





```
1 class Asignatura{
2     public int identificador;
3     public double calificacion;
4     Asignatura(int param){
5         identificador=param;
6     }
7     public int getIdentificador(){
8         return identificador;
9     }
10    public double getCalificacion(){
11        return calificacion;
12    }
13    public void setCalificacion(double param){
14        calificacion=param;
15    }
16}
17 class Alumno{
18     private Asignatura asignatural=null;
19     private Asignatura asignatura2=null;
20     private Asignatura asignatura3=null;
21     public Alumno(Asignatura param1,Asignatura param2,Asignatura param3){
22         asignatural=param1;
23         asignatura2=param2;
24         asignatura3=param3;
25     }
26     public Alumno(int param1,int param2,int param3){
27         asignatural=new Asignatura(param1);
28         asignatura2=new Asignatura(param2);
29         asignatura3=new Asignatura(param3);
30     }
31     public AsignaturagetAsigntaural(){
32         return asignatural;
33     }
34     public AsignaturagetAsigntaura2(){
35         return asignatura2;
36     }
37     public AsignaturagetAsigntaura3(){
38         return asignatura3;
39     }
40 }
41 class Profesor{
42     public void ponerNotas(Alumno param){
43         param.getAsigntaural().setCalificacion(Math.random()*10);
44         param.getAsigntaura2().setCalificacion(Math.random()*10);
45         param.getAsigntaura3().setCalificacion(Math.random()*10);
46     }
47     public double calcularMedia(Alumno param){
48         double nota1=param.getAsigntaural().getCalificacion();
49         double nota2=param.getAsigntaura2().getCalificacion();
50         double nota3=param.getAsigntaura3().getCalificacion();
51         return (nota1+nota2+nota3)/3;
52     }
53 }
54 public class Ejercicio27{
55     public static void main(String args[]){
56         Asignatura programacion=new Asignatura(1);
57         Asignatura redes=new Asignatura(2);
58         Asignatura estadistica=new Asignatura(3);
```

(6.13.1) el método finalize

Es equivalente a los destructores del C++. Es un método que es llamado antes de eliminar definitivamente al objeto para hacer limpieza final. Un uso puede ser eliminar los objetos creados en la clase para eliminar referencias circulares. Ejemplo:

```
public class uno {  
    dos d;  
    public uno() {  
        d = new dos();  
    }  
    protected void finalize() {  
        d = null; //Se elimina d por lo que pudiera pasar  
    }  
}
```

finalize es un método de tipo **protected** heredado por todas las clases ya que está definido en la clase raíz **Object**.

La diferencia de **finalize** respecto a los métodos destructores de C++ estriba en que en Java no se llaman al instante (de hecho es imposible saber cuando son llamados). La llamada **System.gc()** llama a todos los finalize pendientes inmediatamente (es una forma de probar si el método finalize funciona o no).

Normalmente el método finalize no es necesario salvo en casos de objetos que utilicen archivos (que se deberán cerrar) u otros objetos que manipulen recursos del sistema.

Ejercicio n° 30

Crear una clase llamada Monedero, que permita gestionarlo.

La clase deberá tener un constructor que permite crear el monedero con una cantidad de dinero inicial y deberá de tener los métodos para meter dinero, sacarlo y consultar el disponible. evidentemente, no se podrá sacar mas dinero del disponible en ese momento.

Ejercicio n° 31

Hoja adjunta (pdf)

(34)

Ejercicio n° 32

Hoja adjunta (pdf)

En gran parte las clases estáticas solucionan algunos problemas que suceden al no disponer de variables y funciones globales. Es el caso de la función factorial comentada anteriormente. Es un método que es difícil de asociar a un objeto, por eso es estático, porque se asocia a una clase; en definitiva es fácil entender su uso como el de una función global de C.

En UML las propiedades y métodos estáticos se escriben en cursiva para distinguirlos de los otros.

Las variables estáticas no son muy habituales, pero sí las constantes (con el cualificador **final**). Hay que recordar que las constantes se escriben con todas las letras en mayúsculas

(6.12) el método main

Hasta ahora hemos utilizado el método **main** de forma incoherente como único posible mecanismo para ejecutar programas. De hecho este método dentro de una clase, indica que la clase es ejecutable desde la consola. Su prototipo es:

```
public static void main(String[] args){  
    ...instrucciones ejecutables....  
}
```

Hay que tener en cuenta que el método **main** es estático, por lo que no podrá utilizar atributos o métodos dinámicos de la clase.

(6.13) destrucción de objetos

En C y C++ todos los programadores saben que los objetos se crean con **new** (en realidad en C mediante la función **malloc**) y para eliminarlos de la memoria y así ahorrarla, se deben eliminar con la instrucción **delete** (en C con **free**).

Es decir, es responsabilidad del programador eliminar la memoria que gastaban los objetos que se van a dejar de usar. La instrucción **delete** del C++ llama al destructor de la clase, que es un método que se encarga de eliminar adecuadamente el objeto.

La sorpresa de los programadores C++ que empiezan a trabajar en Java es que **no hay instrucción delete en Java**. La duda está entonces, en cuándo se elimina la memoria que ocupa un objeto.

En Java hay un recolector de basura (**garbage collector**) que se encarga de gestionar los objetos que se dejan de usar y de eliminarlos de memoria. Este proceso es automático e impredecible y trabaja en un hilo (**thread**) de baja prioridad, por lo que apenas ralentiza al sistema.

Por lo general ese proceso de recolección de basura, trabaja cuando detecta que un objeto hace demasiado tiempo que no se utiliza en un

Ejemplo:

```
a[0]=new Punto(3,4);
a[1]=p; //suponiendo que p es una referencia válida aun punto,
          //a[1] hará tambien referencia a ese punto
a[2]=null;
...
```

Para usar las propiedades y métodos de los arrays de objetos:

```
a[0].mover(5,6);
System.out.println(a[0].getX());
```

En definitiva los arrays de objetos funcionan como cualquier otro array, sólo que hay que tener más precaución al usarlos por la cantidad de símbolos que manejan.

(6.11) métodos y propiedades genéricos (static)

Cuando se crean varios objetos de una misma clase, éstos poseen una copia propia de cada propiedad y método. Así si *e1* y *e2* son objetos de la clase *Empleado* lo más probable es que *e1.edad* y *e2.edad* sean valores diferentes, porque la edad de cada empleado es distinta.

Pero puede haber propiedades y métodos comunes a todos los objetos de una clase. En ese caso se consideran propiedades y métodos de la clase y no de los objetos. Comúnmente se les conoce como métodos y propiedades estáticas.

Por ejemplo si todos los empleados tuvieran el mismo salario mínimo entonces se definiría esa propiedad como estática. Y para manipularla se utilizaría el nombre de la clase y no de los objetos.

