JAVA - POO

QUÉ ES

JAVA: Lenguaje de programación de propósito general. Lenguaje de alto nivel (más cercano al lenguaje humano que al de las máquinas). Permite escribir codigo ejecutable en distintos S.O. Es un lenguaje que utiliza el paradigma de la programación orientada a objetos. El codigo representa abastracciones. Es un lenguaje fuertemente tipado, es necesario establecer el tipo de datos de antemano.

MÉTODO MAIN

Estandar utilizado por la JVM (Java Virtual Machine). Es el punto de entrada de la aplicacion. Cuando uno apreta play, el contenido de main se ejecuta linea por linea.

```
public class Main{

    public static void main (String [] args) {

        //aquí va tu código
    }

}
```

TIPOS DE DATOS PRIMITIVOS

Tipos primitivos: llamamos así a los tipos de datos que solo nos permiten almacenar un valor. Cuando definimos una variable con estos tipos primitivos, solo podemos almacenar valores.
 Cuando definimos una variable con estos tipos primitivos, solo podemos almacenar valores.

TIPO INTEGER / TIPO FLOATING POINT

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

DECLARACIÓN DE UNA VARIABLE

Para declarar una variable, es necesario indicar el tipo de dato y nombre que se le asigna. Recordar que Java es un lenguaje case sensitive, los tipos de datos siempre se escriben en minúscula. Para los tipos comunes vamos a ver una excepción: el tipo **String** que siempre lo inicializamos con mayúscula.

```
int valor;
double coeficiente;
String nombre;
```

Una vez declarada la variable, sólo podrá utilizarse con datos del tipo indicado, es decir, una variable de tipo int no podrá almacenar un valor de tipo double

ESTRUCTURAS DE CONTROL

Estructuras

Estructura de decisión

```
if (condición){
  //código que se corre si la condición es verdadera
}
else {
  //código que se corre la condición no fue verdadera
}
```

```
switch (variable){
        case valor1:
        //código que se ejecuta si la variable tiene valor1
        break;
        case valor2:
        //código que se ejecuta si la variable tiene valor2
        break;
        .
        .
        default:
        //código que se ejecuta si la variable tiene algún valor no enumerado

}
```

Estructuras de repetición

```
for(Integer i = 0; i < valorMaximo; i++){
        //código que se ejecuta cada vez
}
```

```
for(Object object : listaDeObjetos){
        //código que se va a ejecutar por cada objeto en la lista
}
```

```
while(condición){
        //hacer este código
}
```

CLASES

En este caso tendremos un elemento que, además de almacenar un valor, nos permite realizar ciertas operaciones que ya vienen programadas, a estas operaciones las llamamos métodos. Por ejemplo, String es una clase, por eso, se la inicializa en mayúscula.
Todas las clases las nombramos con la **inicial en mayúscula.**
Las clases Integer y Float son equivalentes a los tipos de datos primitivos, es decir, me permiten almacenar valores de los tipos indicados, pero además me dan ciertas funcionalidades. Se suele decir que envuelven los tipos primitivos.
Algo a tener en cuenta cuando usamos estas clases es que no podemos usar operadores como "==", para efectuar una comparación por igual usamos .equals()
Si queremos comparar si un valor es mayor o menor que otro debemos usar .compareTo()

1 STRING

Para utilizar datos de tipo texto, vamos a declararlos como String. Las Strings nos permiten utilizar funciones ya programadas, que le pertenecen. Las llamamos métodos:

- **.length()**
- **.toUpperCase()**
- **.equals()**
- **.toChar()**

String vacía: Si aún no hemos asignado nada a las String, entonces, contiene un valor null, en ese caso no se pueden usar los métodos.

2 INTEGER

Integer como clase y no como tipo primitivo se utiliza de una forma distinta. Para comenzar a utilizar un Integer tenemos dos posibilidades:
Integer valor=0; En este caso definimos y creamos un Integer, dándole un valor inicial 0.
Integer num= new Integer(1); En la segunda forma hacemos algo similar, pero la parte de la izquierda es la definición y la parte de la derecha la creación con un valor inicial 1.
Cuando solo definimos algo de tipo Integer, su **valor inicial es null**, es necesario darle un valor inicial. Métodos para comprobar la relación entre dos numeros enteros:

- **.equals()**
- **.compareTo()**

3 FLOAT

Float como clase y no como tipo primitivo se utiliza de una forma distinta. Para comenzar a utilizar un Float tenemos dos posibilidades:
Float coeficiente=2.5f; En este caso definimos y creamos un Integer, dándole un valor inicial 2.5f, la f quiere decir float, si no lo ponemos se asume que es algo de tipo Double.
Float num= new Float(0.5); En la segunda forma hacemos algo similar, pero la parte de la izquierda es la definición y la parte de la derecha la creación con un valor inicial 0.5.
Al igual que Integer, si no tiene un valor inicial, está en null.

