

POO. Clases y objetos. Herencia. Interfaces.

1. FUNCIONES, MÉTODOS.

Las **funciones** son un conjunto de líneas de código (instrucciones), encapsulados en un bloque, usualmente reciben parámetros, cuyos valores utilizan para efectuar operaciones y adicionalmente retornan un valor. En java las funciones se llaman **métodos**. Estos métodos pueden usarse referidos a un objeto, es decir, necesitamos crear un objeto para usar el método. O hay métodos que no necesitan de la construcción de un objeto para su uso, estos métodos se llaman ESTÁTICOS y necesitan ser definidos con la palabra reservada *static*.

Por ejemplo, cuando usamos la clase main usamos static porque ese método no va asociado a ningún objeto.

```
package funciones;  
public class Funciones {  
  
    public static void main(String[] args) {  
        // TODO code application logic here  
    }  
  
}
```

Sin embargo en el siguiente ejemplo necesitaremos construir un objeto de tipo Triangulo para poder calcular su área y, por tanto, no usamos **static** con el método:

```
class Triangulo {  
    private long base;  
    private long altura;  
  
    public Triangulo(long base, long altura) {  
        this.base = base;  
        this.altura = altura;  
    }  
  
    public long area() {  
        return (base*altura)/2;  
    }  
}
```

Constructor

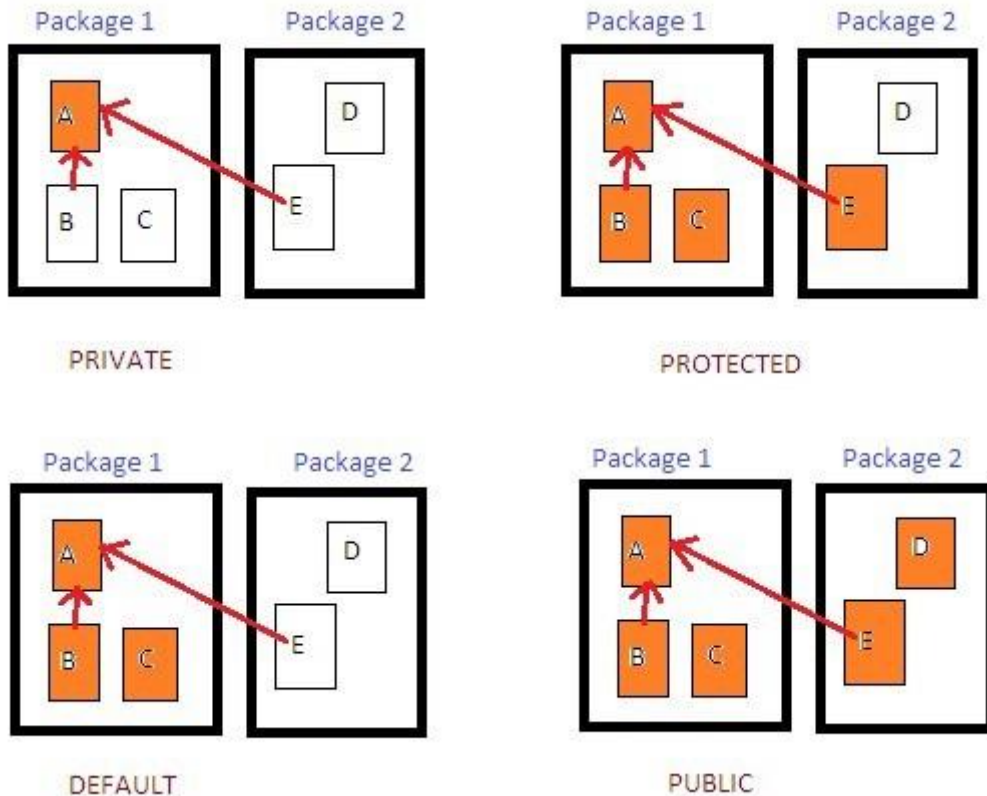
1. DEFINICIÓN DE MÉTODOS.

Las funciones se definirán de la siguiente manera:

```
modificadores (static) tipo_devuelto nombre_función (tipo  
var1, ... , tipo var n){  
  
    return expresión (del tipo devuelto);  
}
```

Donde:

- **Modificadores:** Los modificadores definen las características de visibilidad de la función así:
 - **private:** únicamente la clase puede acceder a la propiedad o método.
 - **package private** (valor por defecto si no se indica ninguno): solo las clases en el mismo paquete pueden acceder a la propiedad o método.
 - **protected:** las clases del mismo paquete y que heredan de la clase pueden acceder a la propiedad o método.
 - **public:** la propiedad o método es accesible desde cualquier método de otra clase.



