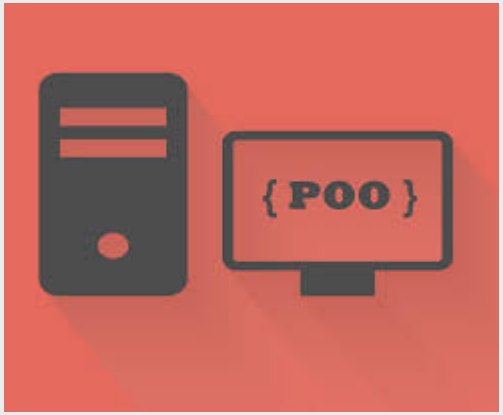
PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

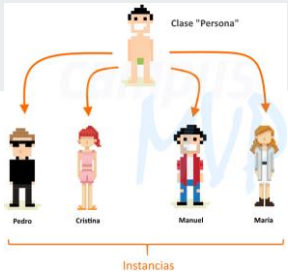
Encarnación Sánchez Gallego

ÍNDICE



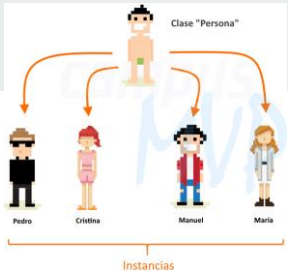
1. Introducción. Conceptos.
2. Características de la POO
3. Propiedades y métodos de los objetos
4. Parámetros y valores devueltos.
5. Constructores y destructores de objetos.
6. Uso de métodos estáticos y dinámicos.
7. Librerías de objetos (paquetes)

INTRODUCCIÓN. CONCEPTOS



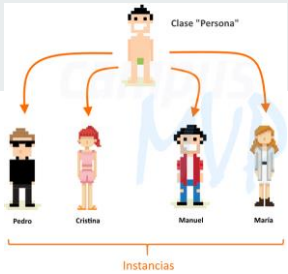
- La POO es un **paradigma de programación** totalmente diferente al método clásico de programación, el cual utiliza objetos y su comportamiento para resolver problemas y generar programas y aplicaciones informáticas.
- Con la POO se aumenta la modularidad de los programas y la reutilización de los mismos. Además en la POO encontraremos nuevos conceptos como son el polimorfismo, el encapsulamiento o la herencia
- Generalmente, los lenguajes de programación de última generación permiten la POO que se entiende como una evolución en el mundo de la programación.

INTRODUCCIÓN. CONCEPTOS



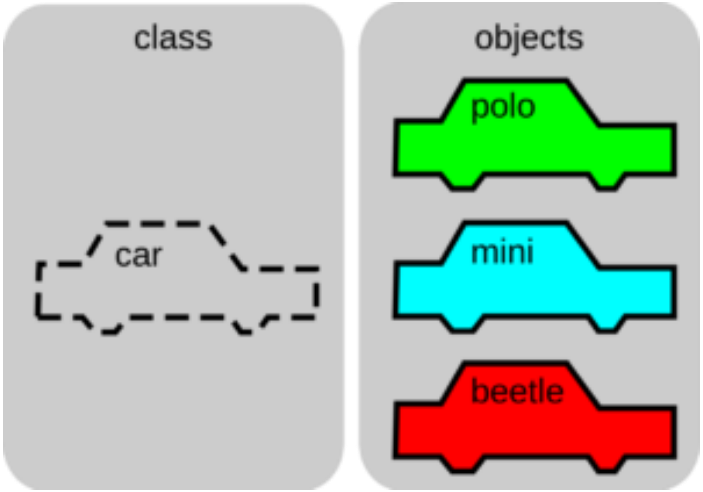
- En POO se reduce la complejidad del problema global utilizando esas pequeñas unidades (objetos) que resuelven problemas más pequeños.
- Los objetos van a poder enviar y recibir mensajes entre ellos que activan comportamientos internos y además podrán “ampliar” su capacidad comunicándose con más y más objetos según las necesidades.
- Una **CLASE** es un molde del que se generan los objetos. Los objetos se instancian y se generan. Instancia y objeto son sinónimos. Por ejemplo “Felipe” es un objeto concreto de la clase “Alumno”. **El nombre de la clase debe coincidir con el nombre del fichero donde se declara.**

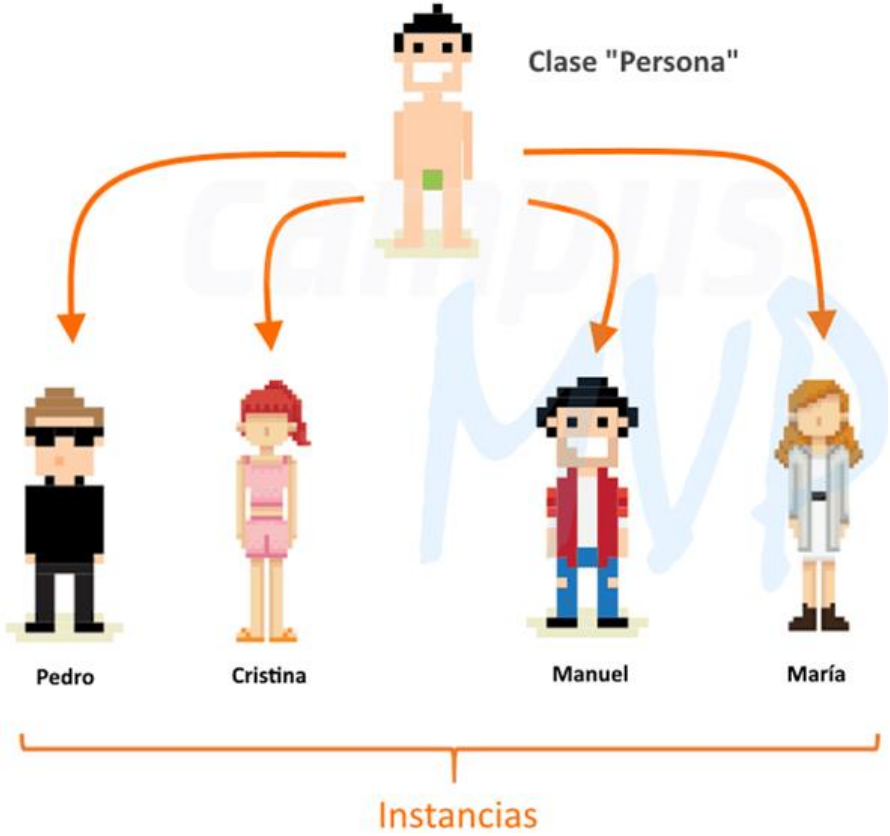
INTRODUCCIÓN. CONCEPTOS



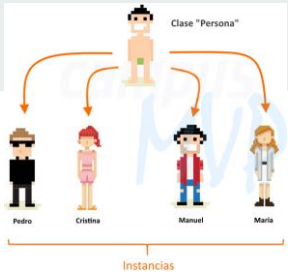
- Las **clases** permiten a los programadores **abstraer** el problema a resolver ocultando los datos y la manera que estos se manejan para llegar a la solución (se oculta la implementación).
- Habitualmente para acceder a la información de una clase (por ejemplo la Edad) no se podrá acceder directamente al dato sino que se hace a través de un método (getEdad()). Igualmente para asignar un valor se hará a través de otro método (setEdad())
- Cuando diseñemos nuestras clases deberemos de cuidar lo siguiente:
  - Acceso limitado a los datos internos (getters y setters).
  - Los métodos actúan de interfaz con otras clases por lo que podremos modificar su código con independenciar del resto.