4 DATE

La clase Date permite trabajar con fechas. A diferencia de las clases que vimos hasta ahora, si definimos un objeto de tipo Date, no es posible hacerlo vacío. Un objeto Date se crea con un valor inicial que el la fecha actual. Para usar Date es necesario agregar import java.util.Date;

```
import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Date fecha=new Date();
        System.out.println(fecha.toString());
```

Muestra la fecha actual.

```
public static void main(String[] args) {
    Date fecha=new Date(120,11,5);
    System.out.println(fecha.toString());
```

Muestra 5/12/2020. Tener en cuenta: al valor que usamos para año, le suma 1900. Los meses los enumera desde 0, o sea 11 en realidad es 12 (diciembre)

5 SYSTEM

Una clase muy importante es System, en ella encontramos System.in y System.out, que nos permitirán interactuar con las entradas y salidas del programa.

6 SCANNER

Es una clase propia de Java, que nos permite ingresar valores. Tiene métodos, funciones ya programadas, que nos permiten ingresar distintos tipos de datos. Creación:

1.1.Cuando definimos nuestro elemento de tipo Scanner, nos aparece una indicación que significa que para poder utilizarlo debemos agregar la clase correspondiente, que se encuentra en java.util.
2. Definición: Cuando aceptamos la sugerencia, nos agrega el import, finalizamos la definición dándole un nombre como lo haríamos con cualquier variable.
3.Creación del objeto Scanner: Luego de definirlo, es necesario crear el objeto u instanciarlo.

Métodos de Scanner

- **nextByte()** para leer un dato de tipo byte.
- **nextShort()** para leer un dato de tipo short.
- **nextInt()** para leer un dato de tipo int.
- **nextLong()** para leer un dato de tipo long.
- **nextFloat()** para leer un dato de tipo float.
- **nextDouble()** para leer un dato de tipo double.
- **nextBoolean()** para leer un dato de tipo boolean.
- **nextLine()** para leer un string hasta encontrar un salto de línea.
- **next()** para leer un string hasta el primer delimitador, generalmente hasta un espacio en blanco o hasta un salto de línea.

Ejemplo:

```
import java.util.Scanner;

public class Add {

    public static void main(String[] args) {
        System.out.println("Input the first number to add");
        Scanner scan = new Scanner(System.in);
        int num1 = scan.nextInt();
        System.out.println("Input the second number to add");
        int num2 = scan.nextInt();
        int num3 = num1 + num2;
        System.out.println("The answer: " + num3);
    }
}
```

FUNCIONES

¿Cómo definimos una función?
Para definirla vamos a considerar 3 cosas:
- **Qué devuelve** la función
- Qué **nombre tiene**
- Los **parámetros** que necesitamos.

```
tipo devuelto nombre (parámetros){
//código que ejecuta la función
}

int suma (int a, int b){
return a+b;
} //devuelve un valor de tipo int

void mostrarMensaje(String mensaje){
System.out.println (mensaje);
} //no retorna, imprime por pantalla.
```

ARRAYS

Los arrays son estructuras de datos estáticas que permiten guardar elementos del **mismo tipo** en forma contigua. En Java, un array es un objeto y, como tal, debe usarse el operador new para crear una instancia, pero a diferencia de las colecciones, los array **son de longitud fija,** la cual debe definirse en la creación, siendo inmutable.

Con los corchetes "[]" se indica que es un array.

```
String[] nombres = new String[5];
```

El tipo puede ser o un tipo primitivo o cualquier clase de objeto.

Al instanciarlo se debe definir dentro del [] el tamaño de la estructura.

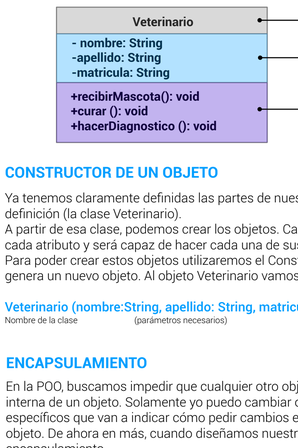
nombres	
0	null
1	null
2	null
3	null
4	null

Establecemos valores a un array a través de su índice. Dado que es una estructura fija, **no se pueden eliminar elementos.**
nombres[0] = "Juan";
nombres[1] = "Mario";
nombres[3] = "Marcelo";

Podemos recorrer un array a través de un ciclo for, while o for each y también utilizar la propiedad length que nos indica el tamaño del array.

OBJETOS Y CLASES

Un objeto es algo que tiene araceterísticas(**atributos**) y **responsabilidades.** Siempre que pensemos en los objetos que compondrán nuestro sistema, tenemos que analizarlos según el contexto en el que estamos trabajando.



Ejemplo orientado a una veterinaria:
¿Qué objetos participan?
1. Cliente
2. Veterinario
3. Diagnóstico
4. Historial de diagnósticos
Veamos al **veterinario:**
¿Cuáles son sus atributos?
Veterinario:
 Nombre
 Apellido
 Matrícula
¿Qué responsabilidades tiene?
Darle diagnostico a las mascotas. (en este contexto)

CLASES E INSTANCIAS

Clase mascota= hace referencia al concepto de mascota, a la idea que representa y abstrae lo que entendemos como una mascota. Es una abstracción , un molde, una idea que representa el concepto de mascota.
Objetos: instancias concretas de ese molde (clase) que representan elementos concretos de nuestro sistema. Por ej de la clase mascota, puedo tener el objeto perro, canario, etc. Las palabras instancias y objetos son sinónimos.