- static:** Se usa para indicar que el método pertenece a la clase y no al objeto. Esto quiere decir que NO tendremos que crear un objeto para usar ese método, es más, no se puede usar referidos a objetos.
- tipo_devuelto:** Si la función devuelve algo, hay que especificar el tipo del valor devuelto (int, boolean, String, array, etc) Si la función no devuelve nada se pone **void**.
- nombre_función:** En java se pueden escribir funciones con el mismo nombre mientras tenga diferente número de parámetros, diferente tipo o ambas.
- return:** en el caso de las funciones que devuelven algo se pone return antes de la expresión o variable a devolver. El tipo de este debe coincidir con el tipo_devuelto. NOTA: Cualquier instrucción que se encuentre después de la ejecución de return NO será ejecutada. Cuando es void no

hace falta usar return pero
acaba la función.

se puede usar para definir dónde

- **argumentos:**

- Una función, un método o un procedimiento pueden tener una cantidad cualquier de parámetros. Aunque habitualmente no suelen tener más de 4 o 5.
- Si una función tiene más de un parámetro cada uno de ellos debe ir separado por una coma.
- Los argumentos de una función también tienen un tipo y un nombre que los identifica. El tipo del argumento puede ser cualquiera y no tiene relación con el tipo del método.
- Al recibir un argumento nada nos obliga a hacer uso de éste al interior del método, sin embargo para qué recibirlo si no lo vamos a usar.

Veamos ejemplo de todo esto:

a. Función que recibe argumentos y devuelve un valor:

```
public static int multiplicar(int valor1, int valor2){  
    return (valor1*valor2);  
}
```

b. Función que recibe argumentos y no devuelve nada:

```
public static void imprime_multi(int valor1, int valor2){  
    System.out.println (valor1*valor2);  
}
```

c. Función que NO recibe argumentos y devuelve un valor:

```
public static int leerymultiplicar() throws IOException{
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String entrada;

    System.out.println ("Introduzca el valor 1");
    entrada = br.readLine();
    int valor1 = Integer.parseInt(entrada);

    System.out.println ("Introduzca el valor 2");
    entrada = br.readLine();
    int valor2 = Integer.parseInt(entrada);

    return (valor1*valor2);
}
```

d. Función que NO recibe argumentos y NO devuelve nada

```
public static void mensaje_error(){
    System.out.println ("se ha producido un error");
}
```

Las funciones vienen definidas por su nombre, el tipo de dato que devuelven y los parámetros que se le pasan. Así podremos tener funciones con el mismo nombre pero que se le pasen diferente número de parámetros y, por tanto, sean diferentes.

2. LLAMADA A FUNCIONES.

Para llamar a una función (usarla), solo tienes que escribir su nombre seguido de los paréntesis de apertura y cierre en la línea donde quieras ejecutarlo. En los paréntesis irá los parámetros que se le pasan a la función (en caso de que sean necesarios)

```
nombre_función ( var1, ... , var n);
```

En el caso de que la función devuelva algo y queramos usar ese valor más adelante, habrá que almacenar el valor devuelto en una variable:

```
variable = nombre_función ( var1, ... , var n);
```

2. CLASES Y

OBJETOS.

Un objeto es: “ un elemento de software que intenta representar un objeto del mundo real. De esta forma un objeto tendrá sus propiedades y acciones a realizar con el objeto. Estas propiedades y acciones están encapsuladas dentro del objeto, cumpliendo así los principios de encapsulamiento.

Un objeto tiene su estado (o estados) y su comportamiento. Esto se modela mediante propiedades (o variables) y métodos.”

Recordemos a su vez que para crear un objeto tenemos que partir de una clase a la que le daremos valores. Siendo una clase una combinación específica de atributos y métodos. Así, una clase define los atributos que componen ese tipo de objetos y los métodos que pueden emplearse para trabajar con esos objetos.

Por tanto, una clase es un plan o prototipo que define las variables y los métodos o funciones comunes a todos los objetos de un cierto tipo, y un objeto es un espécimen de una clase.

Para definir una clase usaremos el siguiente código:

```
[modificador] class Nombre_Clase [extends NombreSuperClase]
[implements NombreInterfacel, NombreInterface2, ... ] {

    //atributos de la clase (0 ó más atributos)

    [modificador] tipo nombreAtributo;

    //métodos de la clase (0 ó más métodos)
    [modificador] tipoDevuelto nombreMetodo([lista
parámetros]) [throws listaExcepciones]{

        // instrucciones del método

        [return valor;]

    }
}
```

Donde lo que se haya entre [] es opcional y los modificadores son los vistos anteriormente para los métodos.

