Si aplicás la palabra `static` a los métodos, se los denomina método de clase. Los métodos de clase permiten acceder a código cuando no se tiene una instancia en particular de un objeto.

Al definir un método de clase, tenés que tener en cuenta que no pueden acceder a atributos que no están definidos como estáticos.

## El método toString()

El método **toString** nos permite mostrar la información completa de un objeto, es decir, el valor de sus **atributos**.

### EJEMPLO Código JAVA:

Este es un ejemplo de su definición:

```
public String toString (){  
    String mensaje="El empleado se llama "+nombre+" "+apellido+" con "+edad+" años " +  
        "y un salario de "+sueldo;  
    return mensaje;  
}
```

**El mensaje puede tener el formato que nos parezca conveniente según el caso.**

Continuando con el ejemplo de la clase `personal()`, incorporaremos el método `toString()` de la siguiente manera:

### EJEMPLO Código JAVA:

```
public class personal{                                //declaración de la clase personal
    //Atributos
    private double sueldo = 3000.00;                //declaro como privado el sueldo
    public int edad = 25;                            // declaro como público la edad
    public String apellido = "Perez";                // declaro como público el Apellido
    public String nombre = "José";                  // declaro como público el nombre
    public double getSueldo() {                      // Incorporo el metodo getSueldo
```

```

        return sueldo;                // el método nos devuelve el valor sueldo
    }

    public void setSueldo(double nuevosueldo) {    // Incorporo el método setSueldo
        this.sueldo = nuevosueldo;                // el método le asigna un valor nuevo a sueldo
    }

    public String toString (){
        String mensaje="El empleado se llama "+nombre+" "+apellido+" con "+edad+" años " +
            "y un salario de "+sueldo;
        return mensaje;
    }
}

```

Como podemos ver en el código de ejemplo, la implementación de un método toString es sencilla y se utiliza normalmente con la idea de recibir un valor o mensaje que proviene por parte de este método.

Para que este método sea funcional, primero tenemos que crear el objeto que instancia a la clase personal y posteriormente solicitar el mensaje, para ello en Netbeans, crearemos un proyecto y en el método principal (main) llamaremos el objeto:

```

public static void main(String[] args) {
    personal administrativo = new personal(); //Instanciamos personal para los administrativos
    administrativo.setSueldo(5000);           // asignamos $ 5000 como sueldo de un adm.
    System.out.println(administrativo);       // llamamos a toString solamente indicando el
                                              // nombre del objeto
}

```

El resultado por pantalla será:

El empleado se llama José Perez con 25 años y un salario de 5000.-

### **Estado de un Objeto.**

El estado de un objeto abarca todos los atributos del objeto, y los valores actuales de cada una de esas propiedades. Las propiedades o atributos de los objetos suelen ser estáticas, mientras los valores que toman estas propiedades o atributos cambian con el tiempo.

Continuando el ejemplo anterior, un objeto de Personas tiene un estado cuando:

```

String nombre = "Juan";
String apellido = "Perez";

```

y tiene otro estado diferente cuando:

```
String nombre = "Roberto";
```

```
String apellido = "Gomez";
```

ya que el valor de los atributos y la situación de sus métodos definen en que estado se encuentra el objeto Personas.

Entre las situaciones a analizar para reconocer el estado de un objeto, generalmente se interpreta que:

- El hecho de que los objetos tengan un estado implica que ocupan un espacio determinado en la memoria de la computadora. Están en funcionamiento.
- El estado de un objeto está influido por la historia del objeto. En nuestro caso, comenzó con “Juan”, y esa historia nos lleva a un estado último definido por “Perez”.
- No deben confundirse los objetos, que pueden cambiar, que si tienen un estado, que pueden ser creados, destruidos y compartidos..., solamente con los valores (los asignados a un atributo, por ejemplo) que en definitiva determinan un dato. El estado puede definirse, además de por el valor de un atributo, por un método o por las posibilidades de interactuar de una clase con otras.
- El estado de un objeto representa, en todos los casos, el efecto acumulado de los cambios, modificaciones, ejecuciones, intervenciones y todas las acciones que se hayan llevado a cabo sobre él o sobre algún otro objeto con el cual interactúa.

## ¿Qué son las relaciones de clases?

Las relaciones entre las distintas clases nos indican cómo se comunican los objetos de esas clases entre sí: Los mensajes “navegan” entre las clases que se encuentran, por ende, relacionadas. La conexión entre objetos será un detalle de como están asociados.

El **alcance o tipo de acceso** es una característica fundamental de la POO ya que te permitirá definir cual es la forma de acceso a la información al comportamiento de un objeto. Los métodos, atributos y clases deben tener un solo alcance.

Cuando diseñes clases, es importante que pienses en términos de **quién debe tener acceso a qué**. Qué cosas son parte privada y deberían ocultarse (y en qué grado) y qué cosas forman parte de la información pública.

Podrás definir alguno de los siguientes **cuatro tipos de acceso**:

---

### ALCANCE

---

Privado

Público

Protegido

Paquete

#### Privado

Si un atributo o método tiene alcance de tipo privado, este solo podrá ser accedido por los métodos definidos en la misma clase. Es importante que sepas que las clases no pueden declararse como privadas.

#### Público

Si un atributo o método tiene un acceso de tipo público es accedido desde cualquier clase, no tiene restricciones de ningún tipo (no oculta la información). Las clases solo pueden tomar dos tipos de modificadores público o de paquete (lo vemos más abajo). Las clases no pueden declararse ni Protegido, ni Privado.

#### Protegido

El concepto de herencia lo veremos en una de las próximas unidades de POO, así que volveremos a mencionar este acceso cuando te expliquemos qué es la herencia.

Este alcance es muy importante cuando se utilizan relaciones de herencia entre las clases, y significa que los atributos y métodos serán accesibles desde las clases heredadas.

#### Paquete

Las clases tienen un comportamiento definido para quienes las usan, conformado por los elementos que tienen un acceso público, y una implementación oculta formada por los elementos privados, de la que no tienen que preocuparse los usuarios

Un Paquete en Java es un conjunto de clases que agrupa las distintas partes de un programa. Por lo general, las clases agrupadas, tienen una funcionalidades y elementos comunes, definiendo la ubicación de dichas clases en una estructura jerárquica.

Con respecto a los métodos, por lo general, aquellos que definas dentro de una clase tendrán un especificador de acceso público. En el caso de querer definir un método y que este no forme parte del grupo o paquete que permite ser accedido desde otros métodos, lo que tendrás que hacer es declararlo como privado. Este método privado puede ser invocado dentro de cualquier otro método de la misma clase, no desde otras.

Ahora que ya te contamos el tipo de alcance que podrás aplicar a cada atributo o método, es momento de modificar la clase PERSONAL para prestar mayor atención en como están declaradas las forma de acceso a sus métodos y atributos.