INTRODUCCIÓN. CONCEPTOS



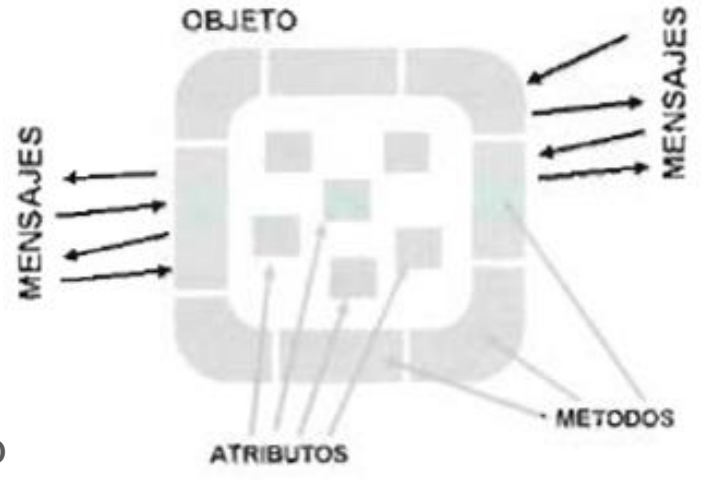


INTRODUCCIÓN. CONCEPTOS

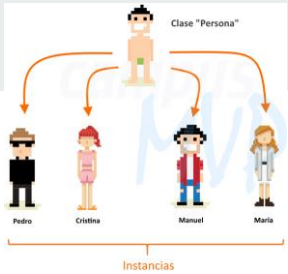


Características de los Objetos:

- **Identidad**. Cada objeto es único y diferente de otros objetos. El loro “Felipe” es diferente de otros loros que también sean verdes, domésticos y tengan 2 años.
- **Estado**. Serán los valores de los atributos de los objetos (sus características), en el caso de los valores de los objetos de la clase pájaro serían nombre, color, edad, doméstico, etc.
- **Comportamiento**. Serán los métodos y procedimientos que tiene dicho objeto. Dependiendo del tipo o clase de objeto, estos realizarán una operación u otras (por ejemplo en el caso del pájaro, cantar, volar, hablar, etc).



INTRODUCCIÓN. CONCEPTOS



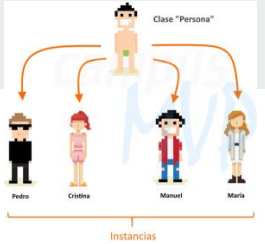
Mensajes:

- Los programas o aplicaciones orientadas a objetos están compuestos por objetos, los cuales interactúan unos con otros a través del paso de mensajes (paso de parámetros). Cuando un objeto recibe un mensaje lo que hace es ejecutar el método asociado (una función programada).

Métodos:

- Son los procedimientos que ejecuta el objeto cuando recibe un mensaje vinculado a ese método concreto. En ocasiones este método envía mensajes a otros objetos, solicitando acciones o información.

En los programas OO se crean los objetos y entre ellos se envían mensajes para procesar la información. Cuando todo termina se destruyen los objetos y se libera la memoria que estaban ocupando.



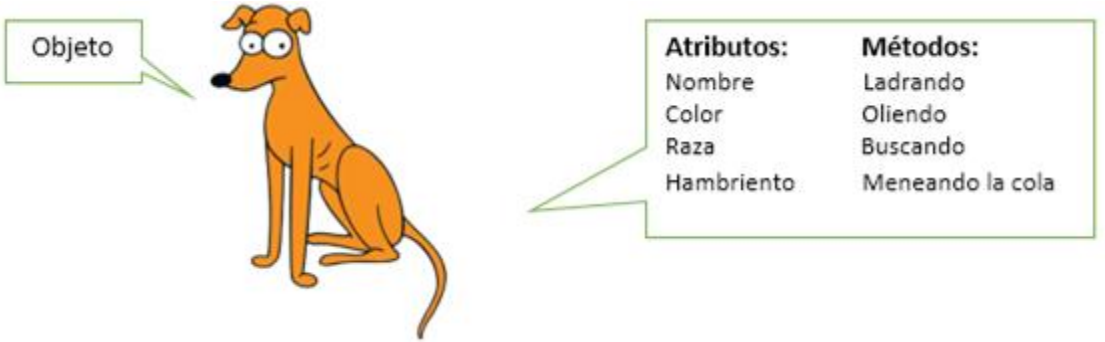
INTRODUCCIÓN. CONCEPTOS

Como hemos visto, los **OBJETOS** del mundo real tienen 2 características, el estado y el comportamiento.

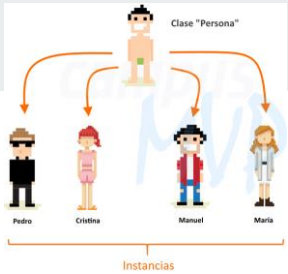
Por ejemplo en el caso de los perros:

- Tienen un estado como su nombre, su color, su raza o si está hambriento.
- Y también su comportamiento como ladrar, oler, buscar o menear la cola.

Al programar una aplicación en Java debemos modelar ambos a nivel de clase y generar los objetos.

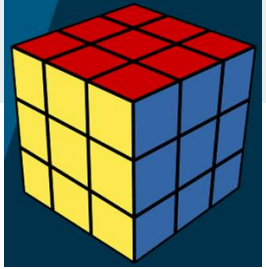


INTRODUCCIÓN. CONCEPTOS



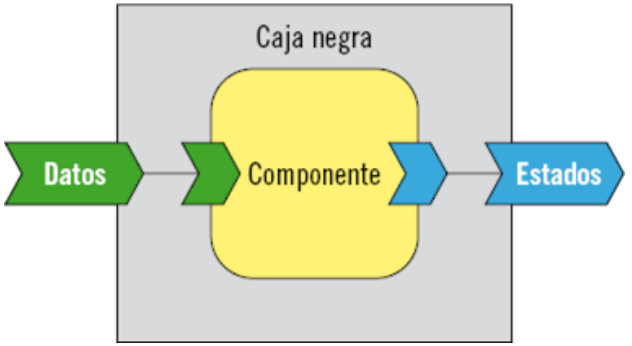
La POO proporciona los beneficios:

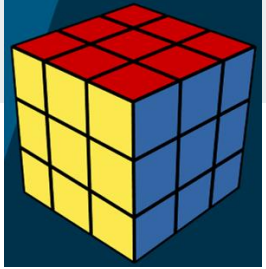
- **Modularidad**: el código fuente de los objetos puede mantenerse y reescribirse sin que ello implique la reprogramación del código de otros objetos.
- **Reutilización de código**: es sencillo reutilizar clases y objetos de terceras personas. La ventaja es que no tenemos que conocer los detalles de su implementación interna sino solamente su interfaz.
- **Facilidad de testeo y reprogramación**: si tenemos un objeto que está dando problemas en una aplicación no tenemos que reescribir el código de toda la aplicación, sino que podemos reemplazar el objeto por otro similar, o bien reprogramarlo sin tocar el resto de la aplicación.
- **Ocultación de información**: En POO se ocultan los detalles de implementación y sólo se publica la interfaz.