Apuntar que **extends** es la palabra reservada para indicar la herencia en Java, e **implements** es la palabra reservada para indicar que la clase implementa el o los interfaces que se indican separados por comas.

Un ejemplo de esto es el visto en

la programación estructurada:

```
class Triangulo {  
    private long base;  
    private long altura;  
  
    public Triangulo(long base, long altura) {  
        this.base = base;  
        this.altura = altura;  
    }  
  
    public long area() {  
        return (base*altura)/2;  
    }  
}
```

NOTA: sobre **This**. This es una palabra reservada en java que quiere decir que hace referencia al objeto actual de la clase, se usa cuando los parámetros de la clase tiene el mismo nombre que los del método para así distinguirlos.

Para crear un objeto de esta clase haríamos:

```
Triangulo t1 = new Triangulo(2.0,3.0);
```

2.1 CONSTRUCTOR.

Las clases tienen unos métodos “especiales” que permiten crear un objeto dándole valores concretos a los atributos generales de las clases. Estos métodos se llaman constructores porque construyen el objeto.

Java proporciona automáticamente un constructor predeterminado. En este caso, las variables no inicializadas tienen sus valores predeterminados, que son **cero, null y false**, para tipos numéricos, tipos de referencia y booleanos, respectivamente. Una vez que defines tu propio constructor, el constructor predeterminado ya no se usa.

El constructor se define de la siguiente forma:

```
public nombre (tipovariable variable1, ...tipovariable
variablen){
    this.variable1 =variable1;
    ...
    this.variablen=variablen;
}
```

Los constructores tienen que tener el mismo nombre que la clase y no se indica el tipo del dato que devuelven. (como no devuelven nada sería void)

Como hemos visto en java podemos tener funciones con el mismo nombre pero que harán cosas diferentes según sean definidas. Con los constructores pasa lo mismo, una clase podrá tener varios constructores.

Veamos un ejemplo:

```
public class Fecha {
    // Atributos o variables miembro
    private int dia;
    private int mes;
    private int anho;

    /**
     * Constructor 1
     * Asigna los valores 1, 1 y 2000 a los atributos
     * dia, mes y anho respectivamente
     */
    public Fecha () {
        this.dia = 1;
        this.mes = 1;
        this.anho = 2000;
    }

    /**
     * Constructor 2
     * @param ndia el dia del mes a almacenar
     * @param nmes el mes del anho a almacenar
     * @param nanho el anho a almacenar
     */
    public Fecha(int dia, int mes, int anho) {
        this.dia = dia;
        this.mes = mes;
        this.anho = anho;
    }
}
```

}

2.2 CÓMO USAR LOS ATRIBUTOS DE LOS OBJETOS.

Como hemos visto en el apartado anterior cuando definimos una clase tenemos que definir el modificador de sus atributos. Según sean estos así podremos usarlos luego al crear el objeto. Vamos a ver dos opciones:

- A. Definir los atributos como públicos (para el paquete o para todos) y así podemos usarlo a posteriori directamente poniendo **nombreObjeto.atributo**, por ejemplo:

```
//definimos la clase en el mismo paquete que la clase principal
class Conductor {

    //definimos los atributos como públicos para el paquete
    String nombre;
    int edad;
    String tipo_carnet;

    //definimos el constructor
    public Conductor (String nombre, int edad, String
tipo_carnet) {
        this.nombre = nombre;
        this.edad = edad;
        this.tipo_carnet= tipo_carnet;
    }
}
```

```
//definimos la clase principal

class Clases {
    public static void main(String[] args) {

        //construimos el objeto
        Conductor conductorA = new Conductor ("Juan", 16,
"AM");
    }
}
```

```
//Imprimimos sus atributos
System.out.println ("El conductor se llama "+
conductorA.nombre + " , tiene " + conductorA.edad + " años
y tiene el carnet de conducir tipo " + conductorA.nombre +
tipo_carnet);
}
}
```

Como vemos en este ejemplo hemos usado los atributos directamente. Pero este método puede generar problemas de “seguridad” al estar los atributos tan expuestos. Lo ideal sería usar la forma B que vamos a definir a continuación.

- B. Definir los atributos como privados (sólo se podrán usar en la propia clase) y definir **métodos públicos llamados get y set**, para trabajar con ellos.