CLASE
(Concepto)

INSTANCIAS
(Objetos concretos)

ATRIBUTOS Y RESPONSABILIDADES

Los **atributos** de un objeto son las características y propiedades distintivas que permiten darle significado.

Mientras que las responsabilidades o comportamientos son la manera en que actúa o reacciona un objeto —es decir, es lo que representa la actividad visible y comprobable exteriormente—, en la programación orientada a objetos vamos a llamar al comportamiento de los objetos: **"métodos"**, los cuales nos van a permitir establecer cómo van a responder los objetos cuando interactuemos con ellos

Representación en diagrama UML
En los metodos anotaremos primero el nombre de la responsabilidad, y luego entre parentesis los parámetros en caso de ser necesarios. Tambien podemos definir si el método va a devolver un resultado, señalandolo con : y el tipo de dato que va a devolver al ejecutarse. Al igual que los atributos, podemos agregar un + o un - del lado izquierdo, aclarando si el método es **público o privado.**

CONSTRUCTOR DE UN OBJETO

Ya tenemos claramente definidas las partes de nuestro objeto, ya podemos dejar lista nuestra definición (la clase Veterinario).
A partir de esa clase, podemos crear los objetos. Cada objeto tendrá sus valores propios de cada atributo y será capaz de hacer cada una de sus responsabilidades.
Para poder crear estos objetos utilizaremos el Constructor que será quien, a partir de la clase, genera un nuevo objeto. Al objeto Veterinario vamos a darle un método constructor:

Veterinario (nombre:String, apellido: String, matricula:String)
Nombre de la clase (parámetros necesarios)

ENCAPSULAMIENTO

En la POO, buscamos impedir que cualquier otro objeto pueda tener acceso a la estructura interna de un objeto. Solamente yo puedo cambiar o mostrar mi estado y con los métodos específicos que van a indicar cómo pedir cambios en dichos atributos desde el exterior del objeto. De ahora en más, cuando diseñamos nuestros objetos, tenemos que tener en cuenta el encapsulamiento.

Importante
- Cuando definamos un objeto, dejar sus atributos privados.
- Los métodos que sean públicos serán vistos por los otros objetos.
- Usar siempre métodos públicos para ver o modificar las características de tus objetos.
- Para cambiar el valor de un atributo se usa un método set, por ejemplo, para cambiar el nombre será setNombre(String)
- Para obtener el valor de un atributo se usa un método get, por ejemplo, para saber el nombre será getNombre(): String
- Los métodos para ver o cambiar atributos se los denomina **getters y setters** respectivamente.

UML

UML son las siglas para Unified Modeling Language, que en castellano significan: Lenguaje de modelado unificado. Es un lenguaje de modelado, de propósito general, usado para la visualización, especificación, construcción y documentación de sistemas orientados a objetos.

:**Diagrama de clases**: Muestra una vista estática de la estructura del sistema, o de una parte de este, describiendo qué atributos y comportamiento debe desarrollar con los métodos necesarios para llevar a cabo las operaciones del sistema.

IMPLEMENTACIÓN EN JAVA

Atributos: Los nombres de los atributos **comienzan con minúscula**, si necesitamos usar más de una palabra, a partir de la segunda inicializamos en mayúscula. EJ: elAtributo

Clases: Los nombres de las clases siempre van con la i**nicial en mayúscula**, si necesitamos usar dos o más palabras para nombrar una clase van pegadas y con todas las iniciales en mayúscula. EJ: CamelCase, Empleado.

Métodos: Se nombran de la misma forma que los atributos, la primera palabra en minúscula y si el nombre tuviera más palabras, todas se inicializan en mayúscula. Recomendamos poner nombres lo más descriptivos posibles, aunque esto implique usar varias palabras. EJ: calculoSueldoNeto

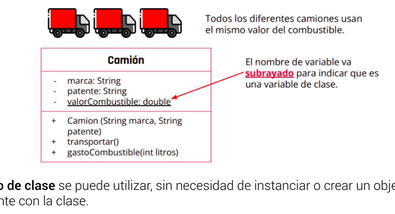
Paquetes: Todas las letras en **minúscula.**

Objetos: La primera palabra en minúscula y si tiene más de una palabra, las siguientes se inicializan en mayúscula. EJ: nombre, importeTotal

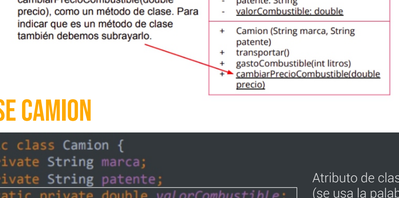
Constantes: Todas las letras en mayúscula y si hay más de una palabra, separadas por guión. EJ: IVA, DIAS_SEMANA

VARIABLES Y MÉTODOS DE CLASE

¿Qué comparten los objetos? Todos los objetos de una clase tienen la misma estructura: los mismos atributos y el mismo comportamiento, es decir, pueden hacer lo mismo. Pero cada objeto tiene sus propios atributos, puede tener distintos valores en sus atributos, tiene un estado propio.
Vamos a llamar **variables de clase** a aquellas variables —atributos— que guardan valores comunes a todos los objetos. Por ej el combustible vale lo mismo para cualquier objeto Camion. En el **UML** se subrayan las variables de clase.