**Veamos un ejemplo.**

**EJEMPLO Código JAVA:**

```
public class Personal{                                //declaración de la clase Personal  
  
    //Atributos  
  
    private double sueldo = 3000.00; //declaro como privado el sueldo  
    public int edad = 25;           // declaro como público la edad  
    public String apellido = "Perez"; // declaro como público el Apellido  
    public String nombre = "José";  // declaro como público el nombre  
    public double getSueldo() {     // Incorporo el metodo getSueldo  
        return sueldo;            // el método nos devuelve el valor sueldo  
    }
```



## ¿Qué es la herencia?

Seguramente, al haber leído la palabra herencia, te habrá venido a la mente lo que sucede con una persona que puede heredar algunos rasgos físicos o Personales de sus padres o abuelos. La herencia es la característica por la cual los padres transmiten algunas características a sus hijos. Del mismo modo que en biología, en el plano económico, una persona le puede heredar sus bienes a otra persona, es decir que se los puede traspasar a otra persona por medio de una relación de jerarquía.

En resumen podríamos expresar que :

La herencia es una propiedad de los objetos, que le permite obtener atributos y métodos de una clase y así ampliar su funcionalidad. Entonces, en este mecanismo, encontraremos que hay una clase que aporta los atributos y métodos y otra que los recibe para utilizarlos sin necesidad de escribir el código en Java nuevamente.

Recordemos cómo está definida la clase Personal. En este caso le agregamos el método constructor que vimos en la unidad de POO para que pueda recibir datos.

### EJEMPLO Código JAVA:

```
public class Personal{                                     //declaración de la clase Personal

    //Atributos

    private double sueldo = 3000.00;    //declaro como privado el sueldo
    public int edad = 25;                // declaro como público la edad
    public String apellido = "Perez";    // declaro como público el Apellido
    public String nombre = "José";       // declaro como público el nombre

    // Método constructor para Personal, que indica los parámetros necesarios para su /
    //funcionamiento

    public Personal(String nombre, String apellido, int edad, double sueldo){

    }

    public double getSueldo() {           // Incorporo el metodo getSueldo
        return sueldo;                   // el método nos devuelve el valor sueldo
    }
}
```

## ¿Qué es la herencia?

Para analizar como programar la herencia, supongamos que de la clase Personal que está detallada anteriormente, tendríamos que registrar a los Medicos que trabajan en un hospital. Ellos sería parte del Personal que trabajan en esa institución, aunque hay algunos otros datos que incorporar como por ejemplo: la matrícula y la especialidad.

Otros datos, ya están referenciados como atributos de la clase, ya que el médico posee edad, apellido, nombre y un sueldo en el Hospital.

¿Cómo resolvemos esto sin tener que repetir todos los atributos de la clase Personal en una clase para los Medicos? Como verás, la idea es ahorrar tiempo de escritura.

Es en este momento donde deberás hacer uso de la herencia. Es así como nuestra nueva clase quedaría con el siguiente modelo.

EJEMPLO Código JAVA:

```
public class Medicos extends Personal{  
    private String Matricula;  
    private String Especialidad;  
  
    public Medicos(String nombre, String apellido, int edad, double sueldo){  
        super(nombre, apellido, edad, sueldo);  
        Matricula = "35853";  
        Especialidad="Clínica Médica";  
    }  
}
```

Como habrás notado, utilizamos la herencia para transmitir los atributos de una clase a otra.

En el segundo ejemplo, la creación de la clase Medicos, tiene como condimento especial la llamada a la herencia que se expresa en JAVA como: **extends Personal** y de esta manera definimos que la clase Medicos, obtendrá los atributos (extends) de la clase Personal sin necesidad de volver a reescribir el código de la clase Personal dentro de Medicos.

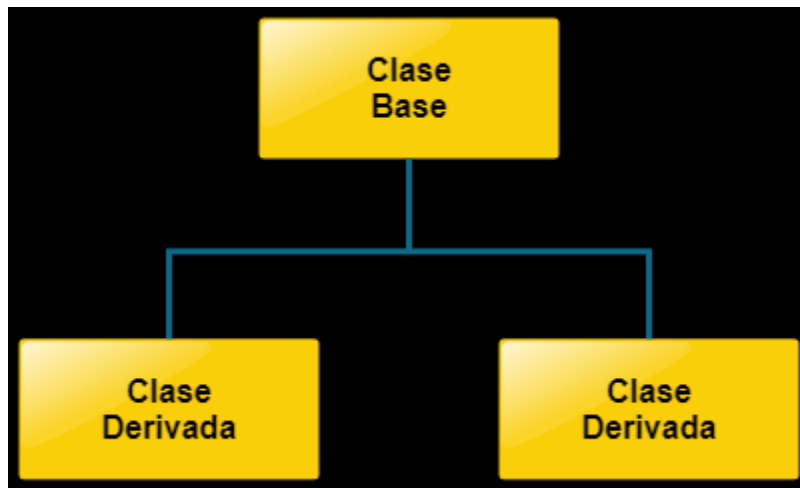
De esta manera podemos decir que:

***La herencia es una propiedad esencial de la Programación Orientada a Objetos que consiste en la creación de nuevas clases a partir de otras ya existentes.***

## Subclases y superclases

En nuestro ejemplo, la clase Personal es una Superclase de la clase Medicos. Y del mismo modo, la clase Medicos es una subclase de la clase Personal.

Un lenguaje orientado a objetos permite heredar a las clases características y conductas, es decir los atributos y métodos, de una o varias clases denominadas superclases, clases bases o padres. A las clases que heredan de otras clases se las denominan subclases, clases derivadas o hijas. Las clases derivadas, a su vez, pueden ser clases bases para otras clases derivadas. De esta manera podrás establecer una clasificación jerárquica similar a la existente en Biología con los animales y plantas.



La herencia te permitirá lograr una de las principales características de la programación, que es la reutilización de código.

Una vez que una clase ha sido probada, el código fuente de dicha clase no necesita modificarse. Su funcionalidad (clase Personal) se puede cambiar derivando una nueva clase que herede (clase Medicos) la funcionalidad de la clase base y le añada otros y/o nuevos comportamientos. De esta manera podrás reutilizar el código existente, ahorrando tiempo y dinero, ya que solamente tendrás que verificar la nueva conducta que proporciona la clase derivada.

Sintetizando...

*La herencia es un mecanismo mediante el cual una clase hereda todo el comportamiento y atributos de otra clase. La clase que hereda se denomina clase hija, clase derivada o subclase. La clase que provee la herencia se llama clase padre, base, o superclase.*

**La herencia está dada por la relación “es un”.**

En la definición de la clase Medicos, se encuentra la palabra reservada extiende (extends) para especificar que es una clase que hereda de la clase Personal.