Así funciona la clase *Math* vista en temas anteriores. Para crear métodos o propiedades estáticas basta con colocar la palabra **static** antes del tipo del atributo o método. Ejemplo:

```
public class Calculadora {
    public static int factorial(int n) {
        int fact=1;
        while (n>0) {
            fact *=n--;
        }
        return fact;
    }
}
```

Ejemplo:

```
class Ficha {  
    private int casilla; //Valor inicial de la propiedad  
    public Ficha(){  
        this(1);  
    }  
    public Ficha(int n){ //constructor  
        casilla = n;  
    }  
    public void avanzar(int n){  
        casilla += n;  
    }  
    public int casillaActual(){  
        return casilla;  
    }  
}
```

La llamada `this(1)` invoca al segundo constructor pasándole como parámetro el valor `1`.

(6.9) comentarios Javadoc

Ya se han comentado la importancia de los comentarios Javadoc. Pero en el caso de las clases aún es mayor ya que documentar bien una clase y todos los métodos y atributos de la misma no debería ser una opción sino una obligación.

La razón, al utilizar clases ajenas es fundamental disponer de esta documentación. Así ocurre con las clases estándar, cuando se usan dentro de nuestro código podemos revisar el Javadoc para así conocer su funcionamiento.

Nosotros debemos hacer lo mismo y eso implica:

- ◆ Realizar un comentario Javadoc inmediatamente delante del nombre de la clase. En ese comentario se describe la clase. Es interesante en él utilizar los modificadores `@author` y `@version`.
- ◆ Escribir un comentario Javadoc inmediatamente delante de la declaración de cada atributo para describirle
- ◆ Escribir un Javadoc inmediatamente delante de cada método y constructor. En ese caso hay que utilizar al menos los modificadores:
 - `@param`. Para comentar cada uno de los parámetros del método o constructor. Se coloca un `@param` por cada parámetro.
 - `@return`. Para los métodos que devuelven valores, se usa para describir el valor de retorno

(6.8) constructores

Cuando se crea un objeto mediante el operador **new**, las propiedades toman un valor inicial. Ese valor puede ser los que Java otorga por defecto o bien los que se asignan a las propiedades si se utiliza la asignación en su declaración.

Pero esa forma de iniciar implícita es muy poco habitual. Es mucho más habitual crear un constructor para colocar esos valores. Un constructor es un método que se invoca cuando se crea un objeto y que sirve para iniciar los atributos del objeto y para realizar las acciones pertinentes que requiera el mismo para ser creado.

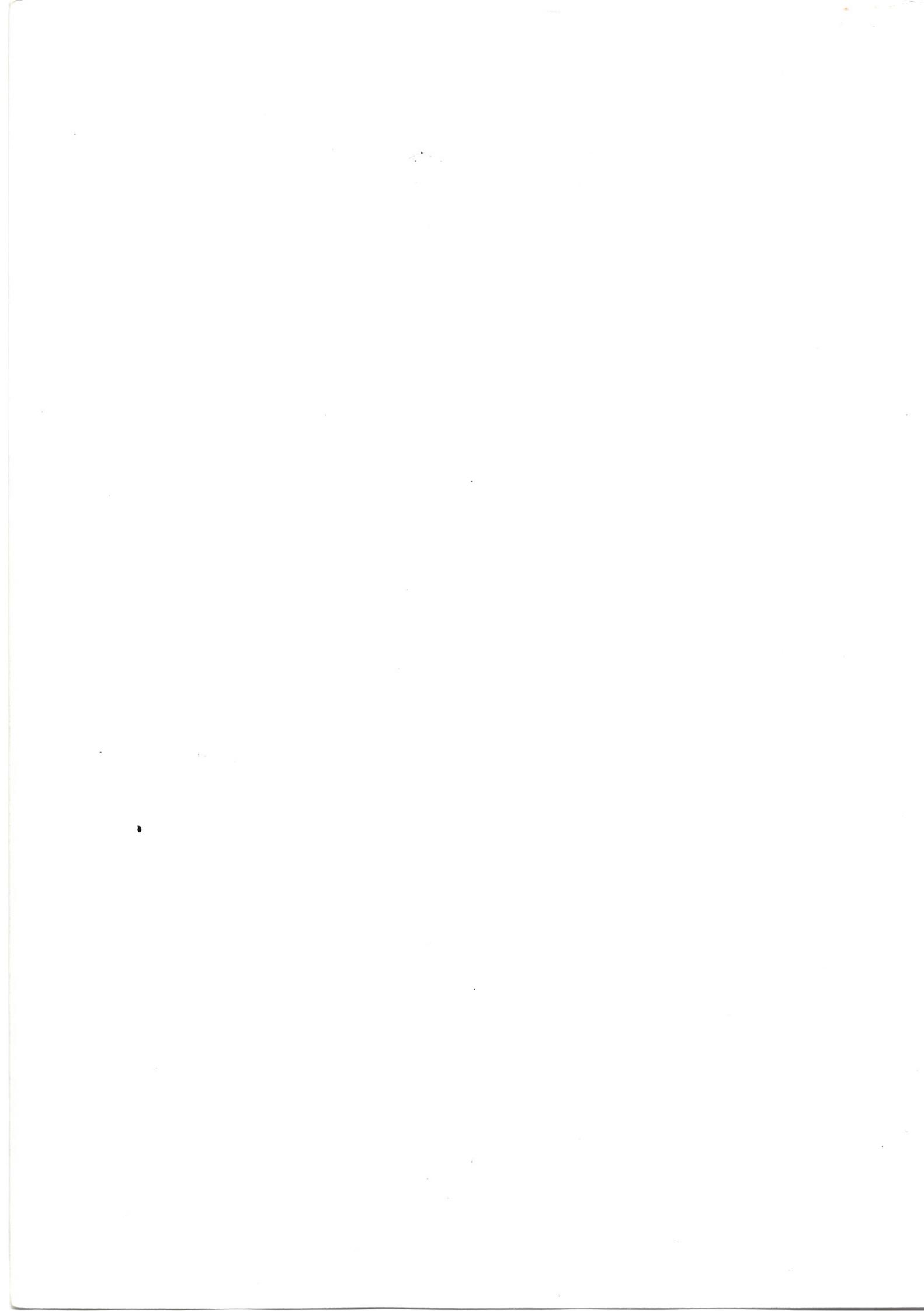
En realidad siempre hay al menos un constructor, el llamado **constructor por defecto**. Ese es un constructor que le crea el compilador en caso de que nosotros no creemos ninguno y se invoca cuando la creación es como:

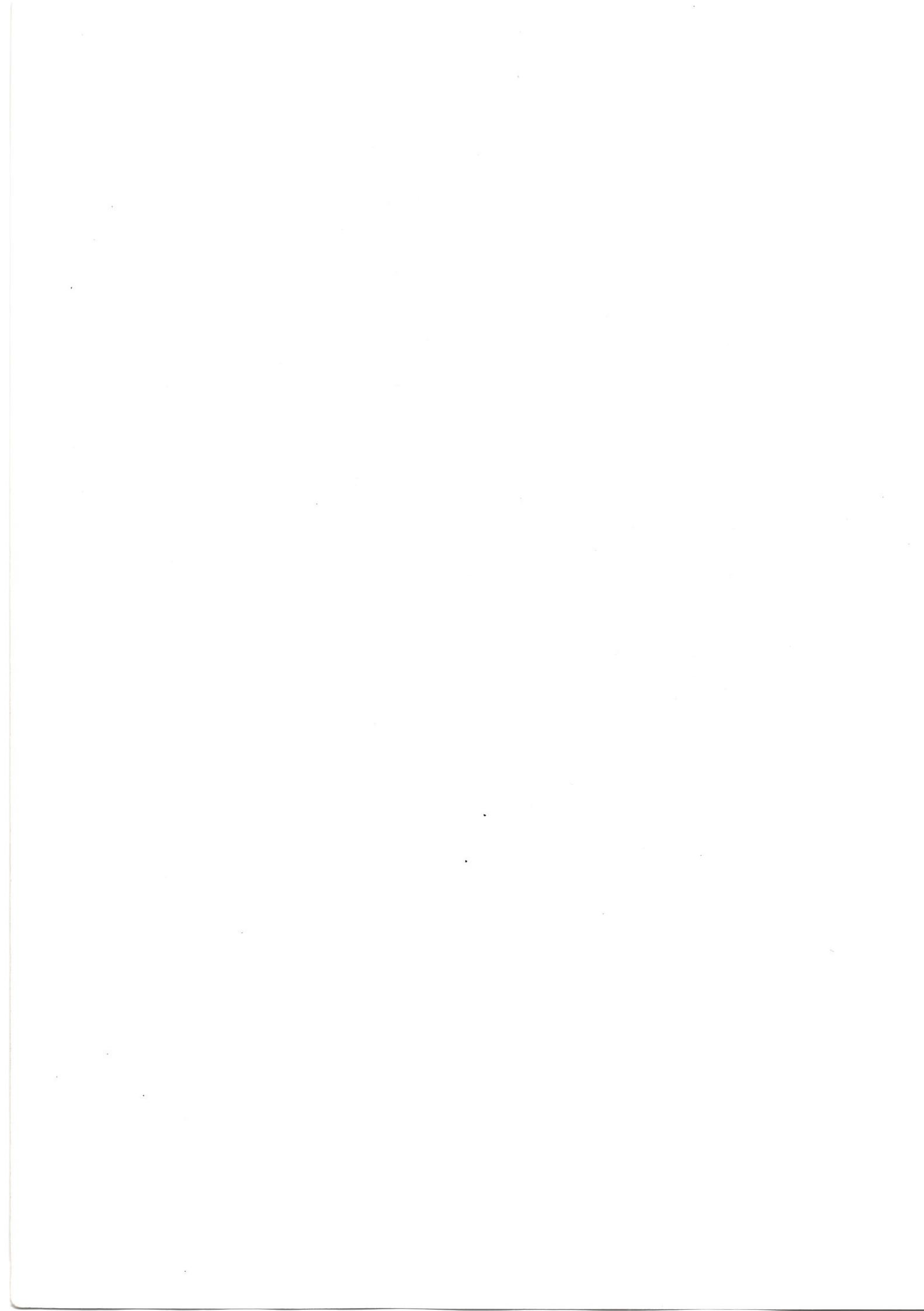
Persona p=new Persona();

Pero nosotros podemos crear nuestros propios constructores ya que un constructor no es más que un método que tiene el mismo nombre que la clase. Con lo cual para crear un constructor basta definir un método en el código de la clase que tenga el mismo nombre que la clase. Ejemplo:

```
public class Ficha {  
    private int casilla;  
  
    public Ficha() { //constructor  
        casilla = 1;  
    }  
  
    public void avanzar(int n) {  
        casilla += n;  
    }  
    public int casillaActual(){  
        return casilla;  
    }  
  
    public class App {  
        public static void main(String[] args) {  
            Ficha ficha1 = new Ficha();  
            ficha1.avanzar(3);  
            System.out.println(ficha1.casillaActual());//Da 4  
        }  
    }  
}
```

En la línea **Ficha ficha1 = new Ficha();** es cuando se llama al constructor, que es el que coloca inicialmente la casilla a 1.





Ejercicio nº 28)

Crear una clase que posea el método sobrecargado suma, de tal forma que se puedan sumar dos números, indicando el tipo de dato que se da:

recursividad cruzada

Hay que tener en cuenta la facilidad con la que la recursividad genera bucles infinitos. Por ello una función nunca ha de llamarse a sí misma si no estamos empleando la recursividad. Pero a veces estos problemas de recursividad no son tan obvios. Este código puede provocar una ejecución inacabable:

```
int a(){  
    int x=1;  
    x*=b();  
    return x;  
}  
  
int b(){  
    int y=19;  
    y-=a();  
    return y;  
}
```

Cualquier llamada a la función **a** o a la función **b** generaría código infinito ya que ambas se llaman entre sí sin parar jamás. A eso también se le llama recursividad, pero recursividad cruzada.

Ejercicio nº 29) Pdf

(6.7) la referencia this

Los objetos pueden hacer referencia a sí mismos, para ello disponen de la palabra **this**. Se utiliza dentro del código de las clases para obtener una referencia al objeto actual y permitir evitar ambigüedad y realizar llamadas a métodos que de otra forma serían complicadas. Ejemplo:

```
public class Punto {  
    int posX;  
    int posY;  
  
    void modificarCoords(int posX, int posY){  
        /* Hay ambigüedad ya que posX es el nombre de uno de  
         * los parámetros, y además el nombre de una de las  
         * propiedades de la clase Punto  
    }  
  
    this.posX=posX; //this permite evitar la ambigüedad  
    this.posY=posY;  
}
```

NOTA 8 BÁSICA

Dentro de una clase, para acceder a un método desde otro que no es el main o no es un método estático, basta con llamar al método directamente. No es necesario crear un objeto o instancia de la clase.

Ejemplo:

```
1 public class EjemploAcceso{
2     void saludoBienvenida(){
3         System.out.println("Estas en el metodo saludoBienvenida");
4         System.out.println("Bienvenido amig@");
5     }
6     void mostrarSaludo(){
7         System.out.println("Estas en el metodo mostrarSaludo");
8         saludoBienvenida();
9         System.out.println("Has vuelto al metodo mostrarSaludo");
10    }
11   public static void main(String args[]){
12       EjemploAcceso ea=new EjemploAcceso();
13       ea.mostrarSaludo();
14   }
15 }
16 }
```

Por consola:

Estas en el método mostrarSaludo

Estas en el método saludoBienvenida

Bienvenido amig@

Has vuelto al método mostrarSaludo

FIN DEL PROGRAMA

NOTA 5 BÁSICA

Para acceder a los métodos y variables de instancia definidos en una clase desde su método main se hace lo mismo que en la nota anterior, es decir, se crea un objeto de esa clase y mediante ese objeto se llama a los métodos y variables de instancia que interesen. No importa que sean privados.