CARACTERÍSTICAS DE LA POO

- Abstracción: Cuando se programa orientado a objetos lo que se hace es abstraer las características de los objetos que van a tomar parte del programa y crear las clases con sus atributos y sus métodos.
- Encapsulamiento: Cuando se programa en orientación a objetos, los objetos se ven según su comportamiento externo, por ejemplo en la clase pájaro se puede mandar un mensaje para que cante y el objeto pájaro ejecuta su método cantar(). El programador no tiene que saber cómo funciona el método. Las clases son como una caja negra de la que sólo se conoce la interfaz para la comunicación.



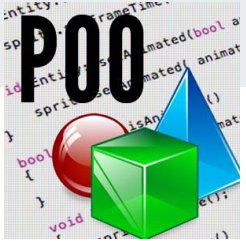


CARACTERÍSTICAS DE LA POO

- Herencia: Todas las clases se estructuran formando jerarquía de clases.

- Polimorfismo: Permite crear varias formas del mismo método, de tal forma que un mismo método ofrece comportamientos diferentes. Por ejemplo el método “sort” para ordenar de la clase “Arrays” se puede llamar para ordenar números enteros o cadenas de texto.

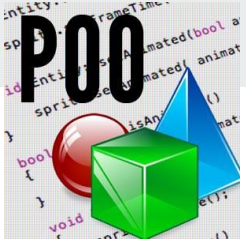
PROPIEDADES Y MÉTODOS DE LOS OBJETOS



En una **clase** se agrupan datos/atributos (variables) y métodos (funciones). Todas las variables y funciones creadas en Java deben pertenecer a una clase, con lo cual no existen variables o funciones globales como en otros lenguajes de programación.

En una **clase** Java los **atributos** pueden ser de **tipos primitivos** como “char”, “int” o “boolean” o de **tipos referenciados** como cualquier objeto de otra clase. Por ejemplo en un objeto de la clase coche podría haber una propiedad que fuera de la clase “motor”

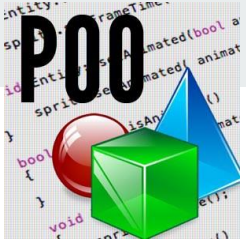
En el siguiente ejemplo vamos a ver una clase con sus atributos de clase (edad y color) y sus métodos (setEdad, setColor, printEdad y printColor).



PROPIEDADES Y MÉTODOS DE LOS OBJETOS

```
public class pajaro {
        private char color; //Propiedad o atributo de color                    /*Atributos o propiedades*/
        private int edad; //Propiedad o atributo de edad
        public void setEdad(int e){edad=e;}                                                    /*Métodos
de la clase*/
        public void printEdad(){System.out.println(edad);}
        public void setColor(char c){color=c;}
        public void printColor(){
                switch(color){          //Los pájaros sólo pueden ser verdes, amarillos, grises, negros o
        blancos
                        case 'v': System.out.println("Verde.");break;
                        case 'a': System.out.println("Amarillo.");break;
                        case 'g': System.out.println("Gris.");break;
                        case 'n': System.out.println("Negro.");break;
                        case 'b': System.out.println("Blanco.");break;
                        default: System.out.println("Color no establecido.");
```

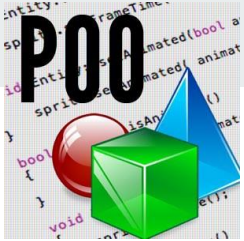
PROPIEDADES Y MÉTODOS DE LOS OBJETOS




```
public class test {
        public static void main(String[] args){
                pajaro p;
                p=new pajaro();
                p.setEdad(5);
                p.printEdad();
        }
}
```

Como se puede ver en el código anterior hay 2 clases diferentes (pajaro y test) los cuales se implementan en 2 ficheros pájaro.java y test.java, en la clase test se crea un objeto de la clase pájaro y se llama a los métodos para actualizar la edad y mostrarla en pantalla. En ningún momento la clase test puede acceder a los métodos o atributos privados de la clase pájaro (por la abstracción). Test sólo puede y debe conocer los métodos públicos.

PROPIEDADES Y MÉTODOS DE LOS OBJETOS

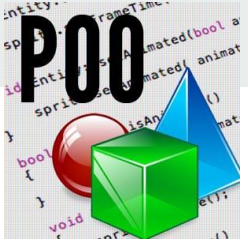


Para todos ellos, Java dispone de varios **niveles de acceso** según la necesidad:

- Public: acceso público. Podrá ser accedido desde cualquier clase.
- Protected: acceso protegido. Sólo accesible en el paquete y por subclases.
- Private: acceso privado. Sólo accesible desde el interior de la propia clase.
- No especificado: accesible desde cualquier clase del paquete. No es recomendable dejar sin especificar el nivel de acceso.

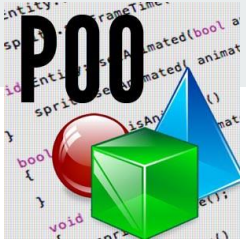
Cuando especificamos el nivel de acceso a un atributo o método de una clase, lo que estamos especificando es el nivel de accesibilidad que va a tener ese atributo o método que puede ir desde acceso más restrictivo (private) al menos restrictivo (public).

Cuando se profundice en la herencia, las subclases o herederas poseen las características de la clase padre y pueden añadir nuevas propiedades.



PROPIEDADES Y MÉTODOS DE LOS OBJETOS

Modificador de acceso	public	protected	private	Sin especificar (acceso paquete)
¿El método o atributo es accesible desde la propia clase?	SÍ	SÍ	SÍ	SÍ
¿El método o atributo es accesible desde Otras clases en el mismo paquete?	SÍ	SÍ	NO	SÍ
¿El método o atributo es accesible desde una subclase en el mismo paquete?	SÍ	SÍ	NO	SÍ
¿El método o atributo es accesible desde subclases en otros paquetes?	SÍ	(*)	NO	NO
¿El método o atributo es accesible desde otras clases en otros paquetes?	SÍ	NO	NO	NO

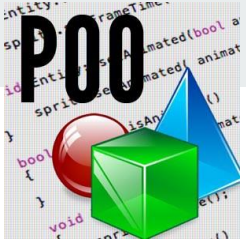


PROPIEDADES Y MÉTODOS DE LOS OBJETOS

También podremos utilizar los **niveles de acceso** para definir las relaciones que una clase tendrá con las otras clases dentro del paquete.

- Public: acceso público. Podrá ser accedido desde cualquier clase.
- No especificado: accesible sólo desde cualquier clase del paquete.

Clase pública	Clase NO definida como pública
`public class miClase { ..... }`	`class miClase { ..... }`
Puede ser utilizada por cualquier clase.	Puede ser utilizada SOLO por clases de su propio paquete.