Los métodos **get** y **set**, son simples métodos que usamos en las clases para mostrar (**get**) o modificar (**set**) el valor de un atributo. El nombre del método siempre será **get** o **set** y a continuación el nombre del atributo, su modificador siempre es **public** ya que queremos mostrar o modificar desde fuera la clase. Por ejemplo, **getNombre** o **setNombre**.

La sintaxis de estos métodos son:

```
public tipo_dato_atributo getAtributo () {
    return atributo;
}
```

Al método get no se le pasa nada pero tendrá que devolver el dato por lo que tendrá que estar definido con ese tipo de dato.

Al método set, al contrario, se le tendrá que pasar el valor que queremos asignar al atributo pero no devolverá nada, por lo que se definirá como void.

```
public void setAtributo (tipo_dato_atributo variable) {
    this.atributo = variable;
}
```

Veamos esto con un ejemplo:

```
//definimos la clase en el mismo paquete que la clase principal
```

```
class Conductor {
```

```
    //definimos los atributos como privados, solo pueden ser  
    //usados en su propia clase.
```

```
    private String nombre;
```

```
    private String tipo_carnet;
```

```
    //definimos el constructor
```

```
    public conductor (String nombre,String tipo_carnet) {
```

```
        this.nombre = nombre;
```

```
        this.tipo_carnet= tipo_carnet;
```

```
    }
```

```
    //definimos los métodos get
```

```
    public String getNombre () {
```

```
        return nombre;
```

```
    }
```

```
    public String getCarnet () {
```

```
        return tipo_carnet;
```

```
    }
```

```
    //definimos los métodos set
```

```
    public void setNombre(String Nombre){
```

```
        this.nombre = nombre;
```

```
    }
```

```
    public void setCarnet(String tipo_carnet){
```

```
        this.tipo_carnet = tipo_carnet;
```

```
    }
```

```
}
```

```
//definimos la clase principal
```

```
class Clases {
```

```
    public static void main(String[] args) {
```

```
//construimos el objeto
Conductor conductorA = new Conductor ("Juan", "AM");

//Cambiamos el valor de uno de ellos.
conductorA.setCarnet("B");

//Imprimimos sus atributos, usando métodos get
System.out.println ("El conductor se llama " +
conductorA.getNombre() + " y tiene el carnet de
conducir tipo " + conductorA.getCarnet());
}
}
```

Este método, como ya hemos visto es más “seguro” pues protege los atributos pero hace necesario crear más métodos para poder usarlos luego.

3. LIBRERÍAS DE OBJETOS.

Una librería, es un código reutilizable que se puede intercambiar y reutilizar en diferentes programas para diferentes propósitos. Una librería de objetos o paquete es una colección lógica de clase.

Indicamos que una clase pertenece a un paquete a través de la palabra reservada **package** al comienzo de la clase.

```
package nombre_paquete;
```

Si miramos en Netbeans la carpeta de nuestro proyecto, veremos que hay una carpeta src con los distintos paquetes del proyecto. Cada vez que se crea un nuevo proyecto con NetBeans se propone la definición de un nuevo paquete donde se alojarán los módulos de código. En proyectos complejos, no obstante, puede ser necesaria la creación de paquetes adicionales.

Un paquete puede contener, además de definiciones de tipos como las clases e interfaces, otros paquetes, dando lugar a estructuras jerárquicas (se verá más adelante)

3.1 Importar paquetes.

Para poder usar un paquete en nuestro proyecto bastará con importarlo usando la palabra reservada **import**, con ella podremos importar un paquete completo

```
import nombrepaquete.*;
```

o solo el elemento que vamos a utilizar:

```
import nombrepaquete.clase;
```

Por ejemplo:

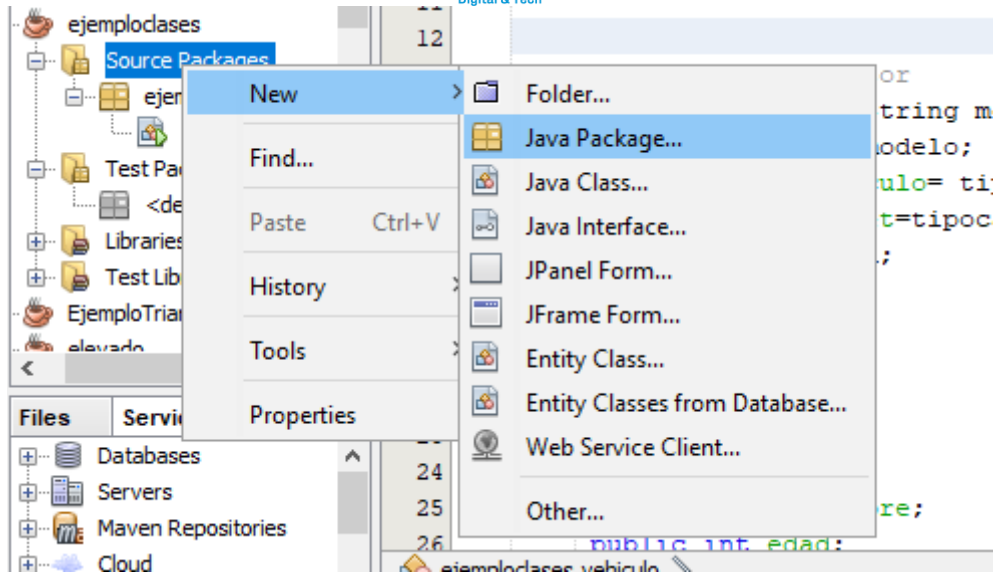
```
import System.io.*;  
import java.lang.Math;
```

Gracias a esto podemos tener clases con el mismo nombre en paquetes distintos.

3.2 Crear paquetes.

Si queremos crear un nuevo paquete para un proyecto con NeatBeans basta con seguir los siguientes pasos:

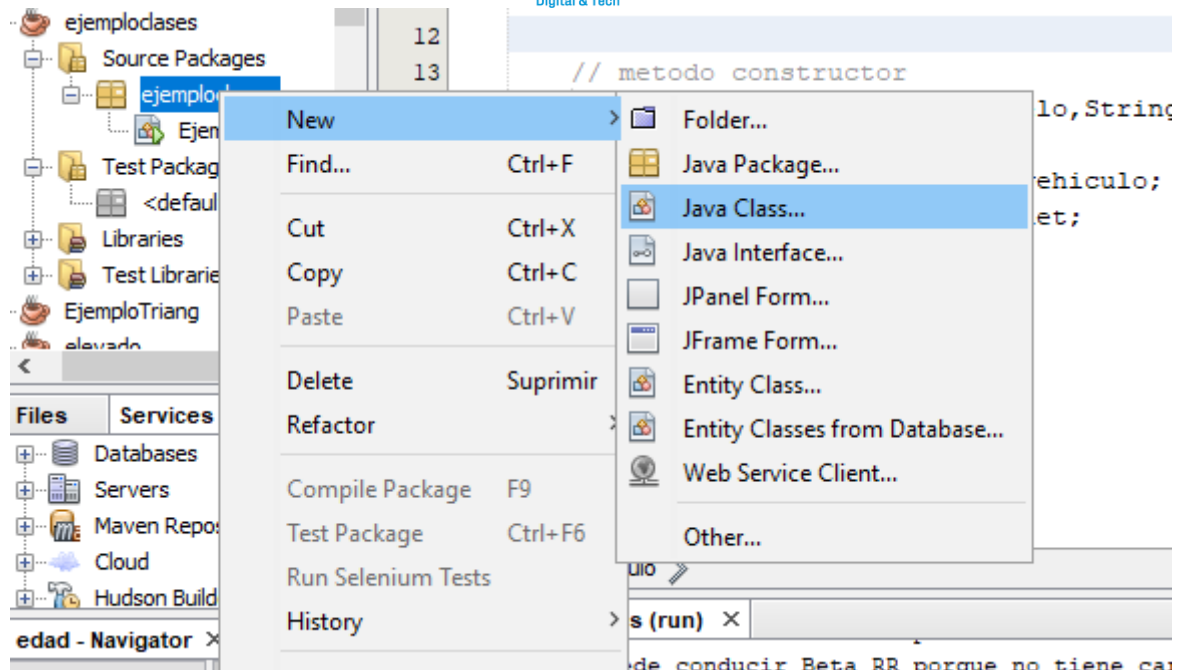
- Ir al explorador de proyectos.
- Pulsar en el botón derecho del proyecto al que le queremos agregar la librería.
- Clic en New, Java Package.
- Escribir el nombre del paquete y clicar en Finish.



Dentro de este paquete podremos crear distintas clases teniendo en cuenta que para visibilidad pública cada clase deberá ir en su archivo .java cuyo nombre sea el mismo de la clase. Por el contrario, las clases con visibilidad en el paquete pueden estar definidas en un archivo .java propio, cuyo nombre coincida con el de la clase o dentro de otro archivo .java perteneciente a otra clase.