Ejemplo:

```
1 public class CalculoCuadrado1{  
2     void mostrarMensaje(){  
3         System.out.println("Has calculado el cuadrado de un numero");  
4     }  
5     int calcularCuadrado(int numero){  
6         int cuadrado;  
7         cuadrado=numero*numero;  
8         return cuadrado;  
9     }  
10    public static void main(String args[]){  
11        CalculoCuadrado1 cc=new CalculoCuadrado1();  
12        int resultado=cc.calcularCuadrado(15);  
13        System.out.println(resultado);  
14        cc.mostrarMensaje();  
15        System.out.println("FIN DEL PROGRAMA");  
16    }  
17}
```

Por consola:

225

Has calculado el cuadrado de un numero

FIN DEL PROGRAMA

NOTA 6 BÁSICA

El anterior programa también se puede codificar utilizando únicamente el método main:

Ejemplo:

```
1 public class CalculoCuadrado2{  
2     public static void main(String args[]){  
3         int numero=15;  
4         int resultado=numero*numero;  
5         System.out.println(resultado);  
6         System.out.println("Has calculado el cuadrado de un numero");  
7         System.out.println("FIN DEL PROGRAMA");  
8     }  
9 }
```

Por consola:

225

Has calculado el cuadrado de un numero

FIN DEL PROGRAMA

```

1 public class Primera{
2     public static void main(String args[]){
3         System.out.println("Esta es la clase Primera");
4     }
5 }
6 class Segunda{
7     public static void main(String args[]){
8         System.out.println("Esta es la clase Segunda");
9     }
10}
11 class Tercera{
12     public static void main(String args[]){
13         System.out.println("Esta es la clase Tercera");
14     }
15}

```

El fichero donde se guarda el código Java debe llamarse "Primera.java" pues es la clase pública del código. El programador puede controlar qué clase va a ser la principal, es decir, qué clase va a contener el método main, desde la consola del DOS . El resto de métodos main no se tendrán en cuenta.

Si desde la consola, y después de compilar con "javac Primera.java" se ejecuta con:

java Primera =>Por consola: Esta es la clase Primera

Si java Segunda => Por consola: Esta es la clase Segunda

Si java Tercera => Por consola: Esta es la clase Tercera

NOTA 3 BÁSICA

Un código fuente Java no tiene por qué tener clase principal (aquella que contiene el método main o punto de entrada del programa).

Ocurrirá lo siguiente: se podrá compilar pero no ejecutar (error de ejecución). El programa carece de punto de entrada.

Se utilizan para recibir llamadas de otras clases, mediante la creación desde la clase llamante de un objeto de la clase a la que se invoca.

Ejemplo:

```

1 public class SinMetodoMain{
2     int cuadrado;
3     int calcularCuadrado(int numero){
4         cuadrado=numero*numero;
5         return cuadrado;
6     }
7 }

```

Es fundamental tener en cuenta cuándo la función debe dejar de llamarse a sí misma en algún momento, es decir es importante decidir cuándo acabar. De otra forma se corre el riesgo de generar infinitas llamadas, lo que bloquearía de forma grave el PC en el que trabajamos (aunque con Java este daño es muy limitado).

Un ejemplo de recursividad es este:

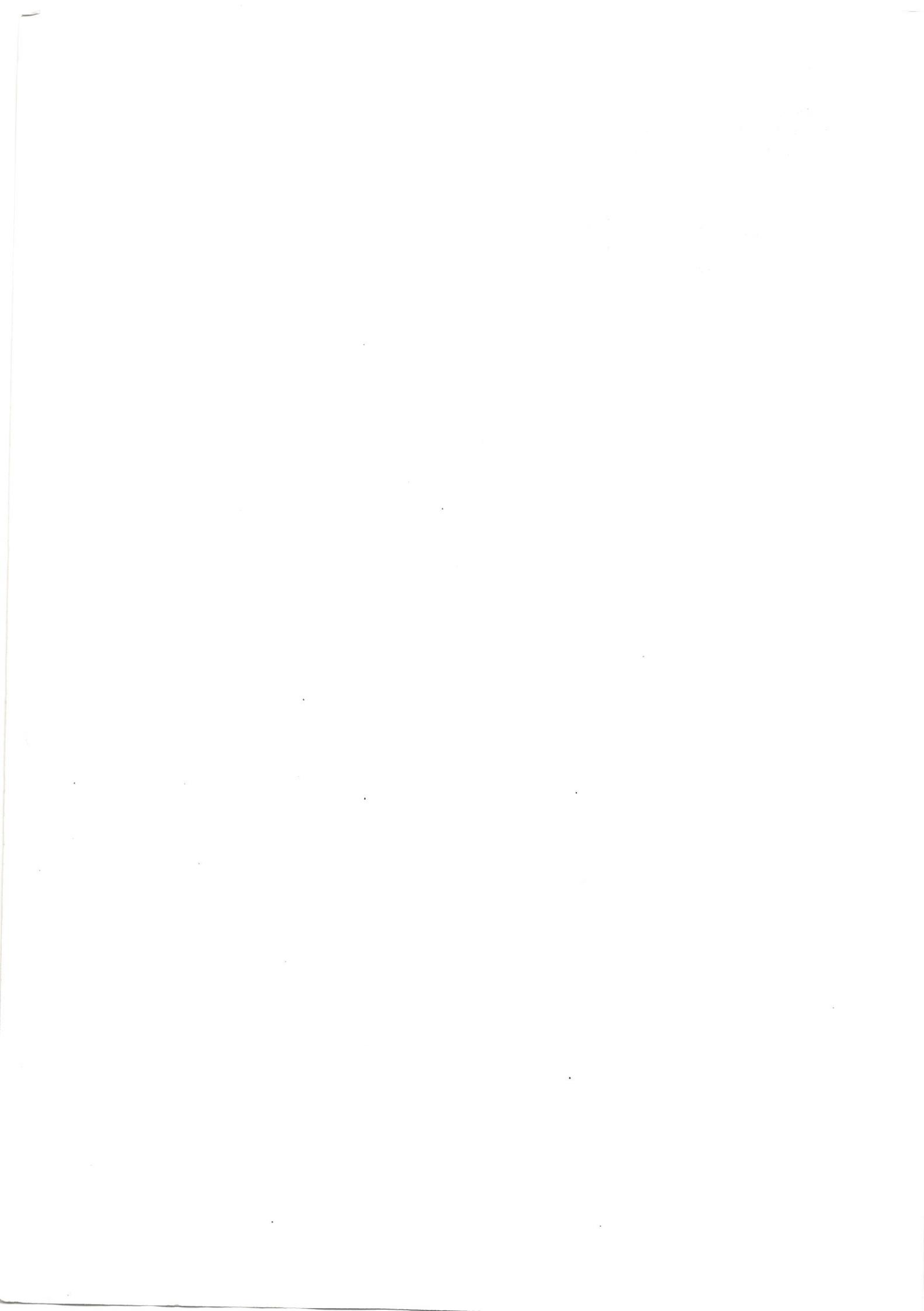
Como ejemplo vamos a ver la versión recursiva del factorial.

```
public class Matematicas{  
    public double factorial(int n){  
        if(n<=1) return 1;  
        else return n*factorial(n-1);  
    }  
}
```

La última instrucción (`return n*factorial(n-1)`) es la que realmente aplica la recursividad. La idea (por otro lado más humana) es considerar que el factorial de nueve es nueve multiplicado por el factorial de ocho; a su vez el de ocho es ocho por el factorial de siete y así sucesivamente hasta llegar al uno, que devuelve uno (es la instrucción que para la recursividad).