Un **método de clase** se puede utilizar, sin necesidad de instanciar o crear un objeto, directamente con la clase.



CLASE CAMION

```
public class Camion {
   private String marca;
   private String patente;
   static private double valorCombustible;

   public Camion(String marca, String patente){
      this.marca=marca;
      this.patente=patente;
   }
   public double gastoCombustible(int litros){
      return litros*Camion.valorCombustible;
   }
   static public void cambiarPrecioCombustible(double precio){
      Camion.valorCombustible=precio;
   }
}
```

Atributo de clase (se usa la palabra reservada **static**)

Método de clase (se usa la palabra reservada **static**)

EN EL MAIN

```
public class Main {

   public static void main(String[] args) {

      Camion miCamion = new Camion("Ford","AB XXX CD");

      Camion.cambiarPrecioCombustible(98.50);

      Sytem.out.println("Gasto " + miCamion.gastoCombustible(40));
   }
}
```

Creamos un objeto de la clase Camion

Utilizamos el método de clase a través de la **clase** y no del objeto.

CLASE ARTICULO

```
public class Articulo{
    private String descripcion;
    private double precioVenta;
    private int stock;

+   public Articulo(String descripcion, int cantidad,double precio)

    public boolean hayStock(){
        return stock>0;
    }
    public double consultarPrecio(){
        return precioVenta;
    }

    public String getDescripcion(){
          return descripcion;}
    public double getPrecioVenta(){
          return precioVenta;}
    public int getStock(){
          return stock;}
    public void setDescripcion(String descripcion){
          this.descripcion= descripcion;}
    public void setPrcioVenta(double precio){
          precioVenta=precio;}
    public void setStock(int stock){
          this.stock=stock;}

}
```

Clase Articulo

Atributos privados

Constructor

Métodos públicos

Getters y Setters

ASÍ SE VE EN EL UML

RELACIONES ENTRE CLASES

TIPOS

1 ASOCIACIÓN

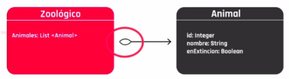
"TIENE" "CONOCE"

Relación estructural que crea una conexión entre clases. Es unilateral, va en un solo sentido, la clase A, conoce a la clase B, pero la B no sabe de la A. Se identifican rapidamente por el uso de las palabras **"tiene" o "conoce"**. Por ej, una persona vive en una dirección. La persona conoce la dirección, pero la direccion no sabe quien vive ahi.

2 AGREGACIÓN

"ES PARTE DE"

Por ejemplo, pensemos en la relación entre la selva y los animales. La selva tiene un conjunto de animales, los cuales son parte de la selva. Se indica con un "rombo" vacio partiendo de la clase que contiene a la otra.



3 COMPOSICIÓN

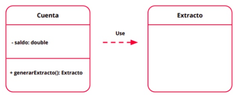
Cuando una clase está compuesta por otra clase. Se indica con un "rombo" relleno. La clase depende de las otras que la componen, **no existe por si misma**, a diferencia de la agregación, donde ambas clases pueden seguir existiendo independientemente.

4 RELACIÓN DE USO

Use

"USA UN"

tipo de asociación que como lo indica su nombre es una relación del tipo "usa un". La particularidad frente al otro tipo de asociación "tiene un" es que no hay una referencia de una clase a la otra, sino que en este caso, **la relación se da porque hay algún método que devuelve o recibe como parámetro una variable que es del tipo de la otra clase.**

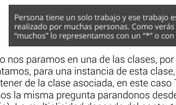


NAVEGACIÓN

Cuando una asociación lleva una flecha indica una **dirección de recorrido** (de navegación). Implica que es posible para un objeto en un extremo acceder al objeto del otro extremo porque el primero contiene referencias especificas a este último (al que apunta la flecha), no siendo cierto en el sentido contrario. Por ejemplo, una persona tiene un trabajo (un objeto de la clase trabajo como atributo)

CARDINALIDAD (O MULTIPLICIDAD)

La multiplicidad también llamada cardinalidad especifica el **número de instancias de una clase** que puede estar **relacionadas con una única instancia de una clase asociada**. La multiplicidad limita el número de objetos relacionados.



Persona tiene un solo trabajo y ese trabajo es realizado por muchas personas. Como verás el "muchos" lo representamos con un "*" o con una N.

Primero nos paramos en una de las clases, por ej Persona, y nos preguntamos, para una instancia de esta clase, cuantas instancias puede tener de la clase asociada, en este caso Trabajo. Luego hacemos la misma pregunta parandonos desde la Clase asociada (Trabajo). La multiplicidad depende del contexto.