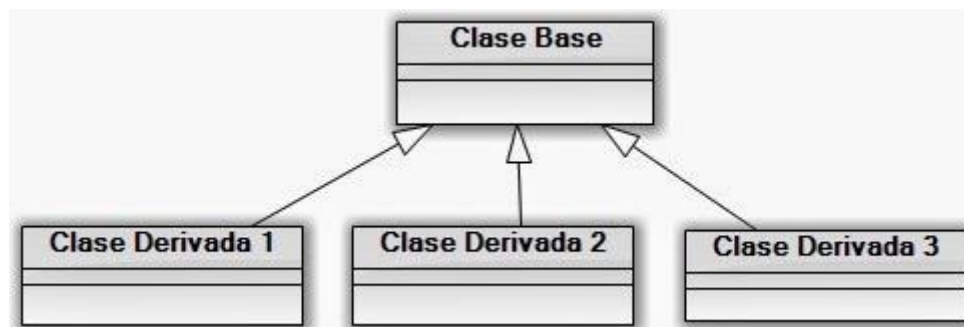
Para crear el archivo de la clase dentro del paquete hacemos:

- Ir al explorador de proyectos.
- Pulsar en el botón derecho del paquete al que le queremos agregar la clase.
- Clic en New, Java Class.
- Escribir el nombre de la clase y clicar en Finish.



4. HERENCIA.

Podemos definir la herencia como la capacidad de crear clases que adquieren de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.



Otra forma de verlo es que podemos sacar factor común en varias clases para optimizar y minimizar el código. Por ejemplo, imaginemos que queremos definir las personas que componen un instituto tendremos las siguientes clases con los siguientes atributos:

- Profesor:
 - Nombre
 - Apellidos
 - Contacto

- Asignaturas
- Turno

- Alumnos:
 - Nombre
 - Apellidos
 - Contacto
 - Curso

- Otros miembros:
 - Nombre
 - Apellidos
 - Contacto
 - Horario
 - Función

Viendo estas tres clases podemos sacar que las tres tienen en común los atributos Nombre, Apellidos y Contacto. Con ellos podemos formar una clase padre o superclase:

```
public class Persona {  
  
    //Los atributos se definirán como protected ya que si  
    //fueran private no lo podrían usar las subclases.  
  
    protected String nombre;  
    protected String apellidos;  
    protected String contacto;  
  
    public Persona (String nombre, String apellidos, String  
    contacto) {  
  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.contacto = contacto;  
    }  
}
```

Luego tendremos 3 subclases
nuevos atributos:

derivadas de esta y que añadirán

```
public class Profesor extends Persona{

    private String asignaturas;
    private String turno;

    public Profesor(String nombre, String apellidos, String
        contacto, String asignaturas, String turno) {

        super (nombre, apellidos, contacto);
        this.asignaturas = asignaturas;
        this.turno = turno;
    }
}

public class Alumno extends Persona{

    private String curso;

    public Alumno (String nombre, String apellidos, String
        contacto, String curso) {

        super (nombre, apellidos, contacto);
        this.curso = curso;
    }
}

public class Miembro extends Persona{

    private String funcion;
    private String horario;

    public Miembros(String nombre, String apellidos, String
        contacto, String funcipn, String horario) {

        super (nombre, apellidos, contacto);
        this.funcion = funcion;
        this.horario = horario;
    }
}
```

En los ejemplos se han remarcado 3 palabras claves:

- **extends**: Esta palabra reservada, indica a la clase hija cual va a ser su clase padre.
- **protected**: sirve para indicar un tipo de visibilidad de los atributos y métodos de la clase padre y significa que cuando un atributo es 'protected' o protegido, solo es visible ese atributo o método desde una de las clases hijas y no desde otra clase.
- **super**: sirve para llamar al constructor de la clase padre y para redefinir métodos.

Lo mismo que ocurre con los atributos pasa con los métodos.