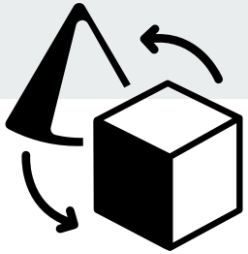
PROPIEDADES Y MÉTODOS DE LOS OBJETOS

Cuando programamos una clase hay una **referencia** al objeto que se está ejecutando que se llama "**this**" (no estáticos) y que será muy utilizada.
En este ejemplo vemos como se puede omitir a veces aunque es recomendable utilizarla para evitar confusiones.

```
class rectangulo
{
    private int ancho = 0;
    private int alto = 0;
    rectangulo(int an, int al){
          ancho = an; //se puede omitir el this
          this.alto = al;
    }
    public int getAncho(){return this.ancho;}
    public int getAlto(){return alto;} //se puede omitir el this
    public rectangulo incrementarAncho(){
          ancho++; //se puede omitir el this
          return this;
    }
    public rectangulo incrementarAlto(){
          this.alto++;
          return this;
    }
}
```

Además se devuelve una referencia a "**this**" en los métodos de

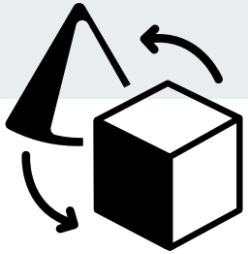
MÉTODOS PROPIOS DE OBJECT



Cualquier clase implementada en Java siempre va a ser una subclase de la clase Object, y por ello va a tener los métodos de object (herencia):

Método	Descripción
clone()	Permite "clonar" un objeto.
equals()	Permite comparar un objeto con otro.
toString()	Devuelve el nombre de la clase.
finalize()	Método invocado por el recolector de basura (garbage collector) para borrar definitivamente el objeto.

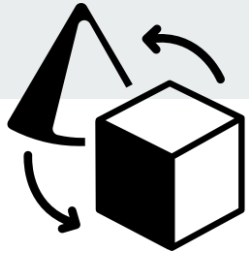
MÉTODOS PROPIOS DE OBJECT



Método “clone()”:

- Copia un objeto. Utilizar este objeto equivale a utilizar un Constructor de copia. Es el mecanismo que utiliza Java para clonar objetos.
- Es posible implementar un método clone() que sobreescriba el método original para actuar de manera más específica.
- Podemos hacer una copia superficial en la que únicamente se hace una copia del contenido de un objeto en otro.
- Y también podemos realizar una copia en profundidad que hace una copia selectiva del contenido de un objeto en otro.
- No es la mejor forma pero al realizar la clonación se debe tener un objeto con “implements Cloneable” e implementar el método “clone”.

MÉTODOS PROPIOS DE OBJECT



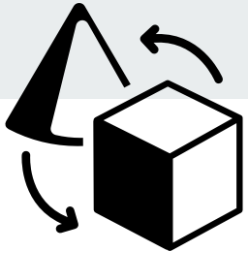
Ejemplo:

```
public class rectangulo implements Cloneable
{
    private int ancho;
    private int alto;
    private String nombre;
    public Object clone(){
        Object objeto=null;
        try{
            objeto =super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" Error al duplicar");
        }
        return objeto;
    }
.....
}
```

```
class testeoclone {

    public static void main(String[] args) {
        rectangulo r1 = new rectangulo(5,7);
        rectangulo r2 = (rectangulo) r1.clone();
        r2.incrementarAncho();
        r2.incrementarAlto();
        r1.setNombre("Chiquito");
        r2.setNombre("Grande");
        System.out.println("Alto: "+r1.getAlto());
        System.out.println("Ancho: "+r1.getAncho());
        System.out.println("Alto: "+r2.getAlto());
        System.out.println("Ancho: "+r2.getAncho());
        System.out.println("Nombre: "+r1.getNombre());
        System.out.println("Nombre: "+r2.getNombre());
    }
}
```

MÉTODOS PROPIOS DE OBJECT

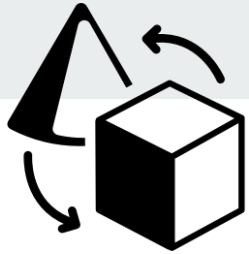


Método “equals()”:

- Permite realizar una comparación entre un objeto y otro. Lo que hace es comprobar que ambas referencias sean iguales, con lo cual no obtenemos más ventaja que con el operador “==”, ya que no se hace una comparación en profundidad. Ejemplo:

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo(5,7);
rectangulo r3 = r1;
if (r1.equals(r2)){
        System.out.println(«Iguales r1 y r2(equals)»);
}
if (r1.equals(r3)){
        System.out.println(«Iguales r1 y r3(equals)»);
}
```

MÉTODOS PROPIOS DE OBJECT



Método “toString()”:

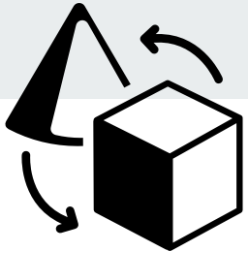
- Permite obtener el nombre de la clase desde la cual fue invocada. Además el nombre de la clase devuelve el carácter @ y la representación hexadecimal del código hash del código.
- Con este ejemplo podemos observar que al hacer r3=r1 lo que hacemos es que ambas referencias apuntan al mismo objeto, con lo que al invocar al método toString(), el resultado es el mismo.

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo(5,7);
rectangulo r3 = r1;
System.out.println(r1.toString());
System.out.println(r2.toString());
System.out.println(r3.toString());
```

Este código devolverá por pantalla lo siguiente:

```
rectangulo@19821f
rectangulo@addbf1
rectangulo@19821f
```

MÉTODOS PROPIOS DE OBJECT



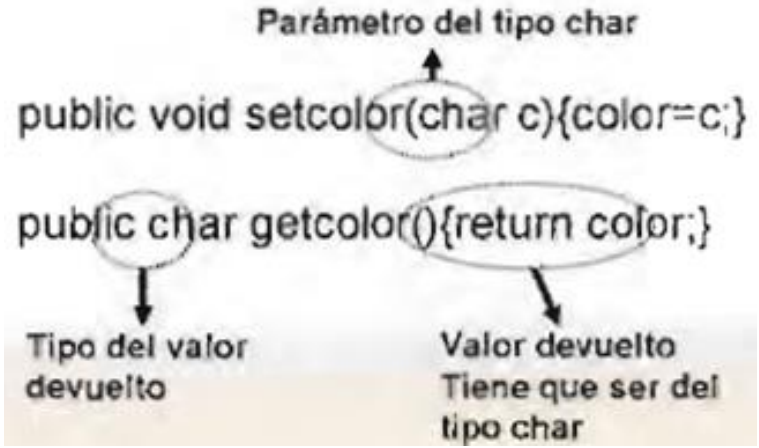
Método “finalize()”:

- Este método está “deprecated” lo que significa que ya no debemos utilizarlo y que en posteriores versiones dejará de existir.
- Éste método es utilizado cuando el recolector de basura (**garbage collector**) elimina un objeto al que ya no apuntan más referencias para liberar su memoria ocupada.
- Si el programador necesita realizar una acción una vez destruido un objeto deberá reescribir este método.

```
wait(long timeoutMillis)                    void
wait(long timeoutMillis, int nanos)         void
finalize()                                  void
```

PARÁMETROS Y VALORES DEVUELTOS

Los métodos pueden permitir que se les llame especificando una serie de valores. A estos valores se les denomina parámetros, estos pueden tener un tipo básico (char, int, boolean, etc) o bien ser un objeto. Además los métodos (salvo el constructor) pueden retornar un valor o no.