MUCHOS A MUCHOS * a *

Una persona tiene muchas reuniones y en una misma reunión participan muchas personas.

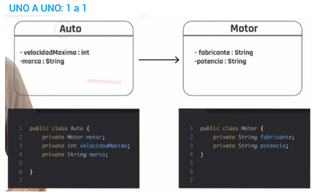
UNO A UNO 1 a 1

Una persona tiene una sola partida de nacimiento y una partida de nacimiento es de una única persona.

UNO A MUCHOS 1 a * * a1

una persona puede tener 2 pulmones y cada pulmón es de una única persona.

UNO A UNO: 1 a 1




```
public class Auto {
    private Motor motor;
    private int velocidadMaxima;
    private String marca;
}
```

```
public class Motor {
    private String fabricante;
    private String potencia;
}
```

UNO A MUCHOS: 1 a *

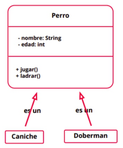

```
public class Auto {
    private Rueda[] ruedas;
    private int velocidadMaxima;
    private String marca;

}

public class Rueda {
    private int tamano;
}
```

5 RELACIÓN DE HERENCIA

Extends

"ES UN"

La herencia es uno de los pilares del paradigma orientado a objetos, también conocida como una relación del tipo **"es un".**



Podemos decir entonces que la **herencia** es un **ordenamiento entre clases** que define **una relación "es un"**. Entonces decimos que un caniche y un dóberman **es un** perro, porque tiene y hace todo lo que hace un perro.

Decimos que Caniche y Dóberman "heredan" el comportamiento de un perro, es decir, la clase Dóberman hereda de la clase Perro, todos sus atributos y responsabilidades favoreciendo la reutilización.

- La Herencia favorece la reutilización de código.
- Una clase que hereda de otra, suma a sus propios atributos y responsabilidades, los de la clase a la cual hereda.
- En JAVA NO está permitida la herencia múltiple.

GENERALIZACIÓN VS ESPECIALIZACIÓN

Nos encontramos en el modelo que estamos realizando un conjunto de clases, por ejemplo, Caniche y Doberman. Nos damos cuenta que ambas tienen algunos atributos y/o responsabilidades comunes. En dicho caso, **creamos una clase de la cual ambas heredarán** ambas y transportamos todos los atributos y/o responsabilidades que eran comunes a esta nueva clase que, en este ejemplo, llamaremos Perro. Este proceso mental de abstracción lo llamamos generalización.

Siguiendo el ejemplo de los perros, creamos una clase y le colocamos estos atributos y/o responsabilidades que únicamente tiene ese tipo de perro, en este ejemplo, esas clases son Doberman y Caniche. Este proceso mental de abstracción lo llamamos especialización.

Con la herencia aparece un modificador de visibilidad nuevo llamado **protegido,** que en los diagramas UML se especifica con el "#" . El método es privado para otras clases, pero público para las clases hijas. Se evita su uso por buena práctica.

HERENCIA EN JAVA

```
public class Persona {
    private String nombre;
    private String dni;

    public Persona(String nombre, String dni){
            this.nombre=nombre;
            this.dni=dni;
    }
```

```
public class Empleado extends Persona {
    private double sueldo;
    private double descuento;
    private String legajo;

    public Empleado(String nombre,String dni,
String legajo){
            super(nombre,dni);
            this.legajo=legajo;
            sueldo=30000;
    }
```

FIRMA DE UN MÉTODO

Definición completa de un método, es decir, su **nombre**, sus **parámetros** y sus **tipos** y el **orden** de aparición de dichos parámetros.
No podrán en una misma clase existir dos métodos con la misma firma, es decir, con el mismo nombre y cantidad de parámetros con sus respectivos tipos en el mismo orden. El valor que devuelve un método y los modificadores de visibilidad no forman parte de la firma.

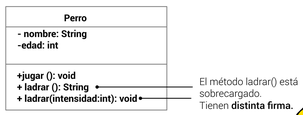
+ sumar(numero1: double, numero2: double): double

+ sumar(numero1: double, numero2: double, numero3: double): double //Distintos parámetros o cantidad de los mismos = firma distinta, son métodos distintos aunque comparten nombre.

MISMO NOMBRE DISTINTO COMPORTAMIENTO (FIRMA DIFERENTE)

SOBRECARGA DE UN MÉTODO

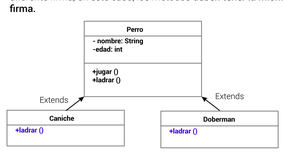
Es posible en el paradigma orientado a objetos tener en una misma clase dos o más métodos con el mismo nombre y cuyo comportamiento sea diferente. Esto es factible porque al momento de invocar dicho método se puede saber a cuál de todos invocar siempre que su firma sea diferente.