4.1 REDEFINICIÓN DE MÉTODOS.

Como se ha visto las clases hijas o subclases pueden usar los métodos de su clase padre. Si una clase usa un método del padre lo hace como si fuera suyo propio. Pero existe la posibilidad de redefinir un método heredado de manera que se le cambie su funcionamiento. Para hacer esto necesitaremos definir el método con el mismo nombre y parámetros. Veamos esto con un ejemplo:

```
class Persona {  
    protected String nombre;  
    protected String apellidos;  
    protected String contacto;  
    public persona (String nombre, String apellidos, String  
    contacto) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.contacto = contacto;  
    }  
  
        public void imprimir () {  
            System.out.println(nombre + apellidos);  
        }  
}  
  
class Profesor extends Persona{
```

```
private          String          asignaturas;
private String turno;

    public Profesor(String nombre, String apellidos, String
        contacto, String asignaturas, String turno) {
        super (nombre, apellidos, contacto);
    this.asignaturas = asignaturas;
    this.turno = turno;
}

    public void imprimir (){
        super.imprimir();
        System.out.println(asignaturas );
    }
}
```

Cuando se redefine un método se puede cambiar su nivel de visibilidad siempre y cuando sea a menos restrictiva.

4.2 FINAL.

Si queremos evitar que una clase tenga clases derivadas debe declararse con el modificador **final** delante de class:

```
public final class A
```

Esto la convierte en clase final. Una clase final no se puede heredar. Si intentamos crear una clase derivada de A se producirá un error de compilación:

```
public class B extends A //extends A producirá un error de compilación
```

Si usamos **final** con un método este no podrá redefinirse.

4.3 CLASES ABSTRACTAS.

Las clases abstractas funcionan como una clase que declara la existencia de métodos pero no su implementación, es decir, nombran al método pero no lo que hace. Cuando tenemos un método abstracto en una superclase la subclase está obligada a redefinir ese método. Una clase abstracta puede contener métodos

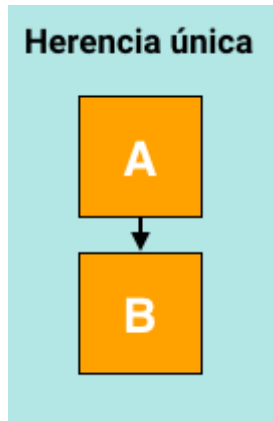
no abstractos pero al menos uno
Veamos un ejemplo:

de los métodos sí debe serlo.

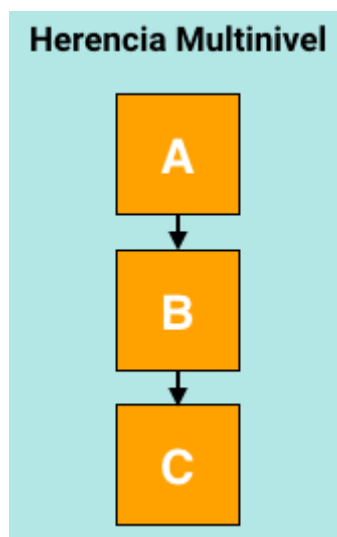
```
public abstract class Producto {  
    protected int precio;  
  
    public Producto(int precio) {  
        this.precio = precio;  
    }  
  
    public int getPrecio(){  
        return this.precio;  
    }  
  
    public abstract String getName();  
}  
  
public class Banana extends Producto{  
  
    public Banana(){  
        super(200);  
    }  
  
    @Override  
    public String getName(){  
        return "banana";  
    }  
}
```

4.4 TIPOS DE HERENCIAS.

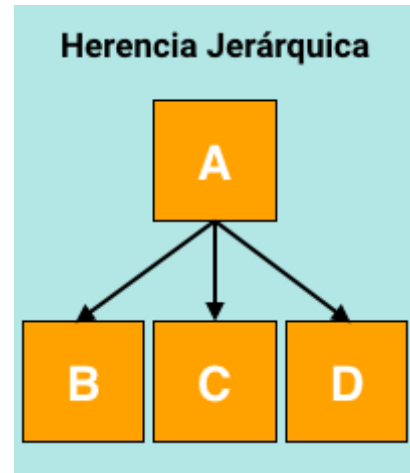
- **Herencia única:** en la herencia única, las subclases heredan las características de solo una superclase. En la imagen a continuación, la clase A sirve como clase base para la clase derivada B.



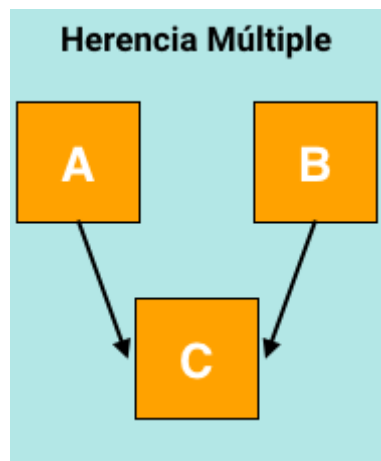
- **Herencia Multinivel:** en la herencia multinivel, una clase derivada heredará una clase padre y, además, la clase derivada también actuará como la clase padre de otra clase. En la imagen inferior, la clase A sirve como clase padre para la clase derivada B, que a su vez sirve como clase padre para la clase derivada C.