En este ejemplo, podemos ver 2 métodos para la clase “pajaro”, **setcolor** la cual admite un parámetro y **getColor** la cual devuelve un valor de tipo char.

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

En Java existen unos **métodos especiales** que son los constructores y destructores de objetos. Estos métodos son opcionales y debemos programarlos cuando los necesitemos aunque siempre se crean por defecto (en blanco en este caso).

- El **constructor** del objeto es un procedimiento que automáticamente se llama cuando se crea un objeto de esa clase y se reserva memoria para el. La función del constructor es inicializar el objeto.
- El **destructor** se ejecuta automáticamente siempre que se destruye un objeto de dicha clase, estos se utilizan para liberar recursos y cerrar flujos abiertos. Los destructores no reciben parámetros, al contrario que los constructores, y la sobrecarga no está permitida.

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

Cuando se **crea** un objeto (“new objeto()”) sucede lo siguiente en Java:

- Crea memoria para el objeto tras el operador “new”.
- Inicializa los atributos del objeto (los no inicializados según los valores del cuadro de abajo).
- Se ejecutan los inicializadores de objeto.
- Llama al constructor adecuado según los parámetros utilizados.

Inicialización predeterminada del sistema	
Atributos numéricos	0
Atributos Alfanuméricos	Nulo o cadena vacía en caso de *String*
Referencias a objetos	*Null*

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

Un constructor puede estar "**sobrecargado**", es decir, que puede estar definido para recibir distinto número y tipo de parámetros y así hacer distintas inicializaciones según los parámetros recibidos.

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

Ejemplo para la clase “pajaro” donde el constructor está **sobrecargado**:

```
public class pajaro {
        private char color;                                    //Propiedad o
atributo de color
        private int edad;
                //Propiedad o atributo de edad
        pajaro(){color='v';edad=0;}                            //Constructor sin
parámetros
        pajaro(char c, int e){color=c;edad=e;}    //Constructor con parámetros
        /*Resto de métodos de la clase*/
        public static void main(String[] args){
                pajaro p1, p2;
```

Java elige el constructor adecuado según los parámetros que se utilicen.

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

Los constructores de las clases devuelven un **objeto**, una **instancia**, de la clase que acabamos de utilizar.
Debido a ello debemos asignar a una **referencia**, variable que apunta a una dirección de memoria, del tipo correspondiente.
En el siguiente ejemplo tenemos 3 referencias, variables, de la clase "rectangulo" y cada caso se llama a un constructor diferente aunque en los 3 casos devuelve una instancia de "rectangulo".

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo();
rectangulo r3 = new rectangulo(8);
```

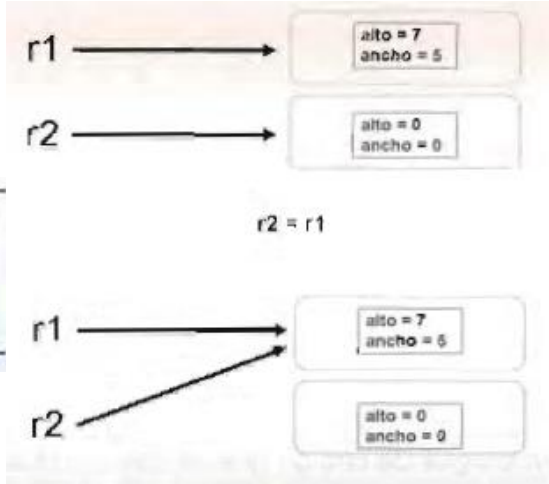
CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

```
public class rectangulo
{
    private int ancho;
    private int alto;
    rectangulo(int an, int al){
          this.ancho = an;
          this.alto = al;
    }
    rectangulo(){ ancho=alto=0;       }
    rectangulo(int dato){ ancho=alto=dato; }
    public int getAncho(){return this.ancho;}
    public int getAlto(){return this.alto;}
    public rectangulo incrementarAncho(){
          ancho++;
          return this;
    }
    public rectangulo incrementarAlto(){
          this.alto++;
          return this;
    }
}
```

¿Qué ocurre?:

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo();
r2=r1;
r2.incrementarAncho();
r2.incrementarAlto();
System.out.println("Alto: "+r1.getAlto());
System.out.println("Ancho: "+r1.getAncho());
```

Opción 1	Opción 2
Alto: 7 Ancho: 5	Alto: 8 Ancho: 6



CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

En el caso anterior sólo se igualan las referencias y por tanto ambas apuntan al mismo objeto.

Para tener 2 objetos diferentes con los mismos valores podemos definir un **constructor copia**, es decir, un constructor que recibe un objeto como parámetro y copia su contenido.

```
rectangulo (rectangulo r){
        this.ancho = r.getAncho();
        this.alto = r.getAlto();
}
```

```
rectangulo r1 = new rectangulo(5,7);
rectangulo r2 = new rectangulo(r1);
r2.incrementarAncho();
r2.incrementarAlto();
System.out.println("Alto: "+r1.getAlto());
System.out.println("Ancho: "+r1.getAncho());
```

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

Dentro de una clase podemos definir **bloques "static"** que se ejecutará sólo una vez cuando se utiliza la clase (da igual las veces que se cree un objeto). Los inicializadores static siguen la siguientes reglas:

- No devuelven ningún valor.
- Son métodos sin nombre.
- Ideal para inicializar objetos o elementos complicados.
- Permiten gestionar excepciones con try…catch
- Puede haber más de un inicializador static y se ejecutará en ese orden.
- Pueden invocar métodos nativos o inicializar variables static.
- Existen inicializadores de objetos usados en clases anónimas y que no tienen modificador static.

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS

Ejemplo:

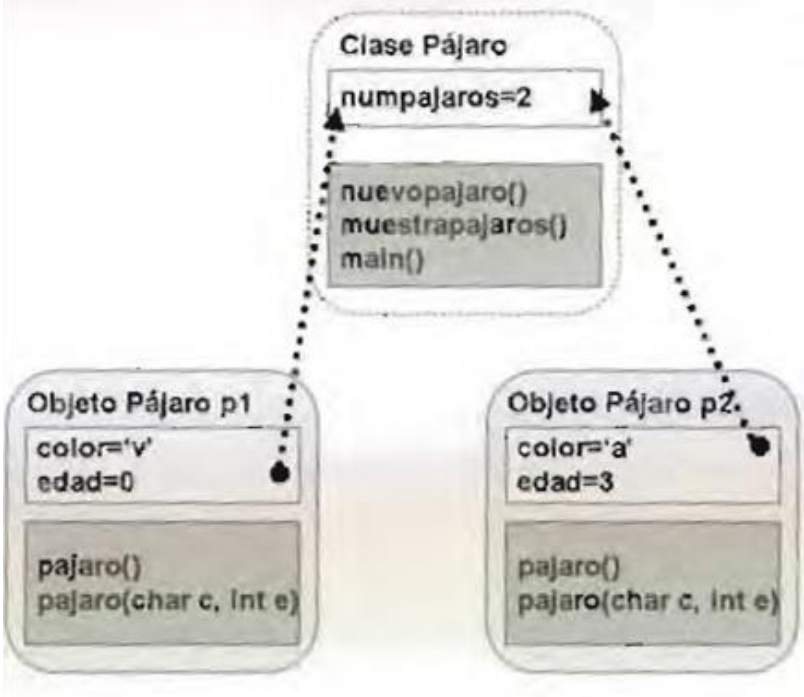
```
public class testInicializador{
        static{
        System.out.println("Llamada al inicializador");
        }
        static{
        System.out.println("Llamada al segundo inicializador");
        }
        testInicializador(){
        System.out.println("Llamada al constructor");
        }
}
```

```
Llamada al inicializador
Llamada al segundo inicializador
Llamada al constructor
Llamada al constructor
Llamada al constructor
Presione una tecla para continuar...
```

```
class test {
        public static void main(String[] args) {
        testInicializador t1 = new  testInicializador();
        testInicializador t2 = new  testInicializador();
        testInicializador t3 = new  testInicializador();
    }
}
```