SOBRESCRITURA DE UN MÉTODO

Para poder sobreescribir métodos **necesitamos una relación de herencia**, ya que lo que vamos a sobreescribir es un método de la superclase para que se comporte diferente en la subclase.
A diferencia de la sobrecarga donde los métodos tienen que tener diferente firma, en este caso, los métodos **deben tener la misma firma.**

MISMA FIRMA



Al escribir en las subclases el método **ladrar()**, decimos que el mismo está subscrito y se comporta diferente. **Misma firma.**

En JAVA usamos la anotación **@Override,** para indicar que el método debajo está siendo sobrescrito.

```
public class Vendedor extends Empleado{
      private int comision;
      private double importeVentas;

      @Override
      public double calcularSueldo(){
            return sueldo+descuentos + importeVentas/100*comision;
            }
      @Override
      public double calcularSueldo(double premio){
            return sueldo+descuentos + premio+ importeVentas/100*comision;
            }
}
```

LA CLASE OBJECT

Todas las clases que creamos en Java derivan de la clase Object, aunque no esté escrito explícitamente. Por eso, cuando creamos una clase nueva, tiene ciertos **métodos que hereda.** De estos métodos vamos a tomar algunos y **para que funcionen correctamente, debemos sobreescribirlos.**

.toString()

Toda clase hereda de Object el método toString(), es decir, si no lo implementamos, los objetos que instanciamos tendrán este método.

```
public class Empleado{
      private   String nombre;
      private   String legajo;
      protected double sueldo;
      protected double descuentos;

      @Override
      public String toString(){
            return "Nombre: " + nombre + "\n" +
                  "Legajo: " + legajo;
      }
}
```

.hashCode()

Cuando se utiliza este método nos devuelve un número único que identifica al objeto, es decir, si tengo dos objetos de la misma clase, el hashCode() generaría un número distinto para cada uno y ese número me va a servir para identificarlo.

```
public class Empleado{
      private   String nombre;
      private   String legajo;
      protected double sueldo;
      protected double descuentos;

      @Override
      public int hashCode(){
            int hash=31;
            hash= hash* nombre.hashCode();
            hash= hash* legajo.hashCode();
            return hash;}
}
```

Para generar un número único se trabaja con números **primos.** Puede ser cualquier número primo, en este caso se usó el 31. Como nombre y legajo son strings, o sea, también son objetos, tienen su propio hashCode(). Multiplicamos todos los números y obtenemos el hashCode del objeto. En una string, el hashCode se genera a partir de los caracteres. Por ejemplo, el número de legajo es siempre distinto.

.equals(Object o)

Cuando creamos un objeto o instancia, lo que tenemos es una referencia a la memoria (RAM), es decir, no se almacenan datos directamente en la variable de tipo objeto, solo la referencia al lugar donde están los valores de los atributos del objeto. Es por eso que **no podemos utilizar el operador "==" para comparar la igualdad entre dos objetos.**

Antes de comenzar a trabajar con el equals, debemos pensar **qué quiere decir que dos objetos son iguales**, por ejemplo, si comparamos dos empleados, podríamos definir que son iguales si sus legajos son iguales. Ejemplo con una clase llamada Empleado:

Para comenzar a escribir nuestro equals, debemos considerar que el parámetro que me está llegando es un Object, no dice que sea un Empleado, entonces, **lo primero a verificar es si realmente es un Empleado**, si no lo fuera ya podemos decir que no son iguales.
Vamos a ver dos formas de comprobarlo:
1 instanceOf: me permite comparar dos instancias:

```
    @Override
    public boolean equals(Object o){
        if (o==null)
            return false;
        if (!(o instanceof Empleado))
            return false;
        else{

        }
    }
```

El operador instanceOf, nos permite comprobar si el object o (el que recibimos como parametro) es una instancia de Empleado, la clase que estamos comparando.

2 .getclass(): nos permite comparar la clase a la que pertenecen los objetos.

```
    @Override
    public boolean equals(Object o){
        if (o==null)
            return false;
        if (this.getClass()==o.getClass())
            return false;
        else{

        }
    }
```

Comprobamos si la clase de la instancia, es la misma clase del objeto recibido como parámetro.

CASTING

Definición: El casteo (casting) es un procedimiento para transformar una variable primitiva de un tipo a otro, o transformar un objeto de una clase a otra clase siempre y cuando haya una relación de herencia entre ambas.
Ahora nos restaría comprobar la igualdad (tener el mismo legajo). Para hacer esta comprobación, vamos a necesitar pedirle a "o", el legajo para compararlo con el de la instancia. Pero "o" es un Object, o sea, no "sabe" que tiene legajo.

```
    @Override
    public boolean equals(Object o){
        if (o==null)
            return false;
        if (!(o instanceof Empleado))
            return false;
        else{
            Empleado empleadoAux=(Empleado)o;
            return
this.getLegajo().equals(empleadoAux.getLegajo());
        }
    }
```

Casteamos "o" y los asignamos a un objeto de clase Empleado. Con el casting lo que logramos es transformar, para poder asignarlo a un objeto de tipo Empleado y de esta forma usar sus métodos.

CLASE ABSTRACTA

<abstract>>

Existirán ocasiones en las que necesitaremos modelar clases, pero que las mismas estarán incompletas. Es decir, que se terminan de implementar en las subclases porque es en ellas donde tienen la implementación específica. Estas las llamamos clases abstractas.