- **Herencia Jerárquica:** en la herencia jerárquica, una clase sirve como una superclase (clase base) para más de una subclase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, C y D.



- **Herencia múltiple**, una clase puede tener más de una superclase y heredar características de todas las clases principales. Tenga en cuenta que Java no admite herencia múltiple con clases. En Java, podemos lograr herencia múltiple **solo a través de Interfaces (ver punto 5)**. En la imagen a continuación, la Clase C se deriva de la interfaz A y B.



5. INTERFACES

En el punto anterior hemos hablado de las clases abstractas y de la herencia múltiple, conceptos necesarios para entender qué es una interfaz y para qué sirve. Una interfaz no es más que una clase donde todos sus métodos son abstractos y solo tiene constantes estáticas.

Las interfaces definen un protocolo de comportamiento y proporcionan un formato común para implementarlo en las clases. De esta manera una interfaz va a definir qué se debe hacer pero no como.

Una interfaz se crea utilizando la palabra clave **interface** en lugar de **class**.

```
public          interface  NombreInterface  [extends  
Interface1, Interface2, ...]{  
  
}  

```

Algunas características de las interfaces:

- La interfaz puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases. Si tiene el modificador public el archivo .java que la contiene debe tener el mismo nombre que la interfaz.
- En los métodos abstractos no es necesario escribir abstract
- Todos los atributos son constantes, públicos y estáticos. Por lo tanto, se pueden omitir los modificadores public static final cuando se declara el atributo. Se deben inicializar en la misma instrucción de declaración.
- Los nombres de las interface suelen acabar en able aunque no es necesario: Configurable, Arrancable, Dibujable, Comparable, Clonable, etc.

NOTA: las siguientes características son avanzadas, no necesario saber, solo para quien quiera por curiosidad

- Los métodos por defecto se especifican mediante el modificador default. En la interfaz se escribe el código del método. Este método estará disponible para todas las clases que la implementen, no estando obligadas a escribir su código. Solo lo incluirán en el caso de querer modificarlo. (Solo a partir de Java 8) *nombreInterface.super.metodo*
- Los métodos estáticos se especifican mediante la palabra reservada static. (A partir de Java 8 las interfaces también pueden contener métodos static. En la interfaz se escribe el código del método. Los métodos static NO pueden ser redefinidos en las clases que la implementan.) *nombreInterface.metodoStatic*
- Los métodos privados se especifican mediante el modificador de acceso private. (A partir de Java 9 las interfaces también pueden contener métodos privados. Estos métodos solo son accesibles dentro de la propia interface, por lo tanto, las clases que la implementen no podrán utilizarse.)

5.1 USO DE INTERFACES.

Para declarar una clase que implemente una interfaz es necesario utilizar la palabra reservada implements en la cabecera de declaración de la clase. Las cabeceras de los métodos (identificador y número y tipo de parámetros) deben aparecer en la clase tal y como aparecen en la interfaz implementada.

```
public class nombreClase implements Interface {  
}
```

5.2 JERARQUÍA ENTRE INTERFACES.

La jerarquía entre interfaces permite la herencia simple y múltiple. Es decir, tanto la declaración de una clase, como la de una interfaz pueden incluir la implementación de otras interfaces. Los identificadores de las interfaces se separan por comas. Por ejemplo, la interfaz Una implementa otras dos interfaces: Dos y Tres.

```
public interface Una implements Dos, Tres {  
    // Cuerpo de la interfaz ...  
}
```

Las clases que implementan la interfaz Una también lo hacen con Dos y Tres.

Veamos un ejemplo de todo esto:

```
public interface Figura {  
    float PI = 3.1416f;    // Por defecto public static  
    final.  
    float area();    // Por defecto abstract public  
}
```

```
public class Cuadrado implements Figura {  
    private float lado;  
  
    public Cuadrado (float lado) {  
        this.lado = lado;  
    }  
    public float area() {  
        return lado*lado;  
    }  
}
```

```
public class Circulo implements Figura{ // La clase  
    implementa la interface Figura  
    private float diametro;  
    public Circulo (float diametro) {  
        this.diametro = diametro;  
    }  
    public float area() {  
        return (PI*diametro*diametro/4f);  
    }  
}
```



Centro Oficial FP
Digital & Tech

}

}