objetos y clases

programación orientada a objetos

Se ha comentado anteriormente en este manual que Java es un lenguaje totalmente orientado a objetos. De hecho siempre hemos definido una clase pública con un método **main** que permite que se pueda visualizar en la pantalla el programa Java.

La gracia de la POO es que se hace que los problemas sean más sencillos, al permitir dividir el problema. Está división se hace en objetos, de forma que cada objeto funcione de forma totalmente independiente. Un objeto es un elemento del programa que posee sus propios datos y su propio funcionamiento.

Es decir un objeto está formado por datos (**propiedades**) y funciones que es capaz de realizar el objeto (**métodos**).

Antes de poder utilizar un objeto, se debe definir su **clase.** La clase es la definición de 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.

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc., etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearía tantos objetos casilla como casillas tenga el juego.

Lo mismo ocurriría con las fichas, la clase **ficha** definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc., etc.,), luego se crearían tantos objetos ficha, como fichas tenga el juego.

propiedades de la POO

- **Encapsulamiento**. Una clase se compone tanto de variables (propiedades) como de funciones y procedimientos (métodos). De hecho no se pueden definir variables (ni funciones) fuera de una clase (es decir no hay variables *globales*).

- **Ocultación**. Hay una zona oculta al definir la clases (zona privada) que sólo es utilizada por esa clases y por alguna clase relacionada. Hay una zona pública (llamada también **interfaz** de la clase) que puede ser utilizada por cualquier parte del código.

- **Polimorfismo**. Cada método de una clase puede tener varias definiciones distintas. En el caso del parchís: partida.empezar(4) 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, que es polimórfico.

- **Herencia**. Una clase puede heredar propiedades de otra.

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 **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.

La **identidad** determina que cada objeto es único aunque tengan el mismo valor. No existen dos objetos iguales. Lo que sí existe es dos referencias al mismo objeto.

Los objetos se manejan por referencias, existirá una referencia a un objeto. De modo que esa referencia permitirá cambiar los atributos del objeto. Incluso puede haber varias referencias al mismo objeto, de modo que si una referencia cambia el estado del objeto, el resto (lógicamente) mostrarán esos cambios.

Los objetos por valor son los que no usan referencias y usan copias de valores concretos. En Java estos objetos son los tipos simples: **int, char, byte, short, long, float, double** y **boolean.** El resto son todos objetos (incluidos los arrays y Strings).

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:

- **Sus atributos.** Es decir, los datos miembros de esa clase. Los datos pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase. También se las llama campos.
- **Sus métodos.** Las funciones miembro de la clase. Son las acciones (u operaciones) que puede realizar la clase.
- **Código de inicialización**. Para crear una clase normalmente hace falta realizar operaciones previas (es lo que se conoce como el constructor de la clase).
- **Otras clases**. Dentro de una clase se pueden definir otras clases (clases internas, son consideradas como asociaciones dentro de UML).

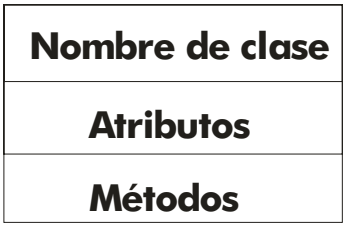

Ilustración 5, Clase en notación UML

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 método1(listaDeArgumentos) {
         ...código del método...
   }
   ...
}
```

La palabra opcional **static** sirve para hacer que el método o la propiedad a la que precede se pueda utilizar de manera genérica (más adelante se hablará de clases genéricas), 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(){...
}
```

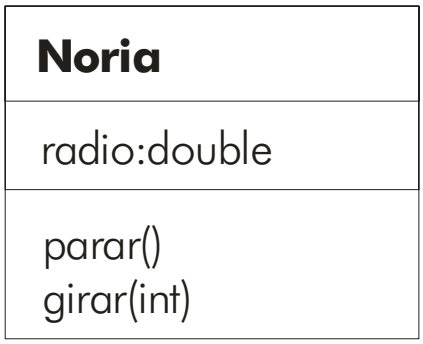

Ilustración 6, Clase Noria bajo notación UML

objetos

Se les llama **instancias de clase**. Son un elemento en sí de la clase (en el ejemplo del parchís, una ficha en concreto). Un objeto se crea utilizando el llamado **constructor** de la clase. El constructor es el método que permite iniciar el objeto.

datos miembro (propiedades o atributos)

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

```
objeto.atributo
```

Por ejemplo:

```
Noria.radio;
```

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 valore snecesarios 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.