Las clases abstractas son aquellas que por sí mismas no se pueden identificar con algo "concreto" (no existen como tal en el mundo real), pero sí poseen determinadas características que son comunes en otras clases que heredarán de esta. Estas clases abstractas **nos permite declarar métodos, pero que estos no estén implementados.** Estos métodos denominados tambien **abstractos,** obligarán a las subclases a sobreescribirlos para darles una implementación.
No olvidemos que no podremos instanciar objetos de una clase abstracta.

Las clases abstractas en el **diagrama UML** las representaremos ya sea indicando su **nombre en forma cursiva** o **explicitando arriba de su nombre que es abstracta <<abstract>>**
Los métodos abstractos los vamos a especificar en los diagramas UML como se muestra a continuación:

Si bien una clase abstracta puede tener uno o varios métodos abstractos, no es obligatorio que los tenga.

EJEMPLO

A continuación, en el siguiente modelo UML, se muestra a una **Compañía** que tiene varios tipos de empleados y cuyo cálculo de sueldo es diferente en cada caso.

EN JAVA

Definimos a las clases abstractas y el comportamiento en abstracto con la palabra clave **"abstract".** Cómo el comportamiento es abstracto (solo decimos qué hacer), los **métodos abstractos** no tienen código asociado, **no tienen "cuerpo".**

abstract class
Especifica que la clase Perro es abstracta.

```
public abstract class Perro{

    public abstract string ladrar();
}
```

Especifica que el método ladrar() es abstracto.

Sobrescritura de métodos: Por ejemplo, Si Caniche quiere SER UN Perro, entonces debe respetar el contrato de los Perros: debe implementar un método que se llame ladrar, que devuelva un String y que no reciba parámetros. En pocas palabras, debe sobrescribir todos los métodos abstractos definidos en Perro.

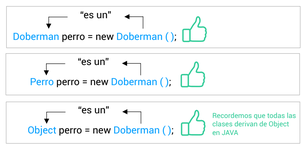
```
public class Caniche extends Perro{

    public string ladrar() {
        return "ladro como un caniche guau...";
    }
}
```

VINCULACIÓN DINÁMICA

Se declara una referencia de un determinado tipo: Es una variable. Todavía no se ha creado ningún objeto. Solo se indica que se reserve lugar **para un objeto** que, eventualmente, apuntará un objeto Doberman (un objeto de tipo Doberman).

Se crea un objeto: Se indica que se reserve espacio para guardar un objeto Doberman en la memoria.

La vinculación dinámica de una referencia funciona es igual que un enchufe. En un enchufe se puede conectar diferentes cosas: un TV, una heladera, una notebook.
Veremos que en una referencia podremos apuntar a diferentes tipos de objetos.
En el ejemplo anterior, la referencia y el objeto referenciado son del mismo tipo: Doberman. Sin embargo es posible que sean de distinto tipo. En JAVA, el objeto debe ser de una clase que tenga una relación de **herencia** ("es un") con respecto a la referencia.



POLIMORFISMO

Es la capacidad de un mismo objeto de comportarse como otro. En otras palabras, es la capacidad de un objeto de funcionar de diversas formas. Veamos con lo ejemplos anteriores:

```
Perro p;

p = new Doberman();
p.ladrar();

p = new Caniche();
p.ladrar();
```

La referencia **p** se puede comportar y ladrar como un Doberman.

Y la misma referencia **p** al vincularse dinamicamente (dynamic binding) con un **Caniche,** se comporta de forma diferente y ladra como un Caniche.

Supongamos que Doberman tiene un método llamado morderComoDoberman(), pero la referencia o sea la variable es del tipo Perro. Para forzar a un perro a que sea un Doberman utilizamos el **casteo**. De esta manera podremos invocar los métodos propios de Doberman.

```
Perro perro = new Doberman();
      perro.ladrar();

((Doberman)perro).morderComoDoberman()
```

Casteo

INTERFACE

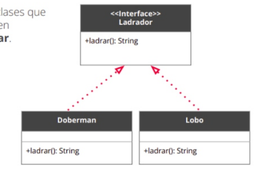
Implements

Las interfaces representan distintos comportamientos que van a tener las clases concretas. Las interfaces no nos permiten definir instancias, sólo nos dan características compartidas por las clases, un contrato que deben cumplir.

En **UML** las interfaces NO tienen atributos, y se distinguen porque llevan su nombre entre símbolos **<< >>.** Son mas faciles de indentificar cuando en el nombre se usan sufijos able/ible.

Son también relaciones del tipo **"es un"**(permiten por lo tanto polimorfismo) , muy similares a las clases abstractas: **se definen con la palabra clave "interface"** en vez de "class". **Todos sus métodos son abstractos**, por lo cual, no es necesaria la palabra "abstract" y, al igual que las clases abstractas, los métodos no definen un cuerpo.
Una interface establece un contrato. Toda clase que implemente una interface está obligada a implementar todos los métodos de esa interface.