Tipo de herencia: 1- Herencia Simple

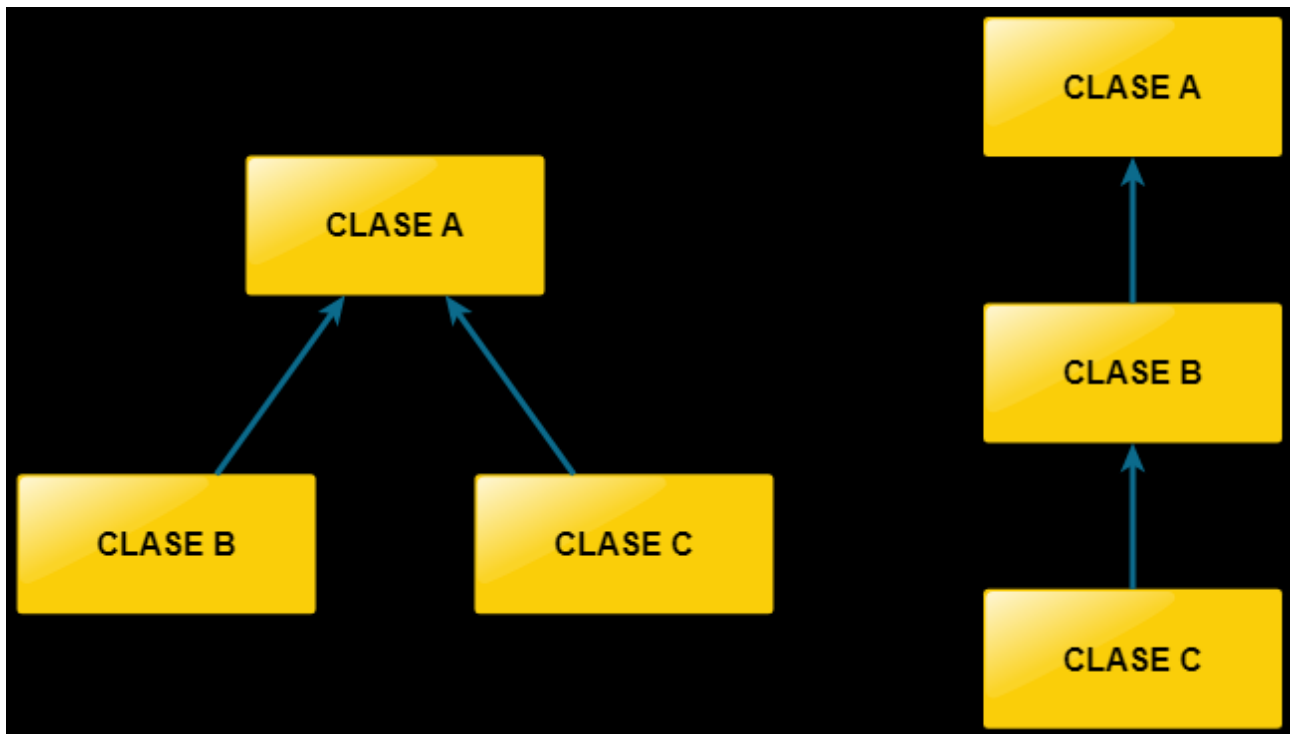
En la **POO existen dos tipos de herencia.**

1- Herencia Simple

Como habrás notado, la clase Medicos de nuestro ejemplo anterior hereda de una sola clase. Este es el caso de una herencia Simple.

En este tipo de herencia, una clase puede extender las características de una sola clase, o sea sólo puede tener un padre.

Ejemplos de herencia simple los podrás ver en la siguiente imagen.



En el **caso de la izquierda**, las clases B y C poseen las mismas características y comportamiento de la clase A.

En cambio, en el **gráfico de la derecha**, la clase B posee las mismas características y comportamiento de la clase A, mientras que la clase C tiene las mismas características y comportamiento de la clase B, pero al haber heredado de A, se dice que C posee las mismas características y comportamiento de la clase A y B.

Tipo de herencia: 2- Herencia Múltiple

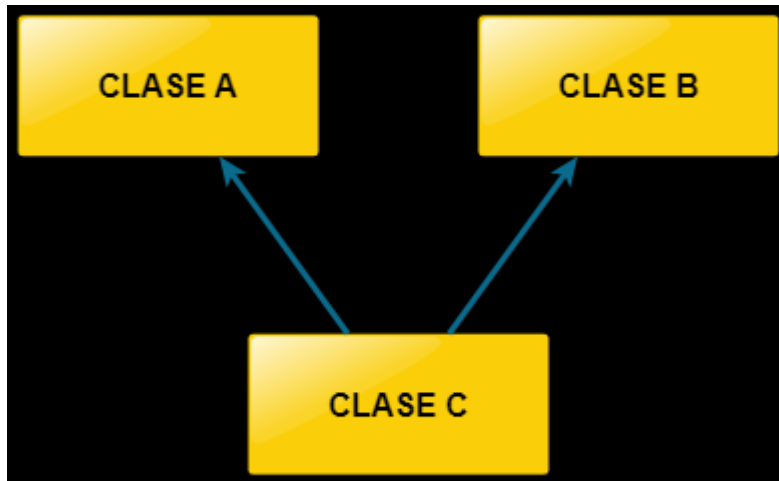
En la **POO existen dos tipos de herencia**.

## 2- Herencia Múltiple

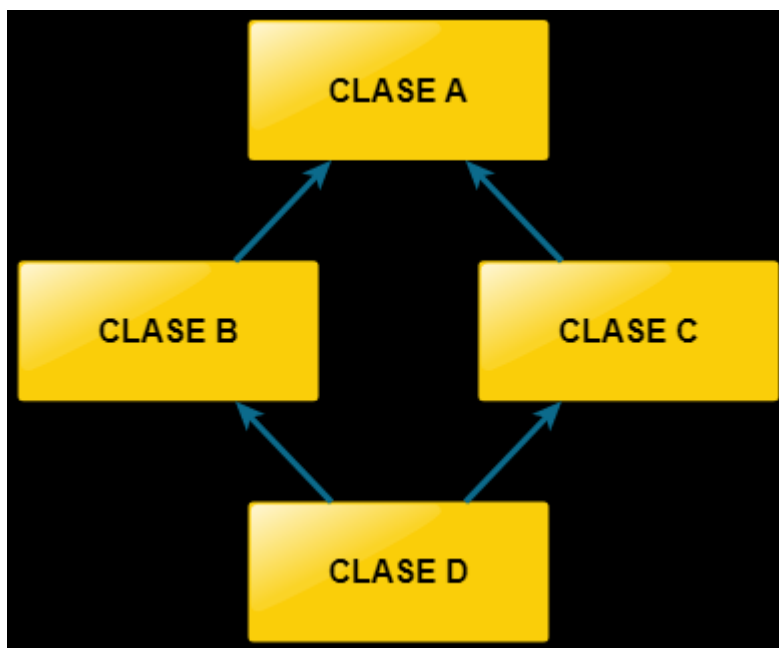
Hay otro tipo de herencia y es en el caso en que una clase puede extender las características de varias clases, es decir, puede tener varios padres. En este aspecto hay discrepancias entre los diseñadores de lenguajes. Algunos de ellos han preferido no admitir la herencia múltiple por las posibles coincidencias en nombres de métodos o datos miembros.

Si una clase cualquiera tiene más de una superclase directa en la jerarquía de clases, se considera que existe herencia múltiple.

En términos concretos, una instancia de objeto de la clase hija poseerá todos los atributos y métodos de sus clases padres.



En este ejemplo, la Clase C posee las mismas características y comportamiento de las Clases A y B. Ante la posibilidad de poder combinar estos tipos de herencia surge el problema del esquema del rombo, que se muestra en la siguiente imagen.



Observando, el problema no se da en lo que se denomina el primer nivel de derivación (herencia entre A, B y C), sino que el problema se encuentra en el segundo nivel de derivación ya que la clase D tendría características de B y C, pero en definitiva tendría una copia doble de la clase A ya que B y C tienen heredadas esas características.

Para el caso de JAVA, los diseñadores del lenguaje prefirieron no aceptar la herencia múltiple como se muestra en el caso anterior, sino que implementaron el concepto de Interfaces para lograr esta cualidad dentro de los objetos. Más adelante detallaremos como trabajar con interfaces.