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*.

creación de objetos de la clase

Una vez definida la clase, se pueden utilizar objetos de la clase. Normalmente consta de dos pasos. Su declaración, y su creación. La declaración consiste en indicar que se va a utilizar un objeto de una clase determinada. Y se hace igual que cuando se declara una variable simple. Por ejemplo:

```
Noria noriaDePalencia;
```

Eso declara el objeto *noriaDePalencia* como objeto de tipo *Noria;* se supone que previamente se ha definido la clase *Noria*.

Para poder utilizar un objeto, hay que crearle de verdad. Eso consiste en utilizar el operador **new.** Por ejemplo:

```
noriaDePalencia = new Noria();
```

Al hacer esta operación el objeto reserva la memoria que necesita y se inicializa el objeto mediante su **constructor**. Más adelante veremos como definir el constructor.

Ilustración 7, *Objeto NoriaDePalencia de la clase Noria en notación UML*

especificadores 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**
Subclase en otro paquete			**X**	**X**
No subclase en otro paquete				**X**

creación de clases

definir atributos de la clase (variables, propiedades o datos de la clases)

Cuando se definen los datos de una determinada clase, se debe indicar el tipo de propiedad que es (String, int, double, int[][],...) y el **especificador de acceso** (public, private,...). El especificador indica en qué partes del código ese dato será visible.

Ejemplo:

```
class Persona {
  public String nombre;//Se puede acceder desde cualquier clase
  private int contraseña;//Sólo se puede acceder desde la
                         //clase Persona
  protected String dirección; //Acceden a esta propiedad
                        //esta clase y sus descendientes
```

Por lo general las propiedades de una clase suelen ser privadas o protegidas, a no ser que se trate de un valor constante, en cuyo caso se declararán como públicos.

Las variables locales de una clase pueden ser inicializadas.

```
class auto{
  public nRuedas=4;
```

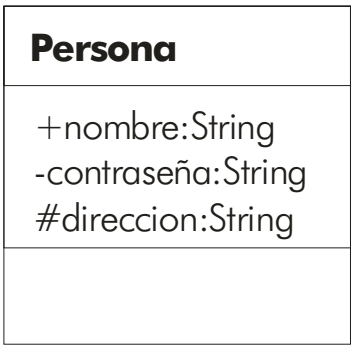

Ilustración 8, La clase persona en UML. El signo + significa *public*, el signo # *protected* y el signo - *private*

definir métodos de clase (operaciones o funciones de clase)

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 o se haga uso de **return**.

Para que un método pueda trabajar, normalmente hay que pasarle unos datos en forma de argumentos o parámetros, cada uno de los cuales se separa por comas. Ejemplos de llamadas:

```
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);
```

Los métodos de la clase se definen dentro de ésta. Hay que indicar un modificador de acceso (**public, private, protected** o ninguno, al igual que ocurre con las variables y con la propia clase) y un tipo de datos, que indica qué tipo de valores devuelve el método.

Esto último se debe a que los métodos son funciones que pueden devolver un determinado valor (un entero, un texto, un valor lógico,...) mediante el comando **return**. Si el método no devuelve ningún valor, entonces se utiliza el tipo **void** que significa que no devuelve valores (en ese caso el método no tendrá instrucción **return**).

El último detalle a tener en cuenta es que los métodos casi siempre necesitan datos para realizar la operación, estos datos van entre paréntesis y se les llama argumentos. Al definir el método hay que indicar que argumentos se necesitan y de qué tipo son.