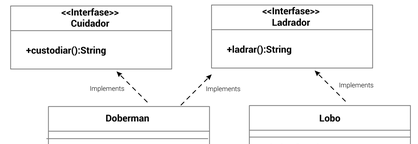
Por ejemplo, todas aquellas clases que implementen **Ladrador** deben implementar el método **ladrar**.



Cuando heredamos de una clase sumamos atributos y comportamientos de la clase padre, mientras que **cuando implementamos una interface solo obligamos a la clase que la implementa a sobrescribir**, es decir, implementar, los métodos de la misma.
Una clase puede solo heredar de una clase, mientras que **puede implementar múltiples interfaces.**

Lo que permiten las interfaces es **independizarse de una jerarquía,** permiten agregar comportamiento a una clase que no se obtenga desde un nivel superior en la jerarquía, se "enchufa" lateralmente a la jerarquía. Incluso podríamos hasta mezclar ambos mecanismos.

EJEMPLO



Interfaz Cuidador.

```
public interface Cuidador{

      public void String custodiar( );
}
```

Interfaz Ladrador.

```
public interface Ladrador{

      public void String ladrar( );
}
```

Clase Doberman:

```
public class Doberman implements Cuidador, Ladrador{

  public void String custodiar(){
     return "estoy atento custodiando la casa";
  }


  public void String ladrar(){
     return "Guauu! Guauu!";
  }
}
```

```
/*Dada una referencia ladrador del tipo Ladrador (Ladrador ladrador)*/

  ladrador = new Doberman(); //ladrador es ahora del tipo Doberman()
  System.out.println(ladrador.ladrar()); //Polimorfismo
```

Clase Lobo:

```
public class Lobo implements Ladrador{


  public void String ladrar(){
     return "guauu! los lobos también ladramos";
  }
}
```

```
/*Dada una referencia ladrador del tipo Ladrador (Ladrador ladrador)*/

  ladrador = new Lobo(); //ladrador es ahora del tipo Lobo()
  System.out.println(ladrador.ladrar()); //Polimorfismo
```

INTERFACE COMPARABLE

No necesitamos crear una interface para comparar objetos porque Java tiene la suya, es la **interface Comparable** y es necesaria utilizarla en otras circunstancias para comparar objetos, por ejemplo, para ordenarlos en las colecciones. El método que obliga a implementar la interface comparable de JAVA es **compareTo.** Para usar la interface Comparable, hay que importar el paquete java.lang.

El método **compareTo** debe devolver:
si son iguales: **0**
si es mayor: un numero **mayor a 0**
si es menor: un numero **menor a 0.**

```
import java.lang.*;


public class Pimiento implements Comparable{

        private String tipo;
        private String color;
        private double tamano;
        private double peso;


public Pimiento(){
}


public int compareTo(Object obj){

        Pimiento p2 = (Pimiento) obj;
        int respuesta = 0;


        if(this.getPeso() > p2.getPeso())
                respuesta = 1;

        if(this.getPeso() < p2.getPeso())
                respuesta = -1;

        return respuesta;
```

```
public class Prueba{


        public void main(String args[]){

        Pimiento p1 = new Pimiento();
                p1.setPeso((20K);
                p1.setColor("rojo");
        Pimiento p2 = new Pimiento();
                p2.setColor("verde");
                p2.setPeso(150);

        if(p1.compareTo(p2) > 0){
                System.out.println("Pimiento amarillo es mayor al rojo");
        }else if(p1.compareTo(p2) < 0){
                System.out.println("Pimiento rojo es mayor al amarillo");
        }else{
                System.out.println("Pimiento rojo es igual al amarillo");
        }

        }
}
```

AYUDA MEMORIA RÁPIDO

RELACIONES

NOMBRE	FLECHA EN UML	TIPO
ASOCIACIÓN	→	"TIENE" "CONOCE"
AGREGACIÓN	◇→	"ES PARTE DE" (relación débil)
COMPOSICIÓN	◆→	"ES PARTE DE" (relación fuerte)
DE USO	Use ⇢	"USA UN"
HERENCIA	Extends →	"ES UN"
IMPLEMENTACIÓN DE INTERFACE	Implements ⇢	"ES UN" (implementa)

HERENCIA EN JAVA

```
public class Empleado extends Persona{

}
```

CLASES

	CONCRETAS	ABSTRACTAS
EN UML	Nombre. -Atributos privados +Métodos públicos	<<abstract>> Nombre. -Atributos privados +Métodos públicos Puede tener métodos abstractos ej: +**abstract** ladrar ( )

EN JAVA	public **class** Perro{
	private String nombre;
	
	public void ladrar(){
	System.out.println ("guau");
	}
	}
		public **abstract class** Perro{
		private String nombre;
		
		public abstract String ladrar();
		}

INTERFACE

EN UML	**<<Interface>>**Nombre **no** tiene atributos **+**métodos públicos, son por defecto todos abstractos. (no hace falta poner "abstract")

EN JAVA	Interfaz Cuidador:
	
	public interface Cuidador{
	
	public void custodiar( );
	}
	
	Clase Doberman:
	
	public class Doberman **implements** Cuidador{
	
	public void custodiar( ){
	System.out.println ("estoy custodiando");
	}
	}