## Especificadores de alcance

Ahora volvemos a comentarte sobre los especificadores de alcance que te presentamos en unidades anteriores.

En el caso de la herencia, utilizamos los mismos especificadores que se utilizan para definir una clase, método o atributo; la única diferencia es el alcance que tienen en la clase derivada. Esto quiere decir que dependiendo del tipo de acceso que se especifique al componente en la clase base, será cómo se lo pueda utilizar en las clases derivadas.

En la siguiente tabla especificamos el alcance en la clase derivada según el alcance que tenga el componente en la clase base.

	CLASE BASE	CLASE DERIVADA	ACCESO
1	Privado	<b>Privado</b>	Solo mediante métodos
2	Protegido	<b>Público</b>	Directo
3	Público	<b>Público</b>	Directo

1- Si los datos en la clase base (Personal) son privados, en la clase derivada (Medicos) también lo serán y que su acceso sólo será mediante los métodos

2- Si los datos en la clase base (Personal) son protegidos, en la clase derivada (Medicos) se convierten en públicos y que su acceso sólo será directo, o sea, no hará falta acceder mediante la invocación de ningún método.

3- Si los datos en la clase base (Personal) son públicos en la clase derivada (Medicos) también lo serán y que su acceso sólo será directo, o sea, no hará falta acceder mediante la invocación de ningún método.

Como habrás notado, entonces, no es necesario, si vas a tener un esquema de herencia, poner en la clase base los atributos de la misma como privados, sino que los podés poner como atributos protegidos. Aunque siempre hay que analizar cual de los especificadores es de mayor conveniencia para el proyecto.

## Características de los atributos protegidos.

Una es la que nombramos antes: que en la **clase derivada** tienen una visibilidad pública, pero en la **clase base** un atributo protegido es igual que un atributo privado, o sea que no puede ser accedido más que a través de métodos, es decir, que de la información en la clase base no deja de estar oculta.

## Sobrecarga

La sobrecarga es cuando **en una misma clase** se redefine un mismo método. Esta técnica tiene una restricción y es:

- Un método sobrecargado debe tener diferentes parámetros (tipos de dato)

Un ejemplo correcto:

### EJEMPLO Código JAVA:

```
public int multiplicar(int x, int y) {  
    return x * y;  
}
```

```
public int multiplicar(float x, float y) {  
    return x * y;  
}
```

Un ejemplo incorrecto:

```
public int multiplicar(int x, int y) {  
    return x;  
}
```

```
public float multiplicar(int x, int y) {  
    return new Float(String.valueOf(x));  
}
```

A pesar que el tipo de retorno es diferente, los parámetros son iguales, por lo que el compilador lo detectará como el mismo método.

## Composición

Ahora te presentamos **otro tipo de relación** que se puede establecer entre Clases, y para eso te proponemos agregar a nuestro modelo de clases la información del domicilio del Personal del hospital.

Para simplificar el ejemplo, definiremos solamente los métodos para mostrar y para obtener el domicilio.



```
public class Personal{           //declaración de la clase Personal
```



```

        //Atributos

        public double sueldo; //declaro como privado el sueldo
        public int edad;           // declaro como público la edad
        public String apellido;    // declaro como público el Apellido
        public String nombre;      // declaro como público el nombre

//constructor

    public Personal(String nombre, String apellido, int edad, double sueldo){

    }

// Incluir los getters y setters necesarios

// Incluyo la clase domicilio dentro de la clase Personal y genero una subclase para que las
// personas puedan contar con los datos necesarios.

public class domicilio {

    public String Calle;
    public int Piso;
    public String Depto;
    public String Localidad;
    public String Provincia;

// Incluir los getters y setters necesarios

}

}

```

Si observamos el código **tenemos dos clases: Clase Domicilio** que es solo una clase que contiene los atributos propios de un domicilio, y la **Clase Personal** que contiene sus propios atributos como Persona además de un objeto de la Clase Domicilio.

La inclusión de este objeto hace que haya composición, o sea que la Clase Personal está compuesta por la clase Domicilio. A diferencia de la herencia, ahora, si querés acceder a los datos de la Clase Domicilio lo debés hacer mediante el objeto, invocando sus métodos.

### **EJEMPLO Código JAVA:**

Si deseamos instanciar a la clase Medicos e incorporar además datos de su domicilio debemos:

```
// Instanciar a la clase domicilio:
```

```
public static void main(String[] args) {  
    domicilio direccion = null;  
    medicos mm = new medicos("Juan", "Perez", 38583, 3000, direccion);  
    direccion.setCalle("Rivadavia 5248");  
    direccion.setDepto("A");  
}
```

Luego de haber analizado este ejemplo podemos decir que:

*Existe composición cuando una clase está compuesta por un objeto de otra clase. En este caso, el rol de la relación es “tiene un” o “está compuesto por”.*

*En la composición no se habla ni de clase base ni de derivada.*

## Herencias vs. Composición

Seguramente te estarás preguntando:

¿En qué situación utilizar la herencia o la composición?

Te comentamos nuestras conclusiones para ayudarte a tomar esa decisión.

Las **dos técnicas fundamentales para construir nuevas clases a partir de otras ya existentes son la herencia y la composición**. Es importante que entiendas que ambos tipos de relaciones pueden existir al mismo tiempo en un problema en donde hay varias clases y diversos tipos de relaciones.

## Sintetizando...

- La herencia permite construir una clase derivada a partir de una clase base. La clase derivada hereda todas las propiedades de la clase base, es decir, sus atributos (datos), y su comportamiento (métodos). Todas las nuevas características que quieras dar a la clase derivada se las tendrás que brindar por medio de nuevos atributos y métodos.

- La composición de clases es un concepto distinto al de la herencia, ya que expresa el hecho de que se pueden componer o constituir clases nuevas a partir de objetos de otras clases. Esto lo podés notar en el mundo real, en donde podés observar objetos formados por otros objetos: por ejemplo, computadoras formadas por teclado y pantalla; vehículos formados por motor, transmisión y chasis, etc.

En ambas técnicas, el código de las clases originales o clases bases no presenta ningún tipo de modificación.

## La cláusula final

A veces tenemos la necesidad de evitar que una clase sea derivada o bien un método sea implementado en otra clase derivada.

Por ejemplo, pensemos en una clase que define un **setSueldo**. Sería lógico que ninguna otra clase pueda implementarla, por lo tanto, necesitaremos indicar de alguna manera que ese método solamente será definido por la propia clase. Para eso, la POO, tiene la característica de las clases o métodos finales.

Veamos el siguiente ejemplo:

### EJEMPLO Código JAVA:

```
final void aumentarSueldo(double sueldo) {  
    this.sueldo = sueldo*1.30;  
}
```

En este ejemplo, la **clase Medicos, clase derivada de Personal no podría reescribir el método aumentarSueldo**, y por tanto cambiar su comportamiento, ya que este método está definido como final en la clase Personal.

Por lo tanto, solamente será implementado en la misma clase Personal.

Por otro lado, evitamos que se pueda seguir extendiendo el árbol jerárquico, es decir, podríamos hacer que la Clase Medicos no puede derivarse. Esto se debe a que es posible definir la clase Medicos como clase final.