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

En este apartado se va a ver cuándo un método debe ser **estático** y cuando no. Cuando un método o atributo se define como **static** quiere decir que se va a crear para esa clase solo una instancia de ese método o atributo. En el siguiente ejemplo se crea un atributo **numPajaros** que cuenta el número de pájaros que se van generando, sino fuera estático, sería imposible generar el número de pájaros.



USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

En este ejemplo, el atributo numpajaros y los métodos nuevopajaro(), muestrapajaros() y main() se comparten por todos los objetos creados.

```
public class pajaro {
        private static int numpajaros=0;                                        /*Atributos
o propiedades*/
        private char color; //Propiedad o atributo de color
        private int edad; //Propiedad o atributo de edad
        pajaro(){color='v';edad=0;nuevoPajaro();}                               //Sin parámetros
        pajaro(char c, int e){color=c;edad=e;nuevoPajaro();}                    //Con parámetros
        static void nuevoPajaro(){numpajaros++;}                                //Resto de métodos
        static void muestraPajaros(){System.out.println(numpajaros);}
        public static void main(String[] args){
                pajaro p1, p2;
                p1=new pajaro();                                p2=new pajaro('a',3);
                p1.muestraPajaros();                    p2.muestraPajaros();
```

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

- Al no poder crear una variable global para toda nuestra aplicación se pueden utilizar los miembros estáticos (**static**) de una clase para utilizar una variable única y que pueda utilizar todos los objetos de dicha clase.
- **Los miembros no estáticos serán "normales" o "de instancia".**
- Ejemplo de una clase con un miembro estático, "numcohetes" que almacenará el número de objetos cohete que se van creando.

```
public class cohete{
    private static int numcohetes=0;
    cohete(){  numcohetes++; }
    public int getcohetes(){ return numcohetes;}
}
```

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

- En este ejemplo, cuando desde otra clase (en el ejemplo) se crean varios objetos de la clase cohete y se llama al método getcohetes() vemos que la variable marca “3”

```
public class testestaticos {
    public static void main(String[] args) {
        cohete c1 = new cohete();
        cohete c2 = new cohete();
        cohete c3 = new cohete();
        System.out.println(c1.getcohetes());
        System.out.println(c3.getcohetes());
    }
}
```

- Al haber declarado “numcohetes” como “private”, no es posible desde “testestaticos” acceder a c1.numcohetes.

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

Métodos de instancia (No estáticos) y de clase (Estáticos):

- Los métodos de una clase son una abstracción del comportamiento de la misma. Los algoritmos formarán parte de los métodos y contendrán la lógica de la aplicación que queremos desarrollar.
- Podemos dividir los métodos en 2 bloques:
  - **Métodos de instancia**: Son aquellos utilizados por la instancia.
  - **Métodos de clase**: Son aquellos comunes para una clase.
- Los métodos de instancia son, por así decirlo, los llamados métodos comunes. Cada instancia u objeto tendrá sus propios métodos independientes del mismo método de otro objeto de la misma clase.

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

Ejemplo de método de instancia:

```
public class cuadrado{
    private int lado;
    cuadrado(int l){ this.lado = l; }
    public int getArea(){ return lado*lado; }
}
```

Los métodos de instancia pueden acceder a los miembros de instancia (no static) y también a los miembros de clase (static) pero no al revés.

```
class test {
    public static int var;
    public int var2;
    public void prueba(){
        var = 3;
        var2 = 5;
    }
}
```

No obstante en vez de utilizar la línea:

```
var = 3;
```

Quizás hubiese sido más correcto utilizar la siguiente:

```
test.var = 3;
```

La llamada a un método de instancia sería la siguiente:

```
test t = new test();
t.prueba();
```

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

Ejemplo de método de clase:

```
public class Test {
    public int dato=0;
    public static int datostatico=0;
    public void metodo(){this.datostatico++;}
    public static void metodostatico(){
            this.datostatico++; // Esto da error al compilar
            datostatico++;
    }
    public static void main(String[] args) {
            dato++;  // Esto da error al compilar
            datostatico++;
            metodostatico();
            metodo();  // Esto da error al compilar
    }
```

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

Resumiendo:

Método	Llamada	Declaración	Acceso
Clase	Clase.metodo(parámetros)	static	Miembros de clase.
Instancia	Instancia.metodo(parámetros)		Miembros de clase y de instancia.

Y como ejemplo podemos poner la clase Math (java.lang.Math) que contiene métodos “static” y que pueden ser llamados anteponiendo el nombre “Math”:

```
Math.random();
Math.cos(angulo);
```

USO DE MÉTODOS ESTÁTICOS Y DINÁMICOS

Math:

Método	Descripción
static int abs(int a) static long abs(long a) static double abs(double a) static float abs(float a)	Devuelve el valor absoluto del parámetro pasado.
static int max(int a, int b) static long max(long a, long b) static double max(double a, double b) static float max(float a, float b)	Devuelve el mayor de los valores a ó b.
static int min(int a, int b) static long min(long a, long b) static double min(double a, double b) static float min(float a, float b)	Devuelve el menor de los valores a ó b.
static double pow(double a, double b)	Potencia de un número. Devuelve el valor de a elevado a b.
static double random()	Números aleatorios. Devuelve un número aleatorio de tipo double entre cero y uno (éste último no incluido).
static int round(float a) static long round(double a)	Redondeo. Redondea el parámetro a al valor entero más cercano.

ENCAPSULACIÓN Y VISIBILIDAD. INTERFACES

En POO existe otro concepto importante, el "**interface**" o interfaz, que tienen la apariencia de una clase con métodos vacíos (sin llaves).

```
public interface intfigura{
    int area();
}
```

- En este ejemplo tenemos un método "area" no definido en el interface (escribimos "interface" en lugar de "class") llamado intfigura.

Con un interfaz podemos definir métodos que serán desarrollados y escritos más tarde en las clases que implementen la interfaz.

