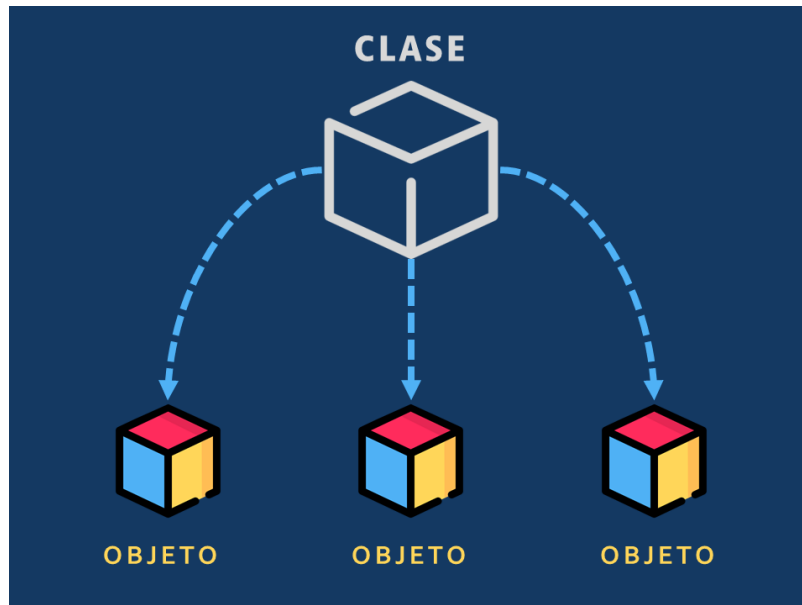


Temas a tratar:

- Herencia
- Paquetes
- Polimorfismo
- Casteo
- Arrays-foreach

¿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.

Si bien este tema lo estudiaremos en detalle más adelante, llegamos a un punto en el que debemos saber que en Java, por transitividad, todas las clases heredan de una clase base llamada Object. No hay que especificar nada para que ocurra esto. Siempre es así.

La herencia es transitiva. Esto quiere decir que, sean las clases A, B y C, si A hereda de B y B hereda de C entonces A hereda de C.

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.

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

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 Médicos? 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.

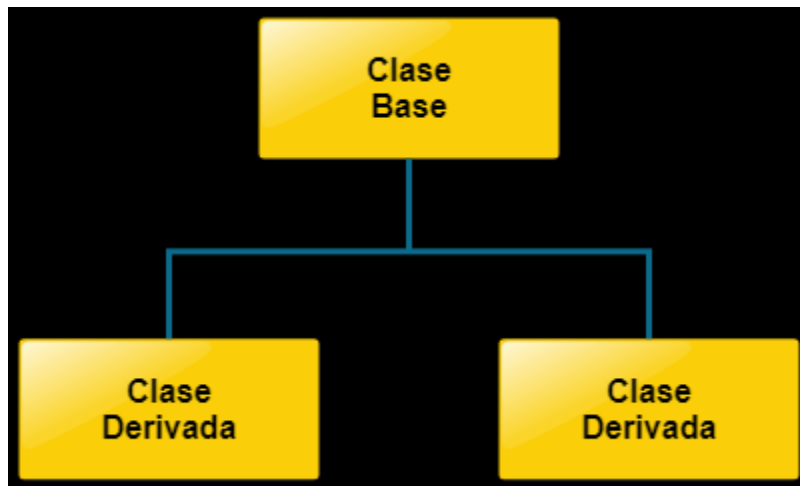
De esta manera podemos decir que:

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

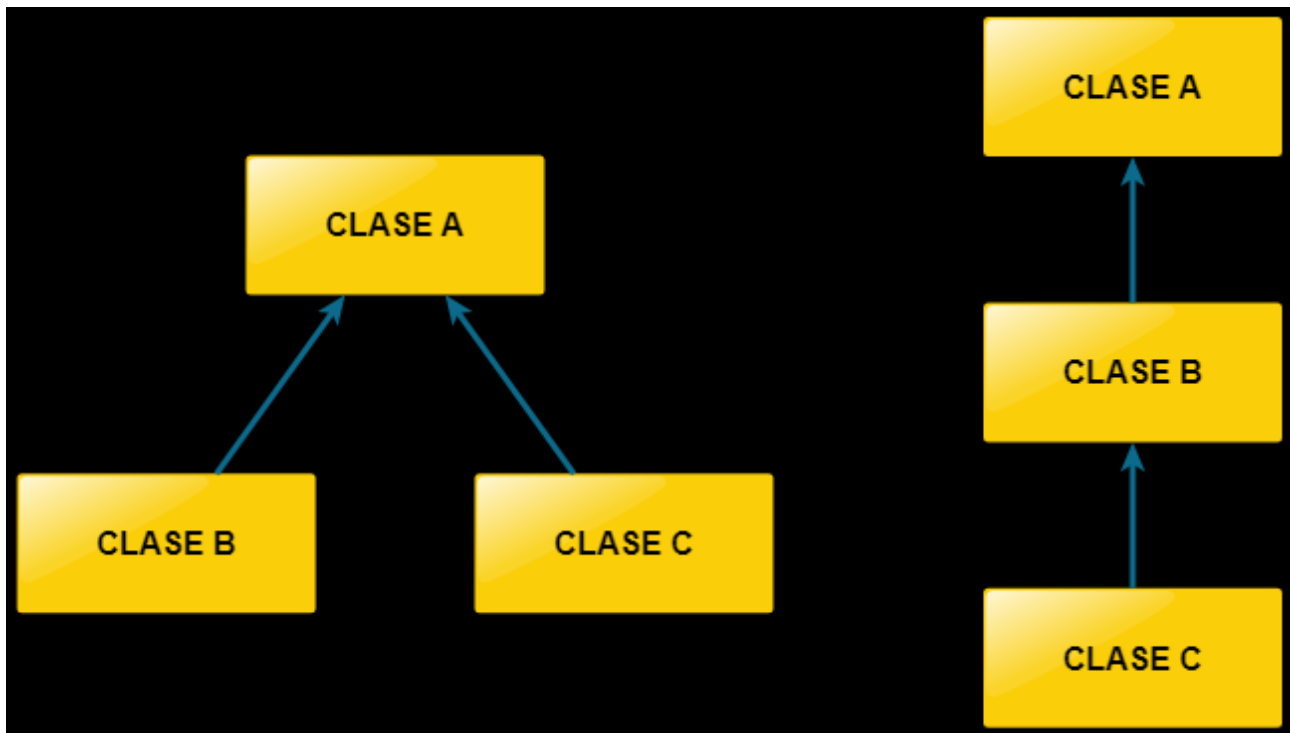
En la **POO existen dos tipos de herencia.**

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

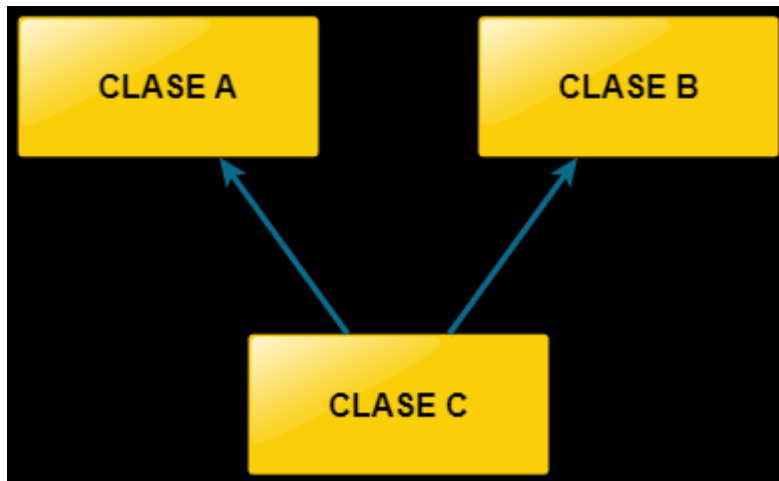
En la **POO existen dos tipos de herencia**.