Veamos el siguiente ejemplo:

### EJEMPLO Código JAVA:

```
final class Medicos extends Personal{  
    private String Matricula;  
    private String Especialidad;  
    public Medicos(String nombre, String apellido, int edad, double sueldo, domicilio dom){  
        super(nombre, apellido, edad, sueldo,dom);  
        Matricula = "35853";  
        Especialidad="Clínica Médica";  
    }  
}
```

## ¿Qué es el polimorfismo?

Seguramente, te estarás preguntando qué es este concepto de polimorfismo. Esta palabra está relacionada con la idea de tener muchas formas. Si consultamos el diccionario de la Real Academia Española, nos dirá que una de las acepciones de polimorfismo es:

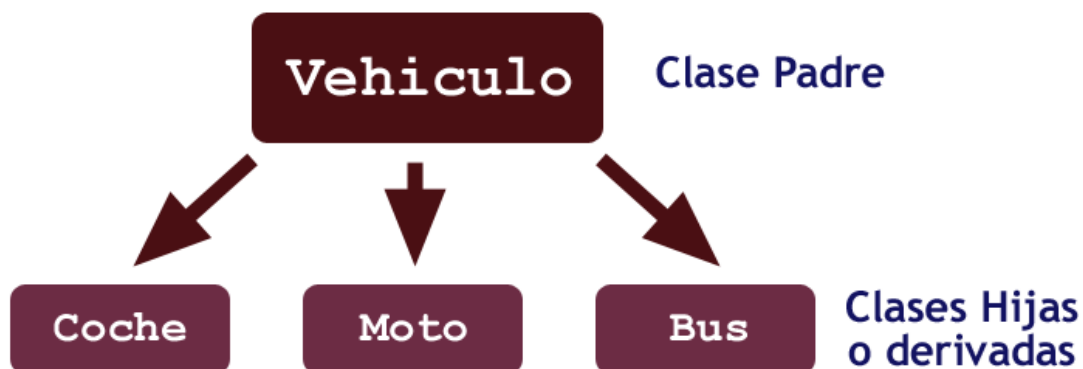
*“m. Cualidad de lo que tiene o puede tener distintas formas”*

**Definición para POO:** El polimorfismo es la capacidad que tiene un objeto de tomar distintas formas, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, ...).

Veremos que el polimorfismo y la herencia son dos conceptos estrechamente ligados. Conseguimos implementar polimorfismo en jerarquías de clasificación que se dan a través de la herencia.

Veamos como expresarlo en un ejemplo.

Tenemos una clase vehículo y de ella dependen varias clases hijas como coche, moto, autobús, etc.



Además, tenemos la clase Parking. Dentro de ésta tenemos un método estacionar(). Puede que en un parking tenga que estacionar coches, motos o autobuses. Sin polimorfismo tendría que crear un método que permitiese estacionar objetos de la clase "Coche", otro método que acepte objetos de la clase "Moto" para estacionarlos, etc. Pero todos estaremos de acuerdo que estacionar un coche, una moto o un bus es bastante similar: "entrar en el parking, recoger el ticket de entrada, buscar un lugar, situar el vehículo dentro de ese lugar...".

Lo ideal sería que nuestro método nos permita recibir todo tipo de vehículos para estacionarlos, primero por reutilización del código, ya que es muy parecido estacionar uno u otro vehículo, pero además porque así si mañana el mercado trae otro tipo de vehículos, como una van, todoterreno híbrido, o una nave espacial, el software sea capaz de aceptarlos sin tener que modificar la clase Parking.

Gracias al polimorfismo, cuando declaramos la función estacionar() podemos decir que recibe como parámetro un objeto de la clase "Vehículo" y el compilador aceptará no solamente vehículos genéricos, sino todos aquellos objetos que hayamos creado que hereden de la clase Vehículo, es decir, coches, motos, buses, etc. Esa cualidad del sistema, para aceptar una gama de objetos diferentes, es lo que llamamos polimorfismo.

Declaro la función:

```
function estacionar ( Vehiculo ) { }
```

Invoco la función: (soporto polimorfismo)

```
estacionar ( Coche ) ;  
estacionar ( Moto ) ;  
estacionar ( Bus ) ;
```

No puedo invocar la función: (no lo permitiría, porque no ser clasificación de herencia de vehículos)

```
estacionar ( Mono ) ;  
estacionar ( INT ) ;
```

En el futuro si podría: (Si creo las clases "Van" o "Nave especial" y heredan de Vehículo)

```
estacionar ( Van ) ;  
estacionar ( Nave espacial ) ;
```

De esta manera, en la POO, se denomina polimorfismo a la capacidad que tiene un objeto de responder al mismo mensaje que se encuentra en distintas clases.

### Polimorfismos, sobrecarga y sobre escritura

Una de las características del polimorfismo es que no se habla de sobrecarga de métodos, sino de sobre escritura.

Veamos, ahora, la definición de cada concepto.

- La **sobrecarga** se produce **dentro de una misma clase** sobre un método que tiene igual nombre, pero distintos parámetros. Dependiendo de los parámetros que se encuentren en la invocación será el método que se termine llamando.

- La **sobre escritura** se produce con **métodos de distintas clases**, donde se encuentra definido un método con el mismo nombre y con los mismos parámetros. Dependiendo del objeto que invoque al método será la clase que resolverá ese llamado.

### **Clases abstractas**

Como te contamos, el polimorfismo es la habilidad que tiene un objeto de tomar diferentes formas en tiempo de ejecución. Nosotros vamos a representar el polimorfismo con la palabra reservada “abstracto”, la cual puede ser utilizada tanto en clases como en métodos.

Es importante que entiendas, también, que para que exista polimorfismo tiene que haber una jerarquía de clases.

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

### **EJEMPLO Código JAVA:**

```
abstract class Estacionar
{
    abstract void acelerar(Auto coupe);
    String frenar() { ... }
}
```

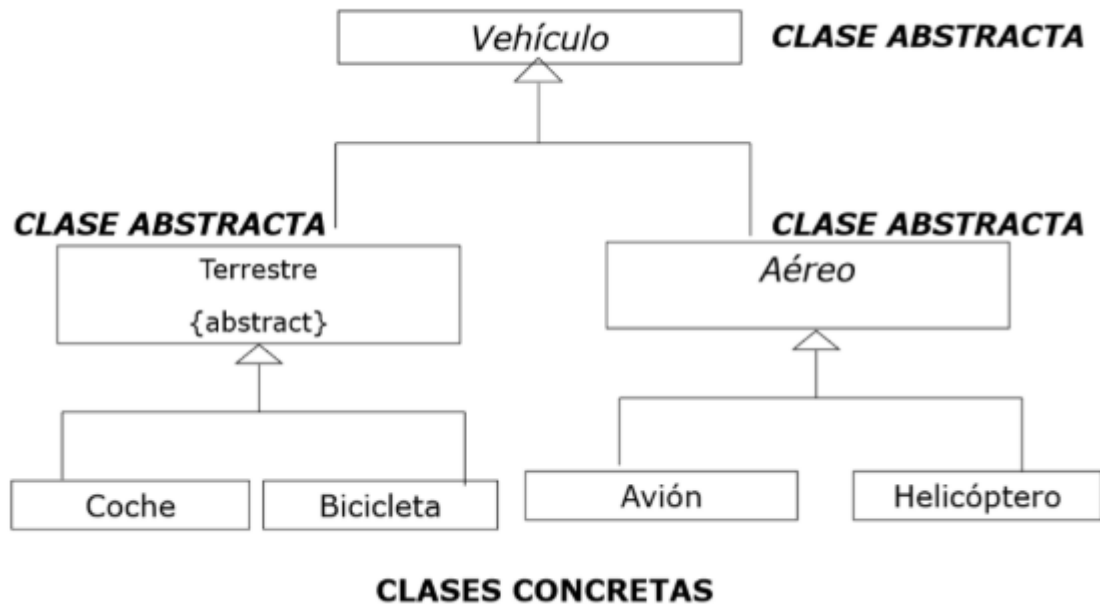
**Una clase abstracta no se puede instanciar pero si se puede heredar y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos.** Si no lo hacen así, las clases hijas deben ser también abstractas.