Con una llamada a ese método con `factorial(4)`; usando el ejemplo anterior, la ejecución del programa generaría los siguientes pasos:

- (1)** Se llama a la función factorial usando como parámetro el número 4 que será copiado en la variable-parámetro `n`
- (2)** Como `n>1`, entonces se devuelve 4 multiplicado por el resultado de la llamada `factorial(3)`
- (3)** La llamada anterior hace que el nuevo `n` (variable distinta de la anterior) valga 3, por lo que esta llamada devolverá 3 multiplicado por el resultado de la llamada `factorial(2)`
- (4)** La llamada anterior devuelve 2 multiplicado por el resultado de la llamada `factorial(1)`
- (5)** Esa llamada devuelve 1
- (6)** Eso hace que la llamada `factorial(2)` devuelva $2*1$, es decir 2
- (7)** Eso hace que la llamada `factorial(3)` devuelva $3*2$, es decir 6
- (8)** Por lo que la llamada `factorial(4)` devuelve $4*6$, es decir **24** Y ese es ya el resultado final



devolución de valores complejos

Como se ha visto anteriormente, los métodos pueden devolver valores básicos (**int**, **short**, **double**, etc.), **Strings**, **arrays** e incluso objetos.

En todos los casos es el comando **return** el que realiza esta labor. En el caso de arrays y objetos, devuelve una referencia a ese array u objeto. Ejemplo:

```
public class FabricaArrays {  
    public int[] obtenArray(){  
        int array[] = {1,2,3,4,5};  
        return array;  
    }  
  
public class ReturnArray {  
    public static void main(String[] args) {  
        FabricaArrays fab=new FabricaArrays();  
        int nuevoArray[] = fab.obtenArray(); //se obtiene la referencia  
                                         //al array  
    }  
}
```

métodos set y get

En Java cada vez es más habitual la práctica de ocultar las propiedades y trabajar exclusivamente con los métodos. La razón es que no es recomendable que las propiedades sean visibles desde fuera de la clase, por ello se declaran con una visibilidad **private** (o **protected**).

Siendo así ¿cómo pueden las otras clases modificar el valor de una propiedad? Mediante métodos que permitan la lectura y escritura de esas propiedades, son los métodos **get** (obtener) y **set** (ajustar).

Los **get** sirven para leer el valor de un atributo, nunca llevan parámetros y el nombre del método es la palabra **get** seguida del nombre de la propiedad (por ejemplo **getEdad**).

Los **set** permiten variar el valor de una propiedad. Por lo que como parámetro reciben el nuevo valor y son métodos siempre de tipo **void**.

Ejemplo:

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre=nombre;  
    }
```

argumentos por valor y por referencia

En todos los lenguajes éste es un tema muy importante. Los argumentos son los datos que recibe un método y que necesita para funcionar. Ejemplo:

```
public class Matemáticas {  
    public double factorial(int n){  
        double resultado;  
        for (resultado=n;n>1;n--) resultado*=n;  
        return resultado;  
    }  
    ...  
    public static void main(String args[]){  
        Matemáticas m1=new Matemáticas();  
        double x=m1.factorial(25);//Llamada al método  
    }  
}
```

En el ejemplo anterior, el valor 25 es un argumento requerido por el método **factorial** para que éste devuelva el resultado (que será el factorial de 25). En el código del método factorial, este valor 25 es copiado a la variable **n**, que es la encargada de almacenar y utilizar este valor.

Se dice que los argumentos son por valor, si la función recibe una copia de esos datos, es decir la variable que se pasa como argumento no estará afectada por el código. Ejemplo:

```
class Prueba {  
    public void metodo1(int entero){  
        entero=18;  
    }  
    ...  
    public static void main(String args[]){  
        int x=24;  
        Prueba miPrueba = new Prueba();  
        miPrueba.metodo1(x);  
        System.out.println(x);//Escribe 24, no 18  
    }  
}
```

Este es un ejemplo de paso de parámetros por valor. La variable **x** se envía como argumento o parámetro para el método **metodo1**, allí la variable **entero** recibe **una copia** del **valor** de **x** en la variable **entero**, y a esa copia se le asigna el valor 18. Cuando el código del **metodo1** finaliza, la variable **entero** desaparece, en todo este proceso la variable **x** no ha sido afectada, seguirá valiendo 24.

(6.6.2) métodos

Un método es una llamada a una operación de un determinado objeto. Al realizar esta llamada (también se le llama enviar un mensaje), el control del programa pasa a ese método y lo mantendrá hasta que el método finalice. La mayoría de métodos devuelven un resultado (gracias a la palabra **return**), por ello cuando se define el método hay que indicar el tipo de datos al que pertenece el resultado del mismo.

Si el método no devuelve ningún resultado se indica como tipo de datos a devolver el tipo **void** (void significa vacío).

Los métodos son los equivalentes de las funciones de la programación modular clásica. De hecho un método es una función, sólo que esa función está asociada a una clase, por lo que se convierte en una operación que esa clase es capaz de realizar. Pero las personas que saben programar funciones, no tendrán mucho problema en entender cómo se programan los métodos (salvo la dificultad que encontrarán en adaptarse a la programación orientada a objetos)

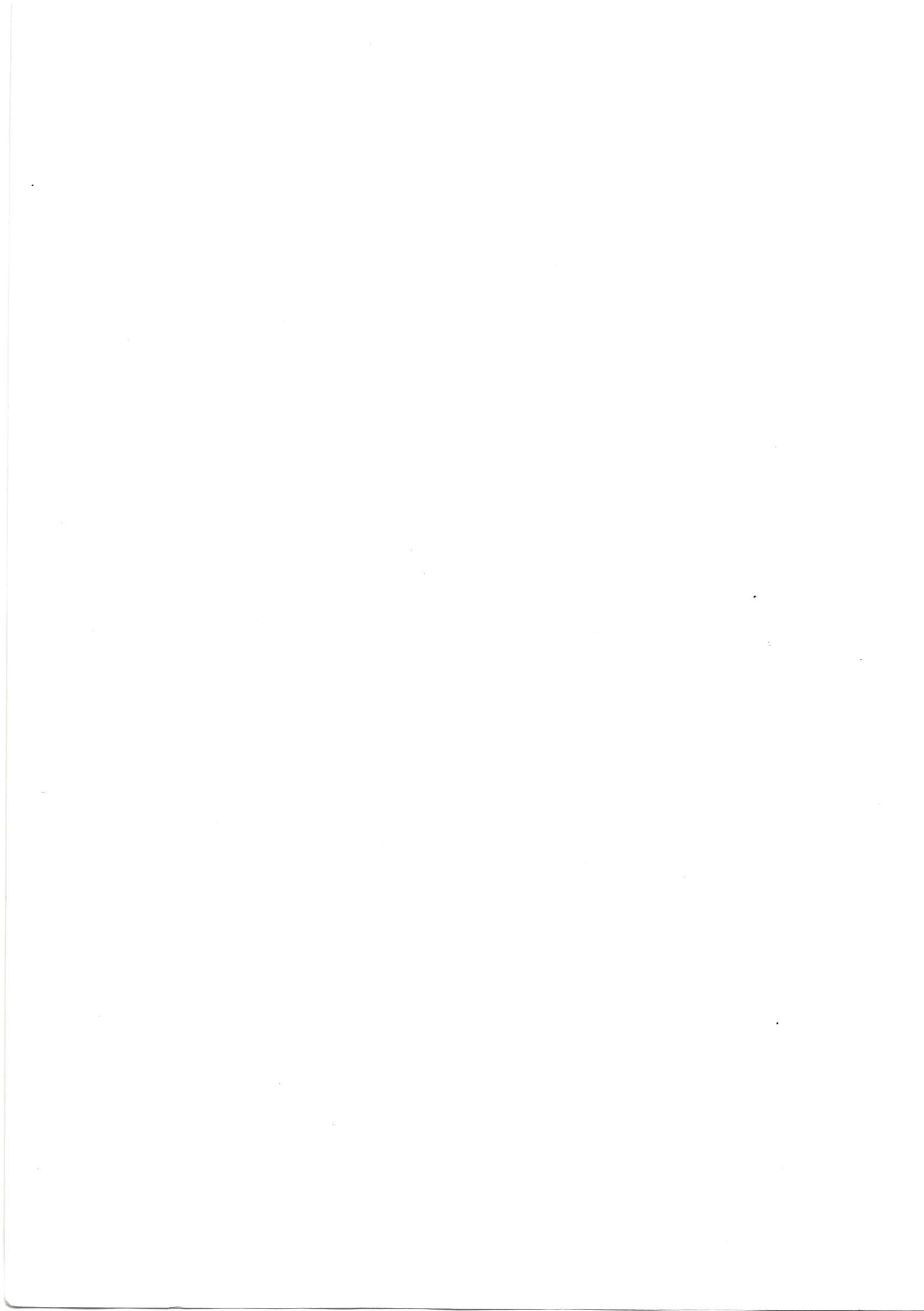
Cuando una clase ya tiene definido sus métodos, es posible invocarlos utilizando los objetos definidos de esa clase. En esa invocación, se deben indicar los **parámetros** (o **argumentos**) que cada método requiere para poder realizar su labor. Ejemplos de uso de métodos:

```
balón.botar(); //sin argumentos  
miCoche.acelerar(10);  
ficha.comer(posición15); //posición 15 es una variable que se  
//pasa como argumento  
partida.empezarPartida("18:15",colores);
```

definir métodos

Es importante entender perfectamente la función de un método. Un método no es más que un código que realiza una determinada operación en un objeto. Para ello necesitamos definir:

- (1) Sus especificadores de alcance o visibilidad.** Si el alcance es privado, el método sólo se podrá utilizar dentro de otro método en la misma clase; si el público podrá ser invocado desde cualquier clase; si es protegido desde la propia clase y sus descendientes y si es amigable, desde clases que estén en el mismo paquete.
- (2) El tipo de datos o de objeto que devuelve.** Si el resultado del método es un número entero, o un booleano, o un String o un objeto de una clase determinada, etc. Si el método no devuelve valor alguno se debe indicar como tipo el valor **void**.
- (3) El identificador del método.** Cumple las mismas reglas que los identificadores de variable y también deben empezar una letra en minúscula



```

        for (int i=1; i<=20; i++)
            System.out.println("Factorial de " + i + " = " + factorialR(i));
    }
public static long factorialR (int n) {
    if (n==0) return 1;
    else return n * factorialR(n-1);
}
}

```

Ejemplo de salida por pantalla:

```

$>java PruebaFactorialR
Factorial de 1 = 1
Factorial de 2 = 2
Factorial de 3 = 6
Factorial de 4 = 24
Factorial de 5 = 120
Factorial de 6 = 720
Factorial de 7 = 5040
Factorial de 8 = 40320
Factorial de 9 = 362880
Factorial de 10 = 3628800
Factorial de 11 = 39916800
Factorial de 12 = 479001600
Factorial de 13 = 6227020800
Factorial de 14 = 87178291200
Factorial de 15 = 1307674368000
Factorial de 16 = 20922789888000
Factorial de 17 = 355687428096000
Factorial de 18 = 6402373705728000
Factorial de 19 = 121645100408832000
Factorial de 20 = 2432902008176640000
$>

```

Como ya se ha visto en un ejemplo anterior, el factorial de un número crece muy deprisa con el operando. Para evitar problemas a la hora de calcular el factorial de enteros superiores a 12 se ha vuelto a indicar el tipo `long` como tipo de dato de retorno del método estático `factorialR`.

En la construcción de métodos recursivos es importante evitar el problema de la recursión infinita. Es decir, que en algún caso, la ejecución del método definido de forma recursiva no implique una nueva llamada al propio método.

1.4. Sobrecarga de métodos

Java permite asignar el mismo identificador a distintos métodos, cuya diferencia reside en el tipo o número de parámetros que utilicen. Esto resulta especialmente conveniente cuando se desea llevar a cabo la misma tarea en diferente número o tipos de variables. La *sobrecarga (overloading)* de los métodos puede resultar muy útil al efectuar llamadas a un método, ya que en lugar de tener que recordar identificadores de métodos distintos, basta con recordar uno sólo. El compilador se encarga de averiguar cuál de los métodos que comparten identificador debe ejecutar. Por ejemplo:

```

/**
 * Demostracion de metodos sobrecargados
 * A. Garcia-Beltran - marzo, 2002
 */
public class PruebaSobrecarga {
    public static void main (String[] args) {
        int a=34;

```

```

public static void main (String [] args){
    tabla(4);                                // ejemplo de llamada
    tabla(7);
}
public static void tabla (int n) {           // de tipo void
    System.out.println("Tabla de multiplicar del numero " + n);
    for (int i=0; i<=10; i++)
        System.out.println(n + " x " + i + " = " + producto(n,i));
    return;                                    // No devuelve ningun valor
}
public static int producto (int a, int b) {
    return a*b;
}
}

```

Ejemplo de ejecución y salida correspondiente por pantalla:

```

$>java PruebaTabla
Tabla de multiplicar del numero 4
4 x 0 = 0
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40
Tabla de multiplicar del numero 7
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70

```

Si no hay sentencia `return` dentro de un método, su ejecución continúa hasta que se alcanza el final del método y entonces se devuelve la secuencia de ejecución al lugar donde se invocó al método.

Un método cuyo tipo de retorno no es `void` necesita siempre devolver algo. Si el código de un método contiene varias sentencias `if` debe asegurarse de que cada una de las posibles opciones devuelve un valor. En caso contrario, se generaría un error de compilación. Por ejemplo:

```

/**
 * Demostracion de la funcion esPositivo
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaPositivo {
    public static void main (String [] args) {
        for (int i=5; i>=-5; i--)
            System.out.println(i + " es positivo: " + esPositivo(i));
    }
}

```