2- Herencia Múltiple

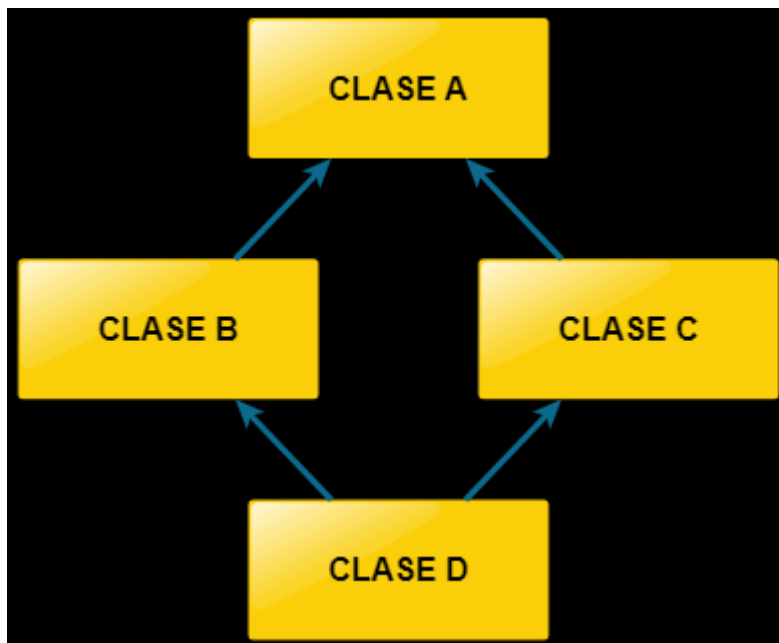
Hay otro tipo de herencia y es en el caso en que una clase puede extender las características de varias clases, es decir, puede tener varios padres. En este aspecto hay discrepancias entre los diseñadores de lenguajes. Algunos de ellos han preferido no admitir la herencia múltiple por las posibles coincidencias 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	Privado	Solo mediante métodos
2	Protegido	Público	Directo
3	Público	Público	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.

El método toString

Todas las clases heredan de Object el método toString, por lo tanto, podemos invocar este método sobre cualquier objeto de cualquier clase. Tal es así que cuando hacemos:

```
System.out.println(obj);
```

siendo obj un objeto de cualquier clase, lo que realmente estamos haciendo (implícitamente) es:

```
System.out.println( obj.toString() );
```

Ya que System.out.println invoca el método toString del objeto que recibe como parámetro. Es decir, System.out.println "sabe" que cualquiera sea el tipo de datos (clase) del objeto que recibe como parámetro este seguro tendrá el método toString

Relación entre clases y objetos

Un objeto es una INSTANCIA de una clase. Por lo tanto, los objetos hacen uso de los Atributos (variables) y Métodos (Funciones y Procedimientos) de su correspondiente Clase.

También lo podemos pensar como una variable de tipo clase. Por ejemplo: el objeto Cesar es un objeto de tipo clase: Persona.

Como se puede observar, un objeto a través de su CLASE está compuesto por 2 partes: Atributos o estados y Métodos que definen el comportamiento de dicho objeto a partir de sus atributos. Los atributos y los métodos pueden ser o no accedidos desde afuera dependiendo de la solución a plantear.

Por lo general los atributos siempre se ocultan al exterior y algunos métodos quedan visibles al exterior para convertirse en la interfaz del objeto.

Por el contrario, los objetos son instancias particulares de una clase. Las clases son una especie de molde de fábrica, en base al cual son contruidos los objetos.

Durante la ejecución de un programa sólo existen los objetos, no las clases. La declaración de una variable de una clase NO crea el objeto.

La creación de un objeto debe ser indicada explícitamente por el programador, de forma análoga a como inicializamos las variables con un valor dado, sólo que para los objetos se hace a través del método CONSTRUCTOR

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.