Si un método tiene antepuesto a su definición la palabra “abstracto” (‘abstract’ en java), entonces decimos que es un método abstracto, es decir que su cabecera se encuentra en la clase base y su implementación se encuentra en alguna clase derivada.

Una clase que posee, al menos, un método abstracto se denomina clase abstracta.

Una **clase abstracta**, por lo tanto, tendrá que derivarse, ya que no podrá hacerse un nuevo objeto o instanciar de esa clase abstracta.

Pondremos otro ejemplo:



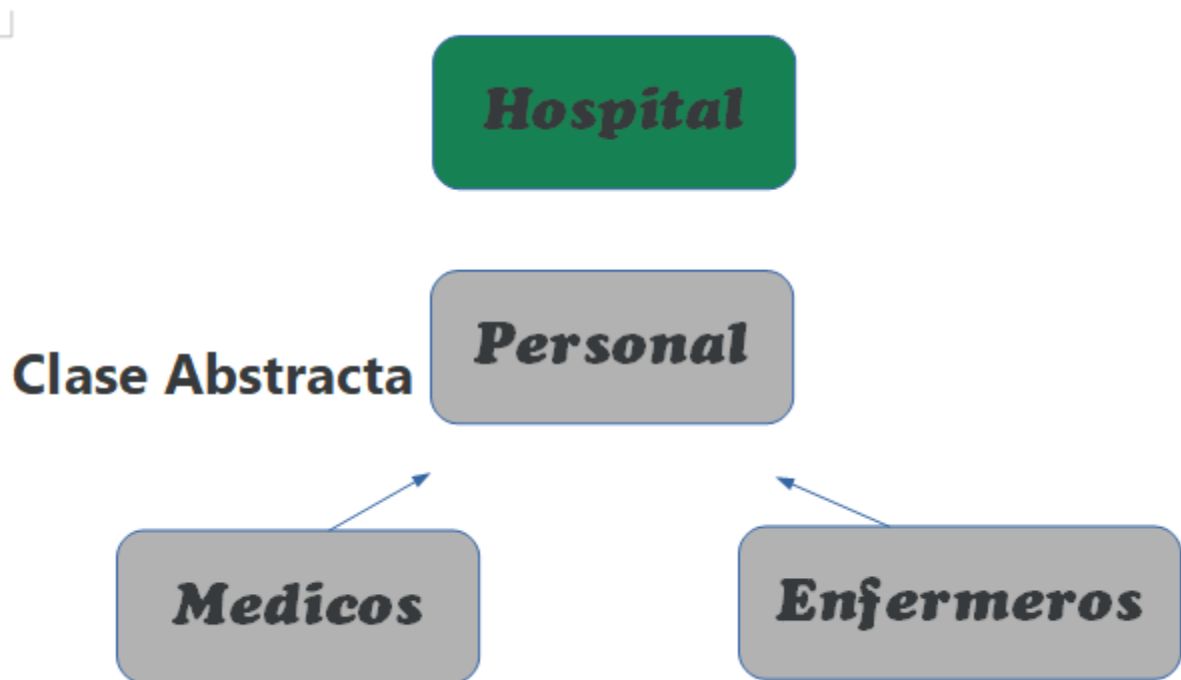
En el cuadro anterior, vemos que la clase Vehículo es Abstracta, las clases Terrestre y Aéreo que heredan a Vehículo, también son abstractas y esto sucede porque necesitamos programar sobre objetos más definidos y con mayor cantidad y calidad de atributos, por ello, heredamos a su vez, las clases Coche (que sabemos que tiene 4 ruedas, un motor, 4 asientos etc.) y la clase Bicicleta (que sabemos que tiene 2 ruedas, pedal, volante etc.) y con ese tipo de clases programamos los objetos que realmente utilizaremos en un sistema.

Trabajar con la abstracción en clases como Vehículo, nos dan la pauta de que ésta clase no será accedida ni modificada por otro usuario y que si necesitamos incorporar un nuevo vehículo “Patineta”, solo nos queda crear la clase y realizar la herencia de Vehículo.

### **EJEMPLO Diagrama:**

Veamos cómo implementamos nuestro ejemplo de polimorfismo.

Para ello, retomemos el ejemplo de la unidad anterior, en donde definimos las clases Personal y Medicos, para realizar las clases que representarían a las personas que trabajan en un Hospital.



A la jerarquía original le agregamos la **clase Enfermeros**, derivada de Personal para que quede más claro cómo funciona el polimorfismo.

Habrás notado que la **clase Personal**, ahora es una clase abstracta, ya que tiene un nuevo método que es abstracto, denominado Sueldo.

Esto significa que cada persona del Hospital tendrá un sueldo diferente, aunque en nuestro programa, las personas serán Médicos o Enfermeros y los objetos serán instancias de las clases Medicos o Enfermeros, no tendremos objetos de Personal porque ahora es abstracto.

Ahora bien, cada uno de los objetos se podrá mostrar el sueldo asignado. Por lo tanto ante el mismo mensaje, las clases responderán con acciones diferentes.

Aquí es donde tenemos que utilizar el polimorfismo para simplificar el desarrollo.

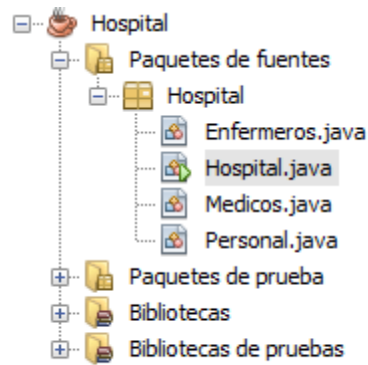
Para eso, definimos un objeto que será Personal y es el objeto polimórfico, ya que en principio será un Medico y luego lo instanciaremos como un Enfermero.

Ese es el momento en que dinámicamente, en tiempo de ejecución, se resuelve a qué clase se estará invocando.

Haremos un proyecto con cuatro archivos, uno para cada una de las clases descriptas en el diagrama anterior.

Tu proyecto debería quedar así:





Ya que el proyecto se llama Hospital, sabemos que el archivo que iniciará el sistema será el que posee el mismo nombre del proyecto, en este caso Hospital.java y que detallamos a continuación. Podrás ver que en él se encuentra el método main() que lo hace iniciar:

### EJEMPLO Código JAVA:

#### 1- Archivo del Proyecto Hospital : Personal.java

```
public class Hospital {  
    public static void main(String[] args) {  
        // Hago la instancia del objeto Medicos y Enfermeros  
        Personal Med = new Medicos("Juan","Perez",33,15000);  
        Personal Enf = new Enfermeros("Roberto","Gomez",25,11000);  
        // Ejecuto los métodos para obtener el sueldo de cada uno  
        Med.Sueldo();  
        Enf.Sueldo();  
    }  
}
```

- En la **primera llamada** se invocará al método Sueldo de la clase Medicos, ya que Personal responde a una instancia como Médico con la sentencia Med.Sueldo();
  - En la **segunda llamada** se invocará al método Sueldo de la clase Enfermeros, ya que Personal responde a una instancia como Enfermeros con la sentencia Enf.Sueldo();

De esta manera es como, ante el mismo llamado del mismo objeto, puede responder de manera distinta ya que el objeto Personal es polimórfico.