```

15! = 2004310016
16! = 2004189184
17! = -288522240
18! = -898433024
19! = 109641728
20! = -2102132736
$>

```

Como se puede observar, el valor del factorial calculado para enteros superiores a 12 es incorrecto. El motivo es que el valor del factorial de un número crece muy deprisa con el operando. Así 12! es 479.001.600, cerca ya del límite superior del intervalo de representación del tipo primitivo int (2.147.483.647). El factorial de 13 (6.227.020.800) supera este límite y genera un error de cálculo por desbordamiento (*overflow*) del intervalo de representación. El problema se puede solventar (de nuevo limitadamente) utilizando un tipo primitivo con un intervalo de representación numérica mayor, por ejemplo, el tipo long:

```

/**
 * Demostracion de la funcion factorial
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaTablaFactorial2 {
    public static void main (String [] args) {
        for (int k=1; k<=24; k++)
            System.out.println(k + "!" + factorial(k));
    }
    public static long factorial (int n) {
        long aux = 1;
        for (int i = 2; i<=n; i++)
            aux *= i;                                // aux = aux * i
        return aux;
    }
}

```

El resultado de la ejecución del programa anterior por pantalla es el siguiente:

```

$>java PruebaTablaFactorial2
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = -4249290049419214848
22! = -1250660718674968576
23! = 8128291617894825984
24! = -7835185981329244160

```

```

/**
 * Demostracion del metodo cubo
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaCubo {
    public static void main (String [] args) {
        System.out.println("El cubo de 7.5 es: " + cubo(7.5)); // llamada
    }
    public static double cubo (double x) { // declaracion
        return x*x*x;
    }
}

```

A diferencia de otros lenguajes de programación, como Pascal, en Java, la declaración del método puede realizarse en el código fuente después de la llamada al propio método. En el caso anterior, `public` y `static` son los modificadores especificados en la cabecera del método. El uso de estos dos modificadores permite que el tipo de método sea similar al de una función global de Pascal o C. El identificador `double` hace referencia al tipo de dato que devuelve la llamada al método, `cubo` es el identificador del método y `x` es el identificador del parámetro en la declaración de la cabecera del método (parámetro *formal*). Ejemplo de ejecución del código anterior y salida correspondiente por pantalla:

```
$>java PruebaCubo
El cubo de 7.5 es: 421.875
```

En Java, los métodos suelen ir asociados con los objetos o instancias en particular a los que operan (*métodos de instancia*). Los métodos que no necesitan o trabajan con objetos (y sí con números, por ejemplo) se denominan métodos *estáticos* o *de clase* y se declaran con el modificador `static`. Los métodos estáticos o de clase son equivalentes a las rutinas (funciones o procedimientos) de los lenguajes que no emplean la programación orientada a objetos. Por ejemplo, el método `sqrt` de la clase `Math` es un método estático. También lo es el método `cubo` del ejemplo anterior. Por otro lado, todo programa o aplicación Java (que no sea un *applet*) tiene un método principal `main` que será siempre un método *estático*.

¿Por qué se emplea la palabra `static` para los métodos de clase?. El significado o la acepción más común de la palabra *estático* (*que permanece quieto en un lugar*) parece no tener nada que ver con lo que hacen los métodos *estáticos*. Java emplea la palabra `static` porque C++ lo utiliza en el mismo contexto: para designar métodos *de clase*. Aprovechando su empleo en variables que tienen una única localización en memoria para diferentes llamadas a métodos, C++ lo empezó a utilizar en la designación de los métodos *de clase* para diferenciarlos de los métodos *de instancia* y no confundir al compilador. El problema es que nadie pensó en que el uso de la palabra `static` pudiera causar confusiones *humanas*.

Un ejemplo un poco más complicado

Las sentencias incluidas en el cuerpo del método no tienen porqué reducirse a una única sentencia `return`. Por ejemplo, en el siguiente código se introduce la declaración del método *estático* `factorial` que devuelve el factorial de un valor entero `n` dado como parámetro o argumento. Dentro del método `factorial` se declara localmente la variable `aux` de tipo `int` y se incluye una sentencia `for`. El factorial, $n!$, se define como el producto de $1 \cdot 2 \cdot 3 \cdots \cdot (n-1) \cdot n$ cuando `n` es mayor que 1, siendo $1! = 1$.

```
/**
```

Puede parecer sorprendente que el radio de la noria *n1* sea siete en la última línea, ya que el radio que se ha cambiado es de *n2*. La cuestión es que *n1* y *n2* son referencias a la misma *noria*. Cuando se realiza la instrucción *n2=n1* lo que ocurre no es que *n2* sea una copia de *n1*, sino que *n2* será una referencia a *n1*. Es decir cuando se cambia *n2*, se cambia *n1* y viceversa.

Evidentemente entre objetos, sólo se puede utilizar la asignación entre objetos de la misma clase.

(6.5) modificadores de acceso

Se trata de una palabra que antecede a la declaración de una clase, método o propiedad de clase. Hay tres posibilidades: **public**, **protected** y **private**. Una cuarta posibilidad es no utilizar ninguna de estas tres palabras; entonces se dice que se ha utilizado el modificador por defecto (**friendly**).

Los especificadores determinan el alcance de la visibilidad del elemento al que se refieren. Referidos por ejemplo a un método, pueden hacer que el método sea visible sólo para la clase que lo utiliza (**private**), para éstas y las heredadas (**protected**), para todas las clases del mismo paquete (**friendly**) o para cualquier clase del tipo que sea (**public**).

En la siguiente tabla se puede observar la visibilidad de cada especificador:

zona	private (privado)	sin modificador (friendly)	protected (protegido)	public (público)
Misma clase	X	X	X	X
Subclase en el mismo paquete		X	X	X
Clase (no subclase) en el mismo paquete		X	X	X
Subclase en otro paquete			X	X
No subclase en otro paquete				X

El modificador debe de indicarse antes de indicar el tipo de datos de la propiedad o el método. Por ejemplo:

```
public class Noria {  
    private double radio;  
    public void girar(int velocidad){  
        ...//definición del método  
    }  
    public void parar(){...  
}
```

El modificador **friendly** en realidad es la visibilidad que se utiliza cuando no se indica ningún modificador de acceso. modificador **protected** (que se explica

Más adelante se explica en detalle el uso de **new**.

(6.4.2) acceso a las propiedades del objeto

Para poder acceder a los atributos de un objeto, se utiliza esta sintaxis:

objeto.atributo

Por ejemplo:

miNoria.radio

Para asignar valores a una propiedad, podemos hacerlo así:

miNoria.radio=9;

(6.4.3) métodos

Los métodos se utilizan de la misma forma que los atributos, excepto porque los métodos poseen siempre paréntesis, dentro de los cuales pueden ir valores necesarios para la ejecución del método (parámetros):

objeto.método(argumentosDelMétodo)

Los métodos siempre tienen paréntesis (es la diferencia con las propiedades) y dentro de los paréntesis se colocan los argumentos del método. Que son los datos que necesita el método para funcionar. Por ejemplo:

miNoria.gira(5);

Lo cual podría hacer que la Noria avance a 5 Km/h.

(6.4.4) herencia

En la POO tiene mucha importancia este concepto, la herencia es el mecanismo que permite crear clases basadas en otras existentes. Se dice que esas clases *descienden* de las primeras. Así por ejemplo, se podría crear una clase llamada **vehículo** cuyos métodos serían *mover*, *parar*, *acelerar* y *frenar*. Y después se podría crear una clase **coche** basada en la anterior que tendría esos mismos métodos (les heredaría) y además añadiría algunos propios, por ejemplo *abrirCapó* o *cambiarRueda*.

La herencia se trata en profundidad en el tema siguiente.

(6.4.5) representación de objetos

Cuando se crea un objeto se dice que se está *instanciando* una clase (del inglés *instantiate*). Por eso se dice que entre las clases y los objetos hay una relación de tipo *instance*.

El formato general para crear una clase en Java es:

```
[acceso] class nombreDeClase {  
    [acceso] [static] tipo atributo1;  
    [acceso] [static] tipo atributo2;  
    [acceso] [static] tipo atributo3;  
    ...  
    [access] [static] tipo nombreMétodo1([listaDeArgumentos]) {  
        ...código del método...  
    }  
    [access] [static] tipo nombreMétodo2([listaDeArgumentos]) {  
        ...código del método...  
    }  
    ...  
}
```

Recordar que en cuando ese escriben sintaxis el color resaltado en azul se usa en palabras reservadas (las palabras que obligatoriamente hay que escribir exactamente así), lo que está en color normal es el tipo de elemento que hay que escribir, el texto entre corchetes significa que no es obligatorio, que se pone o no dependiendo de las circunstancias o necesidades. Los puntos suspensivos indican que ese código se puede repetir más veces (en el anterior por ejemplo que se pueden declarar más atributos y más métodos).

La palabra opcional **static** sirve para hacer que el método o la propiedad a la que precede se pueda utilizar de manera genérica, los métodos o propiedades así definidos se llaman **atributos de clase** y **métodos de clase** respectivamente. Su uso se verá más adelante. Ejemplo:

```
class Noria {  
    double radio;  
    void girar(int velocidad){  
        ...//definición del método  
    }  
    void parar(){...  
}
```

Noria
radio:double
girar(velocidad:int)
parar()

Ilustración 6-2, La clase Noria en notación UML de diagrama de clases

empieza una partida para cuatro jugadores, `partida.empezar(rojo, azul)` empieza una partida de dos jugadores para los colores rojo y azul; estas son dos formas distintas de emplear el método `empezar`, por lo tanto este método es polimórfico. Esto simplifica la programación y reutilización de clases.

- ◆ **Herencia.** Mediante la POO podemos definir clases que utilicen métodos y propiedades de otras (que hereden dichos métodos y propiedades). De esta forma podemos establecer organizaciones jerárquicas de objetos. La herencia se trata en la unidad siguiente.

(6.2) introducción al concepto de objeto

Un objeto es cualquier entidad representable en un programa informático, bien sea real (`ordenador`) o bien sea un concepto (`transferencia`). Un objeto en un sistema posee: una **identidad**, un **estado** y un **comportamiento**.

El **estado** marca las condiciones de existencia del objeto dentro del programa. Lógicamente este estado puede cambiar. Un coche puede estar parado, en marcha, estropeado, funcionando, sin gasolina, etc. El estado lo marca el valor que tengan las propiedades del objeto.

El **comportamiento** determina como responde el objeto ante peticiones de otros objetos. Por ejemplo un objeto conductor puede lanzar el mensaje `arrancar` a un coche. El comportamiento determina qué es lo que hará el objeto (es decir, qué ocurre cuando se arranca el coche).

La **identidad** es la propiedad que determina que cada objeto es único aunque tenga el mismo estado. No existen dos objetos iguales. Lo que sí existen son dos **referencias** al mismo objeto

Los objetos se manejan mediante referencias. Cuando se crea un objeto necesitamos una referencia al mismo, de modo que esa referencia permitirá cambiar los atributos del objeto o invocar a los métodos de dicho objeto.

Puede haber varias referencias al mismo objeto, de modo que si una referencia cambia el estado del objeto, el resto (lógicamente) mostrarán esos cambios.

(6.3) clases

Las clases son las plantillas para hacer objetos. Una clase sirve para definir una serie de objetos con propiedades (**atributos**), comportamientos (**operaciones o métodos**), y semántica comunes. Hay que pensar en una clase como un molde. A través de las clases se obtienen los objetos en sí.

Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

- ◆ **El nombre o identificador de clase.** Debe empezar con letra mayúscula y seguirle letras minúsculas (si el nombre se compone de varias palabras, la inicial de cada palabra se deja en mayúscula). El

(6.1) introducción a la programación orientada a objetos

(6.1.1) qué es la programación orientada a objetos

A Java se le considera un lenguaje totalmente orientado a objetos. Aunque este **totalmente** es a veces criticado por que hay elementos de Java que no cumplen estrictamente las normas orientadas a objetos (como los tipos básicos o la clase **String** por ejemplo), desde luego es un lenguaje en el que desde el primer momento se manejan clases y objetos. De hecho siempre que se crea un programa en Java, por simple que sea, se necesita declarar una clase (con el **public class** correspondiente). El concepto de clase pertenece a la programación orientada a objetos.

Es decir; en Java no se pueden crear aplicaciones que no sean orientadas a objetos.

La Programación Orientada a Objetos (**POO**) es una técnica de programar aplicaciones ideada en los años setenta y que ha triunfado desde los ochenta, de modo que actualmente es el método habitual de creación de aplicaciones.

La **programación estructurada** impone una forma de escribir código que potencia la legibilidad del mismo. Cuando un problema era muy grande, aún que el código es legible, ocupa tantas líneas que al final le hacen inmanejable.

La **programación modular** supuso un importante cambio, ya que el problema se descompone en módulos (en muchos lenguajes llamados funciones) de forma que cada uno se ocupa de una parte del problema. Cuanto más independiente sea cada módulo, mejor permitirá el mantenimiento de la aplicación y el trabajo en equipo. Ya que cada persona puede programar cada módulo sin tener en cuenta cómo se han programado los otros.

Pero nuevamente esta técnica se queda corta. Al final realmente los módulos necesitan datos globales conocidos por todos, lo que resta independencia a los mismos. Además los datos y las funciones se manejan de forma distinta e independiente, lo que provoca problemas.

Con la **POO** se intenta solucionar esta limitación ya que el problema se dividen en **objetos**, de forma que cada objeto funciona de forma totalmente independiente. Un objeto es un elemento del programa que integra sus propios datos y su propio funcionamiento. Es decir un objeto está formado por **datos (propiedades)** y por las funciones que es capaz de realizar el objeto (**métodos**).

Esta forma de programar se asemeja más al pensamiento humano. La cuestión es detectar adecuadamente los **objetos** necesarios para una aplicación. De hecho hay que detectar las distintas **clases** de objetos.

Una clase es lo que define a un tipo de objeto. Al definir una clase lo que se hace es indicar como funciona un determinado tipo de objetos. Luego, a partir de la clase, podremos crear objetos de esa clase.

Realmente la programación orientada a objetos es una programación orientada a clases. Es decir lo que necesitamos programar es como funcionan