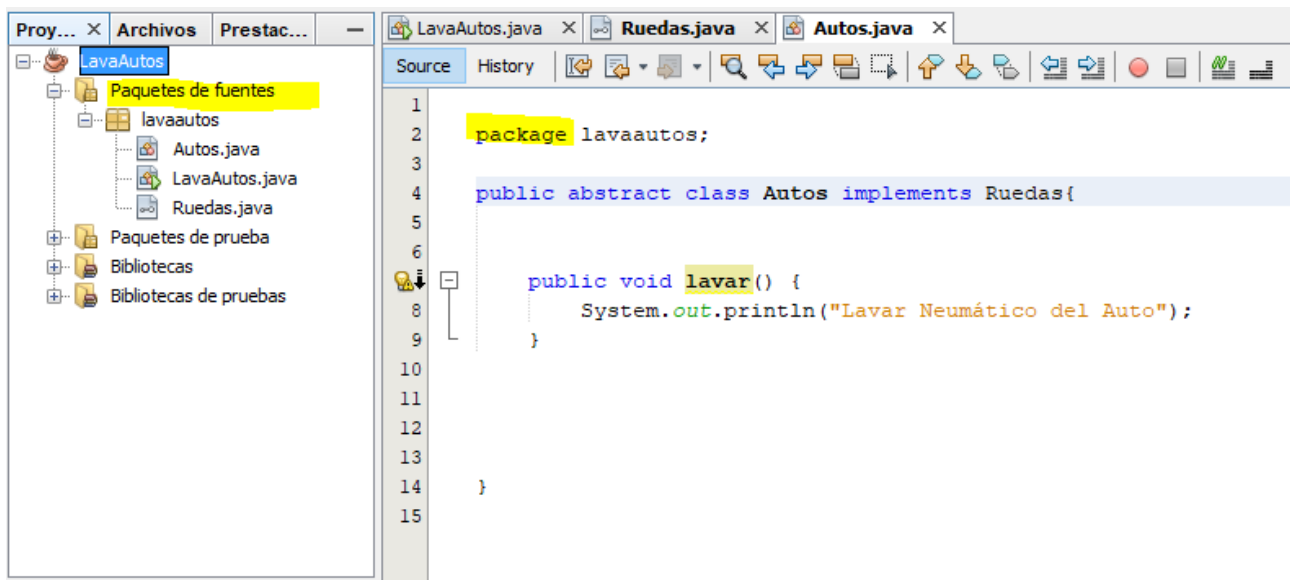
La forma general de la declaración **package** es la siguiente:

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.

¿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:

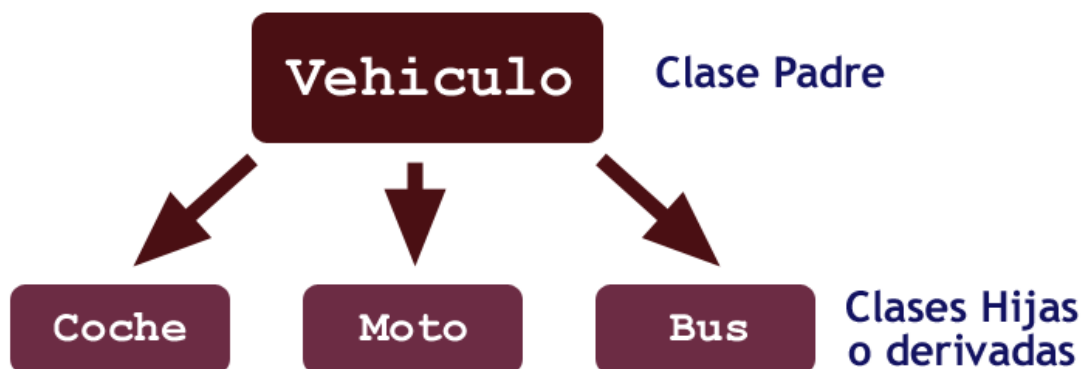
“m. Cualidad de lo que tiene o puede tener distintas formas”