Para analizar como es que Personal es polimórfico, debemos analizar los códigos de cada clase:

### EJEMPLO Código JAVA:

#### 2- Archivo del Proyecto Hospital : Personal.java

```
public abstract class Personal{           //declaración de la clase Personal
```

```

//Atributos

    public double sueldo; //declaro como privado el sueldo
    public int edad;      // declaro como público la edad
    public String apellido;    // declaro como público el Apellido
    public String nombre;      // declaro como público el nombre
    public Personal(String nombre, String apellido, int edad, double sueldo){
    }

    abstract void Sueldo();
}

```

Podemos observar en el código anterior que la clase Personal se definió como abstracta y posee un método abstracto (Sueldo) que no tiene ninguna línea de código ya que tendrá que ser codificado en las clases que lo utilicen.

### **EJEMPLO Código JAVA:**

#### **3- Archivo del Proyecto Hospital : Medicos.java**

```

public class Medicos extends Personal{
    private String Matricula;
    private String Especialidad;

    public Medicos(String nombre, String apellido, int edad, double sueldo){
        super(nombre, apellido, edad,sueldo);
        this.sueldo = sueldo;
        Matricula = "35853";
        Especialidad="Clínica Médica";
    }

    @Override
    void Sueldo() { System.out.println("El sueldo de Médico es de " + this.sueldo + "-"); }
}

```

En el caso del código anterior, vemos que existe una clase Medicos que extiende (hereda) de Personal sus atributos y métodos y que utiliza el polimorfismo para modificar el método Sueldo() sobrescribiéndolo con un mensaje que mostrará cual es el sueldo asignado a un médico del Hospital, si vemos el código de Hospital.java veremos que se le asignaron \$ 15,000,-

### **EJEMPLO Código JAVA:**

#### **4- Archivo del Proyecto Hospital : Enfermeros.java**

```

final class Enfermeros extends Personal{

    private String Matricula;

    public Enfermeros(String nombre, String apellido, int edad, double sueldo {
super(nombre, apellido, edad, sueldo);

    this.sueldo = sueldo;

    Matricula = "25836";

    }

    @Override

    void Sueldo() { System.out.println("El sueldo de Enfermero es de " + sueldo + "-"); }

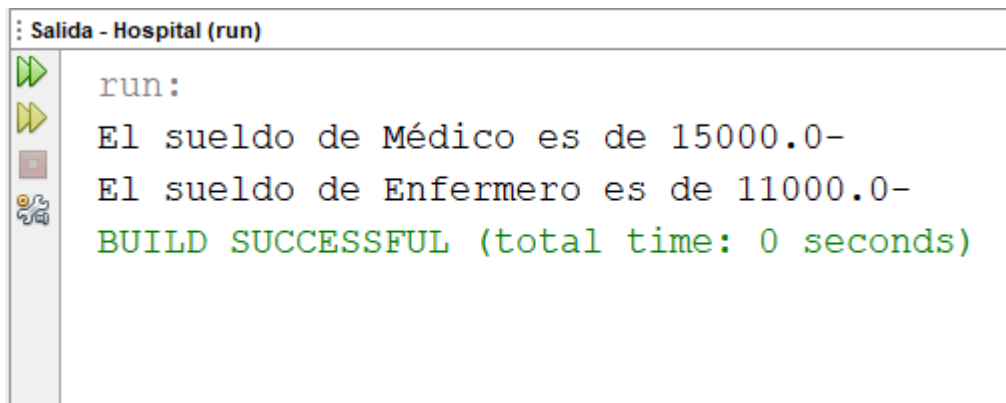
}

```

En el caso del código anterior, vemos que existe una clase Enfermeros que extiende (hereda) de Personal sus atributos y métodos y que utiliza el polimorfismo para modificar el método Sueldo() sobrescribiéndolo (Override) con un mensaje que mostrará cual es el sueldo asignado a un enfermero del Hospital, si vemos el código de Hospital.java veremos que se le asignaron \$ 11,000,-

Definimos a la clase Enfermeros como final, ya que se utiliza declarar una clase así, cuando no nos interesa crear clases derivadas (heredadas) de dicha clase.

Una vez ejecutado el proyecto, tendría que arrojarlos los siguientes mensajes:



```

: Salida - Hospital (run)
run:
El sueldo de Médico es de 15000.0-
El sueldo de Enfermero es de 11000.0-
BUILD SUCCESSFUL (total time: 0 seconds)

```

En donde podemos observar que cada clase Médicos y Enfermeros, utiliza el mismo método Sueldo() para informar cual es el sueldo asignado a cada rama del Personal del Hospital.

## Interfaces

Antes de explicarte este concepto, te hacemos una aclaración con respecto a la palabra interface. Nuevamente, consultamos a la Real Academia Española y nos dice que la palabra “interface” no existe.

El **vocablo correcto, en nuestro idioma, es “interfaz”,** el cual está definido como:

- 1. f. Conexión o frontera común entre dos aparatos o sistemas independientes.*
- 2. f. Inform. Conexión, física o lógica, entre una computadora y el usuario, un dispositivo periférico o un enlace de comunicaciones.*

Sin embargo, en éste como en muchos otros casos, prevalece el uso de un anglicismo – palabra copiada del inglés – y en la jerga de la profesión solemos encontrar “interface” y no exactamente con el sentido que le da el diccionario.

Ahora bien, volviendo a nuestras interfaces en programación, te presentamos el siguiente ejemplo, de un Lavadero de Autos, en el sistema se solicita que cada auto posea un método para Lavar y Lustrar las ruedas:

### **EJEMPLO Código JAVA:**

Definiendo una interfaz

```
public interface Ruedas {  
    public void lavar();  
    public void lustrar();  
}
```

La interfaz es una clase, pero definida con la palabra reservada “interface” ( interfaz). Ésta es una clase especial que contiene todos sus métodos abstractos, por lo tanto, tendrán que ser definidos en las clases que lo implementen.

### **EJEMPLO Código JAVA:**

Implementando una interfaz en el Lavadero de Autos.

```
public class Autos implements Ruedas{  
    public void lavar() {  
        System.out.println("Lavar Neumático del Auto");  
    }  
    public void lustrar() {  
        System.out.println("Lustrar llantas del Auto");  
    }  
}
```

En el ejemplo podemos ver como también está definida la **clase Autos que implementa la interfaz Ruedas**, por medio de la palabra reservada “implements”. Esta definición significa que la clase **Autos** tendrá que definir todos los métodos de la clase **Ruedas**. De esta manera es que se encuentra implementado el método lavar() en Autos, el cual indica "Lavar Neumático del Auto" ya que es un Auto.

Luego de haber analizado este ejemplo, podemos generalizar diciendo que:

Una **interface es una colección de declaraciones de métodos sin definirse**.