ENCAPSULACIÓN Y VISIBILIDAD. INTERFACES

Utilizando el interfaz anterior, cuando queramos implementar el interfaz debemos desarrollar la función “area”. Si tuviera más métodos también deberían ser desarrollados para que compile la clase “rectangulo”:

```
public class rectangulo implements intfigura{
    private int ancho;
    private int alto;
    rectangulo (int an, int al){
            this.ancho = an;
            this.alto = al;
    }
```

LIBRERÍAS DE OBJETOS (PAQUETES)

El **paquete** o **package** es un conjunto de clases relacionadas entre sí y organizadas en grupos. Las clases que forman parte de un paquete no derivan todas ellas de una misma superclase. Por ejemplo java.io agrupa las clases que permiten a un programa realizar la entrada y salida de información. Un paquete puede contener a otro paquete. Con el uso de paquetes se evita conflictos, como llamar a 2 clases con el mismo nombre (si existe, estarán cada una en un paquete diferente).

Mediante la sentencia **Import** podemos usar una clase de otro paquete:

```
import java.lang.System;
```

También podemos cargar todas las clases de un paquete en particular:

```
import java.awt.*;
```

LIBRERÍAS DE OBJETOS (PAQUETES)

Sobre los paquetes hay que tener en cuenta:

- Un paquete es un conjunto de clases relacionadas entre sí.
- Un paquete puede contener a su vez subpaquetes.
- Java mantiene su biblioteca de clases en una estructura jerárquica.
- Para utilizar una clase hay que referirse a ella indicando el paquete al que pertenece (y subpaquetes). Por ejemplo “java.io.File”.
- Si es una clase dentro del mismo paquete, no es necesario referirnos al paquete al utilizar dicha clase.
- Las clases no pueden llamarse de la misma manera dentro del mismo paquete, por lo que para evitar dicho conflicto deben estar en paquetes diferentes.

LIBRERÍAS DE OBJETOS (PAQUETES)

Para definir que una clase pertenece a un paquete utilizamos la sintaxis:

package nombre_del_paquete;

package es.java.instituto.aulaNormal.mobiliario;

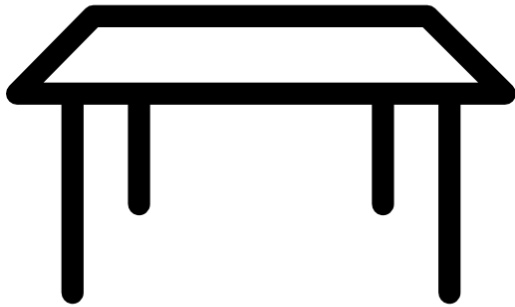
Y por tanto el identificador completo de una clase es el formado por el paquete junto con el nombre de la clase:

nombre_del_paquete +”.” + nombre_de_la_clase

es.java.instituto.aulaNormal.mobiliario.mesa;

De esta forma podríamos tener otra clase mesa:

es.java.instituto.aulaDibujo.mobiliario.mesa;



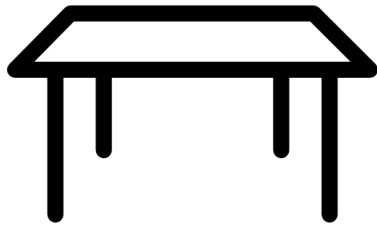
LIBRERÍAS DE OBJETOS (PAQUETES)

Como ya se dijo, para utilizar una clase en nuestro código, debemos indicar todo el paquete.

```
es.java.instituto.aulaNormal.mobiliario.mesa m = new
                        es.java.instituto.aulaNormal.mobiliario.mesa();
```

Para simplificar el código existe la instrucción “import” que permite indicar una clase concreta o un paquete completo:

```
import es.java.instituto.aulaNormal.mobiliario.mesa;
    import es.java.instituto.aulaNormal.mobiliario.*;
...
mesa m = new mesa();
```

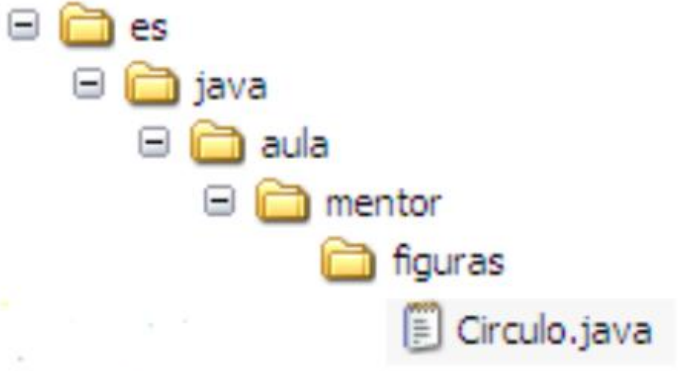


LIBRERÍAS DE OBJETOS (PAQUETES)

La forma habitual en la que se crean las librerías en Java y sus paquetes es que cada subpaquete se corresponde con un directorio o carpeta, siendo un árbol jerárquico.

Por ejemplo el paquete

```
package es.java.aula.mentor.figuras;
public class Circulo {...}
```



LIBRERÍAS DE OBJETOS (PAQUETES)

Paquete o librería	Descripción
java.io	Librería de Entrada/Salida. Permite la comunicación del programa con ficheros y periféricos.
java.lang	Paquete con clases esenciales de Java. No hace falta ejecutar la sentencia import para utilizar sus clases. Librería por defecto.
java.util	Librería con clases de utilidad general para el programador.
java.applet	Librería para desarrollar *applets*.
java.awt	Librerías con componentes para el desarrollo de interfaces de usuario.
java.swing	Librerías con componentes para el desarrollo de interfaces de usuario. Similar al paquete awt.
java.net	En combinación con la librería java.io, va a permitir crear aplicaciones que realicen comunicaciones con la red local e Internet.
java.math	Librería con todo tipo de utilidades matemáticas.
java.sql	Librería especializada en el manejo y comunicación con bases de datos.
java.security	Librería que implementa mecanismos de seguridad.
java.rmi	Paquete que permite el acceso a objetos situados en otros equipos (objetos remotos).
java.beans	Librería que permite la creación y manejo de componentes *javabeans*.