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() { ... }
}
```

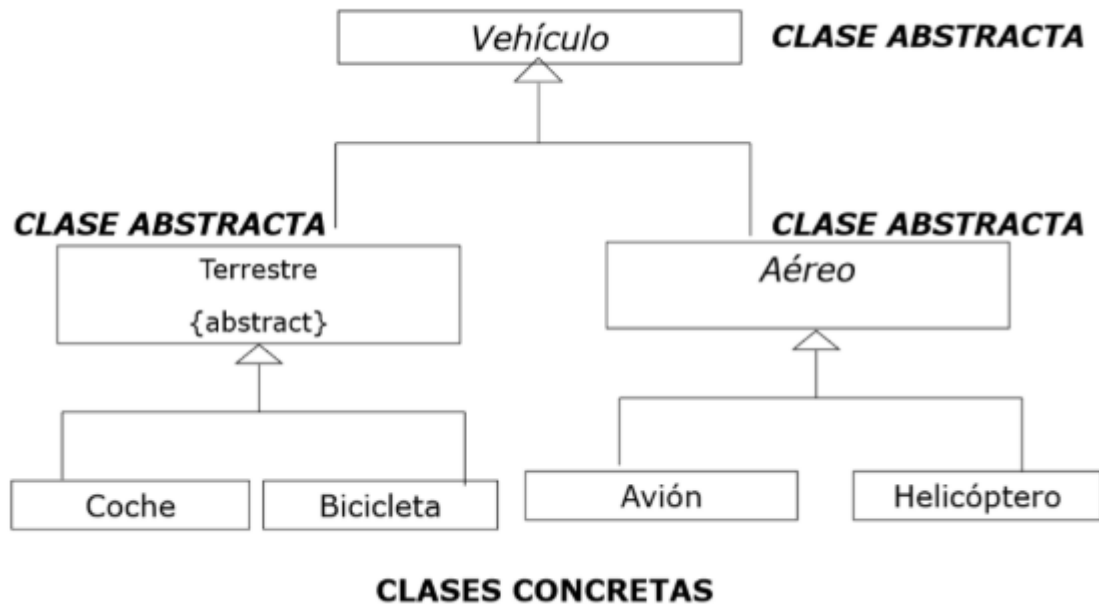
Una clase abstracta no se puede instanciar pero si se puede heredar y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

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:



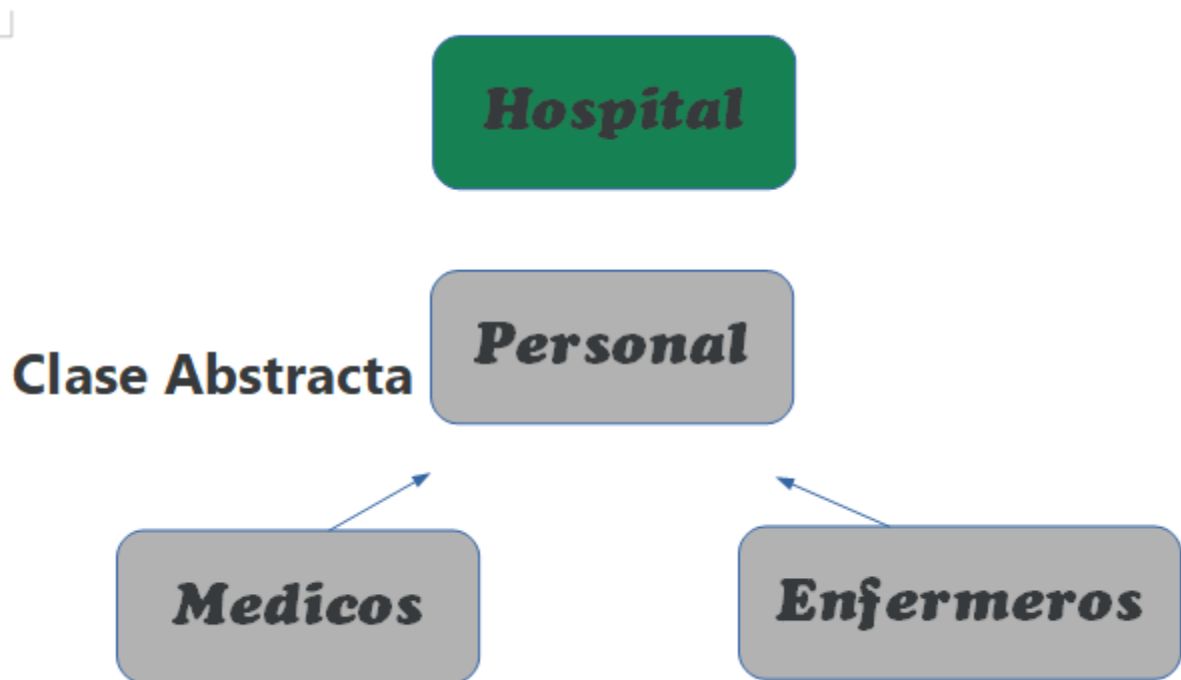
En el cuadro anterior, vemos que la clase Vehículo es Abstracta, las clases Terrestre y Aéreo que heredan a Vehículo, también son