Las interfaces se implementan en una clase para poder modelar el comportamiento en común.

Es importante explicarte que una clase puede implementar más de una interface. De esta manera podrás simular el comportamiento de una herencia múltiple. Esta es una característica fundamental ya que mediante las interfaces podrás definir compartimientos comunes sin necesidad de forzar una herencia de clases.

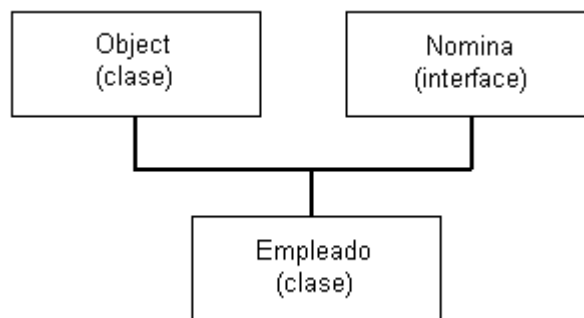
A diferencia de lo que sucede con la Herencia, la **implementación de interfaces** no fuerza una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de comportamiento similares.

### Interfaces y clases abstractas

Ahora te contamos algunas características entre ambos tipos de clases.

- Las **interfaces** son completamente abstractas, no tienen ninguna implementación.
- Con **interfaces** no hay herencia de métodos, con **clases abstractas** sí.
- De una **clase abstracta** no es posible crear instancias; de las **interfaces** tampoco.
- Una clase solamente puede extender una **clase abstracta** (o concreta) pero puede implementar más de una **interface**.

En este ejemplo, la clase Empleado tiene una clase padre llamada Object (implícitamente) e implementa a la interface Nomina, quedando el diagrama de clases de la siguiente manera:



La clase que implementa una interface tiene dos opciones:

- 1) Implementar todos los métodos de la interface.
- 2) Implementar sólo algunos de los métodos de la interface pero esa clase debe ser una clase abstracta (debe declararse con la palabra **abstract**).

### Control de acceso a miembros de una clase

Los modificadores más importantes desde el punto de vista del diseño de clases y objetos, 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:

	<b>public</b>	<b>protected</b>	<b>(sin modificador)</b>	<b>private</b>
Clase	SI	SI	SI	SI
Subclase en el mismo paquete	SI	SI	SI	NO
No-Subclase en el mismo paquete	SI	SI	SI	NO
Subclase en diferente paquete	SI	SI/NO (*)	NO	NO
No-Subclase en diferente paquete (Universo)	SI	NO	NO	NO

**(\*) Los miembros (variables y métodos) de clase (static) si son visibles. Los miembros de instancia no son visibles.**

Como se observa de la tabla anterior, una clase se ve a 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 desde los `public` hasta los sin-modificador. Las subclases de otros paquetes pueden ver los miembros `public` y a los miembros `protected`, éstos últimos siempre que sean `static` ya de no ser así no serán visibles en la subclase (Esto se explica en la siguiente página). El resto del universo de clases (que no sean ni del mismo paquete ni subclases) pueden ver sólo los miembros `public`.

### **Paquetes /Package**

Para hacer que una clase sea más fácil de localizar y utilizar así como evitar conflictos de nombres y controlar el acceso a los miembros de una clase, las clases se agrupan en paquetes.

- **Paquete**

Un paquete es un conjunto de clases e interfaces relacionadas.

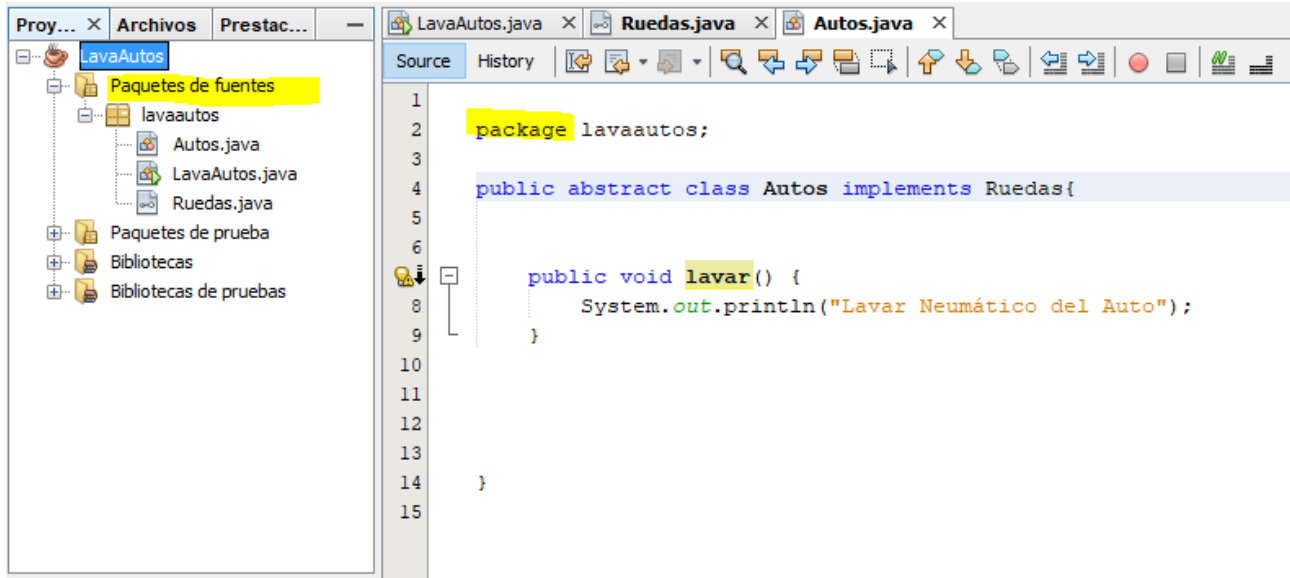
La forma general de la declaración **package** es la siguiente:

**package** nombrePaquete;

donde nombrePaquete puede constar de una sola palabra o de una lista de nombres de paquetes separados por puntos.

Ejemplo

En Netbeans verás como los paquetes se forman automáticamente gracias al IDE:



En el caso de realizar la escritura manual de un archivo java los paquetes pueden definirse:

## EJEMPLO Código JAVA:

### Ejemplo 1.

**package** miPaquete;

class MiClase

```
{
...
}
```

### Ejemplo 2

**package** nombre1.nombre2.miPaquete;

class TuClase

```
{
...
}
```

Los nombres de los paquetes se corresponden con el nombre de directorios en el sistema de archivos. De esta manera, cuando se requiera hacer uso de estas clases se tendrán que importar de la siguiente manera.

### Ejemplo 3

```
import miPaquete.MiClase;  
import nombre1.nombre2.miPaquete.TuClase;
```

```
class OtraClase  
{  
    /* Aqui se hace uso de la clase 'MiClase' y de la  
    clase 'TuClase' */  
    ...  
}
```

Para importar todas las clases que están en un paquete, se utiliza el asterisco ( \* ).

### Ejemplo 4

```
import miPaquete.*;
```

#### otros ejemplos

```
package arraylist011;  
import java.util.ArrayList;
```

```
package ejer10gui2;  
import java.io.*;
```

Si no se utiliza la sentencia package para indicar a que paquete pertenece una clase, ésta terminará en el package por default, el cual es un paquete que no tiene nombre.