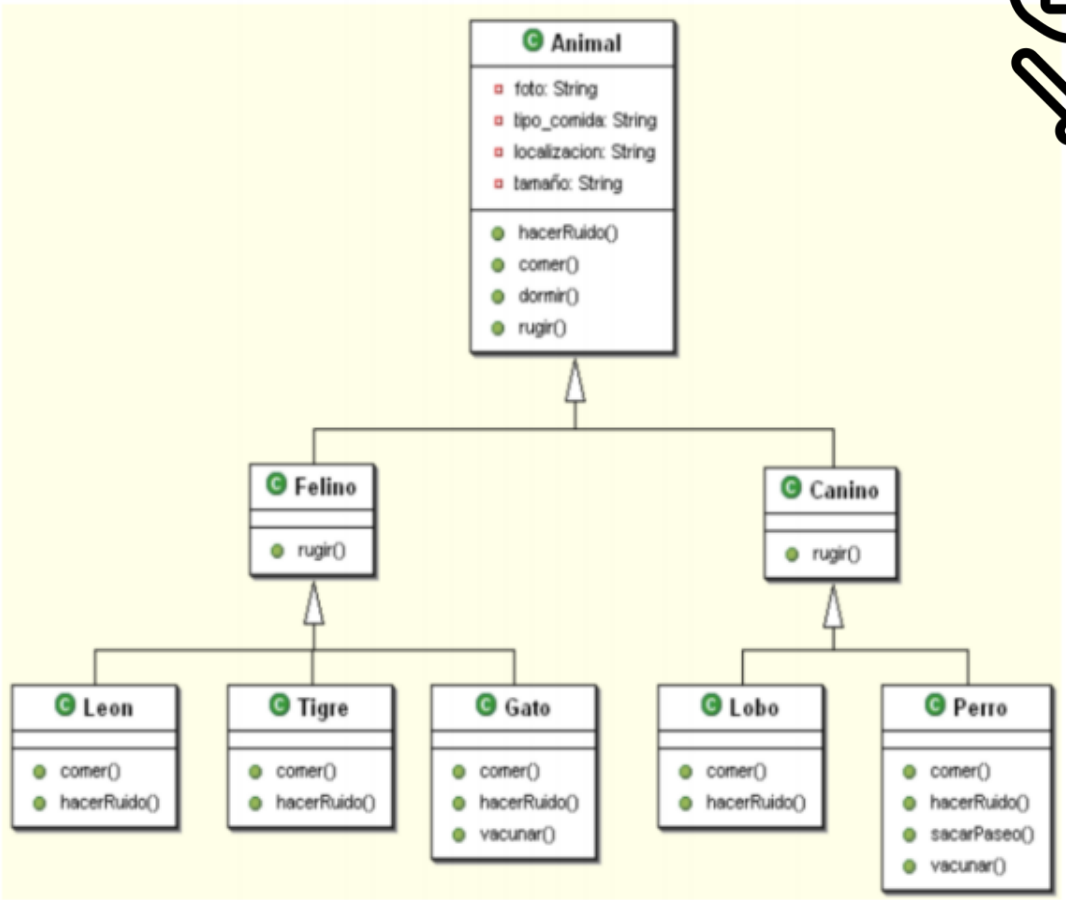
HERENCIA

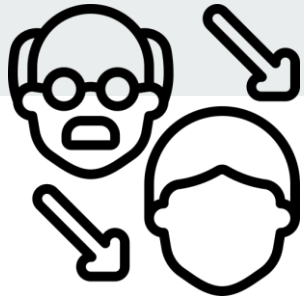
El último concepto que veremos es la **herencia**:

- La herencia es la base de la reutilización de código.
- Cuando una clase deriva de una clase padre, ésta hereda todos los atributos y métodos de su antecesor. También es posible **redefinir** los métodos y adaptarlos a la nueva clase o bien ampliarlos (@**Override**).
- Una clase puede tener sus propios atributos y métodos adicionales a lo heredado.
- Las clases por encima en la jerarquía de una clase dada, se denominan **superclases** o clases **padre**.
- Las clases por debajo en la jerarquía a una clase dada, se denominan **subclases** o clases **hijas**.

HERENCIA

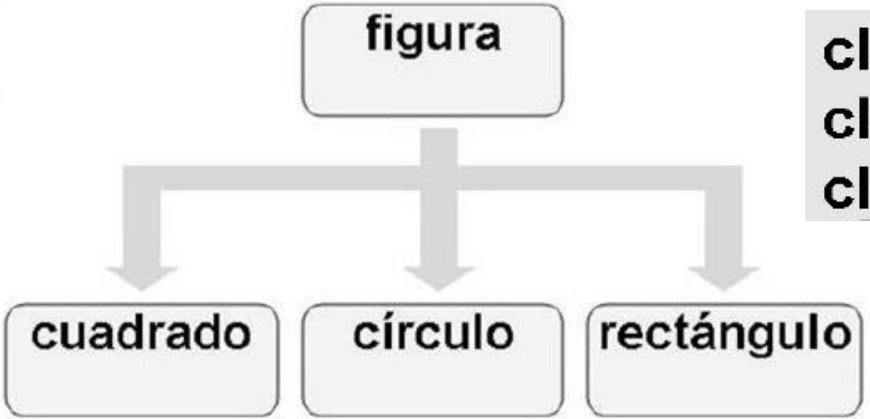
Ejemplo:





HERENCIA

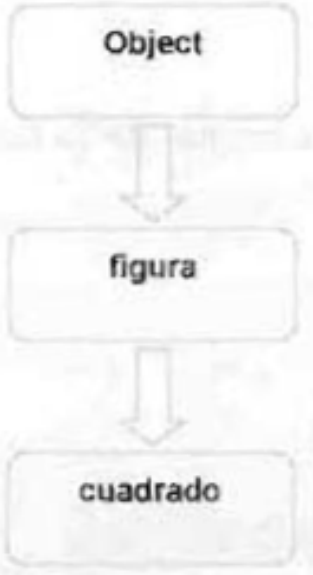
En este ejemplo vemos la herencia de la clase “figura”.
De ella heredarán otras 3 clases, “cuadrado, “circulo” y “rectangulo” lo que viene indicado por la palabra “**extends**”.




```
class rectangulo extends figura { ... }
class circulo extends figura { ... }
class cuadrado extends figura { ... }
```

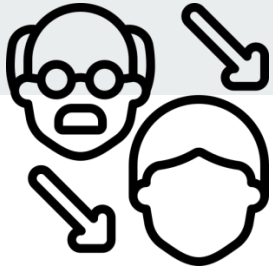
HERENCIA

Todas las clases tienen una superclase aunque no utilicemos la palabra “extends” en cuyo caso hereda de la clase “java.lang.Object”.
Volviendo al mismo ejemplo “figura” hereda de “Object”;




```
public class figura{
   String color;
   public void setColor(String s){color=s;}
   public String getColor(){return color;}
}
public class cuadrado extends figura{
   private int lado;
   cuadrado(int l){ this.lado = l; }
   public int getArea(){ return lado*lado; }
}
```

HERENCIA



Al utilizar la cláusula extends lo que indicamos es lo siguiente:

- La clase “cuadrado” es una subclase de la clase figura.
- La clase “cuadrado” puede utilizar los métodos de la clase figura aunque no están declarados en la clase cuadrado (siempre y cuando no estén como private de la clase figura).
- Obviamente, los métodos de la subclase no pueden ser utilizados en la superclase o clase principal.

Podemos probar este ejemplo con la siguiente clase (solo la clase “cuadrado” hace falta):

```
class testFiguras {
    public static void main(String[] args) {
        cuadrado c=new cuadrado(5);
        c.setColor("Verde");
        System.out.println(c.getColor());
        System.out.println(c.getArea());
    }
}
```

EJERCICIOS

1. Realiza una clase Temperatura, la cual convierte grados Celsius a Farenheit y viceversa. Se crearán 2 métodos celsiusToFarenheit y FarenheitTocelsius.
2. Con la siguiente clase coche añade los siguientes métodos:

   class coche{private int velocidad;coche(){velocidad=0;}}

   a. Int getVelocidad ()-> devuelve la velocidad actual
   b. Void acelera (int mas)-> este método actualiza la velocidad a más kilómetros más.
   c. Void frena (int menos) ->este método actualiza la velocidad a menos kilómetros menos.

1. Crea una clase Rebajas con un método descubrePorcentaje () que devuelva el descuento aplicado a un producto. El método recibe el precio original del producto y el rebajado y hay el porcentaje.