abstractas y esto sucede porque necesitamos programar sobre objetos más definidos y con mayor cantidad y calidad de atributos, por ello, heredamos a su vez, las clases Coche (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.

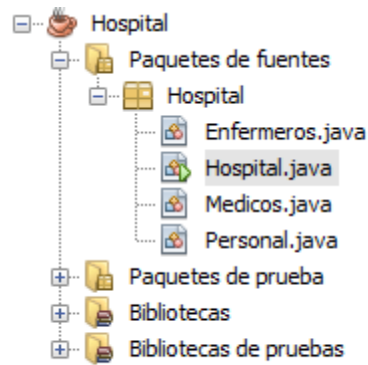
Aquí es donde tenemos que utilizar el polimorfismo para simplificar el desarrollo.

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 + "-"); }

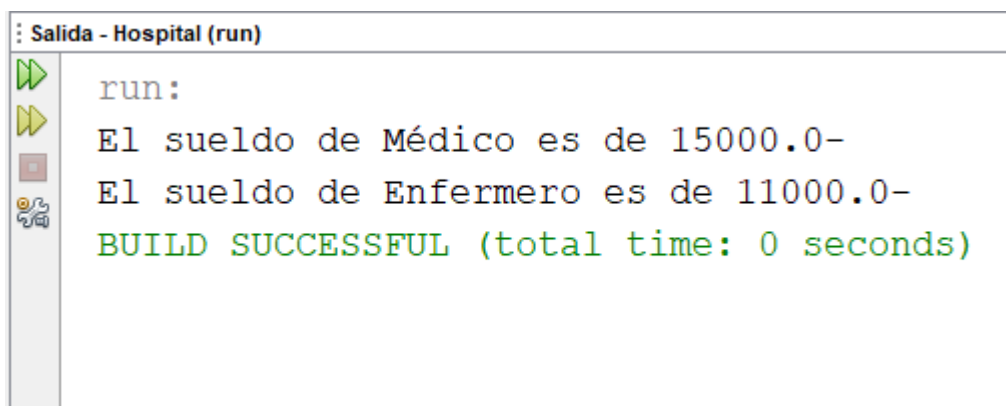
}

```

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.

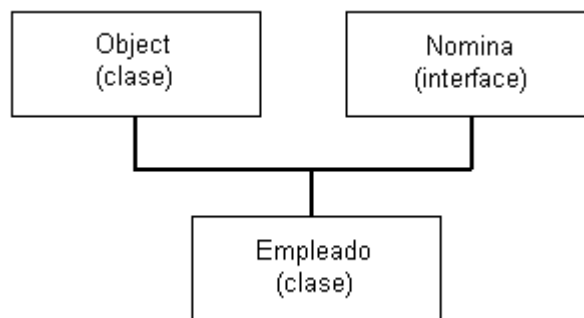
A diferencia de lo que sucede con la Herencia, la **implementación de interfaces** no fuerza una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de comportamiento similares.

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:

	public	protected	(sin modificador)	private
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`.

Casteo

Muchas veces nos vamos a encontrar con la necesidad de convertir algún tipo de dato; por ello java al ser un lenguaje de tipado fuerte nos facilita las herramientas para poder lograrlo, existen dos tipos de casteo, implícito y explícito

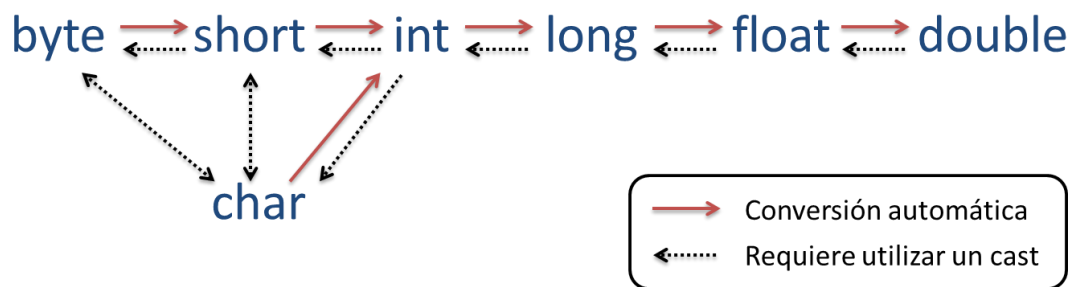
Casteo implícito

El mismo ocurre cuando necesitamos un tipo de dato más pequeño en otro más grande, lo podemos ver en varios ejemplos de números y hace de manera automática:

```
byte x = 5;
```

```
int y = x;
```

El valor asignado en x, se convertirá de forma automática en entero al asignarlo en y



Casteo Explícito

Este tipo de casteo va a ocurrir cuando queremos asignar un tipo de dato más grande a un tipo de dato más pequeño, es tarea del usuario especificar al nuevo tipo de dato al cual se va a transformar. Se escribe de forma explícita entre paréntesis.

```
byte a = 20;
```

```
int b = (int) a;
```

al escribir int entre los paréntesis se fuerza a cambiar al tipo de dato entero.

Clases envoltorio (Wrapper)

Para todos los tipos de datos primitivos, existen unas clases llamadas Wrapper, también conocidas como envoltorio, ya que proveen una serie de mecanismos que nos permiten envolver a un tipo de dato primitivo permitiéndonos con ello el tratarlos como si fueran objetos.

Tipos primitivos (no son objetos y por tanto no poseen métodos)	Wrappers(son objeto y por tanto poseen métodos)
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

Vectores

Los arreglos (o arrays en inglés), son un conjunto de datos que se caracterizan por almacenarse en memoria de manera contigua, bajo un mismo nombre, pero con diferentes “índices” para diferenciar la ubicación de cada uno de ellos.

Los arreglos son estructuras fijas, es decir, que una vez declarados e inicializados, mantienen su tamaño durante toda la ejecución del programa. ¿Qué quiere decir esto? Por ejemplo, si se declara e inicializa un arreglo con 5 posiciones, estas 5 se mantendrán de principio a fin de la ejecución del programa que se esté desarrollando sin la posibilidad de cambiar su tamaño.

Los arreglos, al igual que las variables comunes, deben poseer un solo tipo de dato determinado. Este tipo de dato debe ser único para todos los elementos que conforman el array con el que se esté trabajando. Veamos un ejemplo de un arreglo de tipo numérico en el cual declaramos el tipo arraya así como la cantidad de elementos:

```
int numerosB[] = new int[3];  
  
numerosB[0]=10;
```

Una vez declarado e inicializado un vector, es posible asignarle diferentes valores en cada una de sus posiciones a partir de conocer el índice donde estos datos.

índice
↓

índice	0	1	2	3
valores	15	30	45	50

Tamaño del array = 4

Un detalle muy importante a tener en cuenta es que, por convención mundial, los vectores comienzan su índice en el valor 0. ¿Qué quiere decir esto? Que si tenemos un vector de 4 posiciones, sus índices irán del 0 al 3, por lo que si hacemos referencia al índice 4, no estaríamos posicionados en la 4ta posición, esto, al tratarse de un vector de únicamente 4 posiciones provocaría un error por desbordamiento.

```
int numeros[]={1,2,3,4,5,6};
```

For-each

Al trabajar con arrays, es común encontrar situaciones en las que cada elemento de una matriz debe examinarse, de principio a fin. Por ejemplo, para calcular la suma de los valores contenidos en una matriz, cada elemento de la matriz debe examinarse.

```
int numeros[]={1,2,3,4,5,6};

for(int i=0; i<numeros.length;i++){

    System.out.println("Los numeros son " + numeros[i]);
}
```

El uso de Java ForEach nos permite recorrer la lista de elementos de una forma más compacta y el código se reduce.

```
for(int cadena:numeros){

    System.out.println("Los numeros son "+cadena);
}
```

Este bucle recorre un arreglo de principio a fin de forma secuencial, es importante recordar que al no tener un índice o un valor por defecto no funciona para modificar un valor.