Ejemplo:

```
public class vehiculo {
    /** Función principal */
    int ruedas;
    private double velocidad=0;
    String nombre;
    /** Aumenta la velocidad*/
     public void acelerar(double cantidad) {
        velocidad += cantidad;
    }
    /** Disminuye la velocidad*/
     public void  frenar(double cantidad) {
        velocidad -= cantidad;
    }
    /** Devuelve la velocidad*/
    public double obtenerVelocidad(){
         return velocidad;
     }

public static void main(String args[]){
      vehiculo miCoche = new vehiculo();
      miCoche.acelerar(12);
      miCoche.frenar(5);
      System.out.println(miCoche.obtenerVelocidad());
    } // Da 7.0
```

En la clase anterior, los métodos **acelerar** y **frenar** son de tipo **void** por eso no tienen sentencia **return**. Sin embargo el método **obtenerVelocidad** es de tipo double por lo que su resultado es devuelto por la sentencia **return** y puede ser escrito en pantalla.

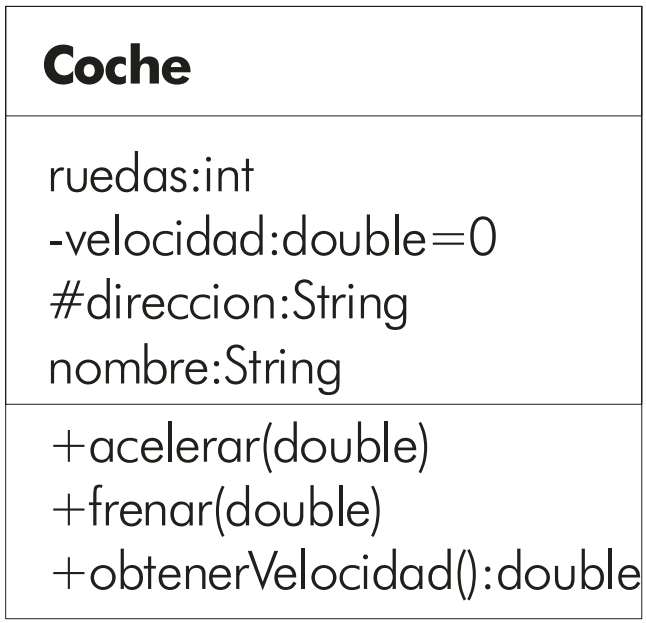

Ilustración 9, Versión UML de la clase Coche

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 pasa 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. Sin embargo la variable x no está afectada por esta asignación.

Sin embargo en este otro caso:

```
class prueba {
  public void metodo1(int[] entero){
        entero[0]=18;
...
  }
...
public static void main(String args[]){
        int x[]={24,24};
       prueba miPrueba = new prueba();
        miPrueba.metodo1(x);
        System.out.println(x[0]); //Escribe 18, no 24
```

Aquí sí que la variable x está afectada por la asignación entero[0]=18. La razón es porque en este caso el método no recibe el valor de esta variable, sino la **referencia**, es decir la dirección física de esta variable. *entero* no es una replica de *x*, es la propia x llamada de otra forma.

Los tipos básicos (**int, double, char, boolean, float, short** y **byte**) se pasan por valor. También se pasan por valor las variables **String.** Los objetos y arrays se pasan por referencia.

devolución de valores

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:

```
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();
    }
}
```

sobrecarga de métodos

Una propiedad de la POO es el polimorfismo. Java posee esa propiedad ya que admite sobrecargar los métodos. Esto significa crear distintas variantes del mismo método. Ejemplo:

```
class Matemáticas{
  public double suma(double x, double y) {
        return x+y;
  }
  public double suma(double x, double y, double z){
        return x+y+z;
  }
  public double suma(double[] array){
        double total =0;
        for(int i=0; i<array.length;i++){
              total+=array[i];
        }
        return total;
  }
```

La clase matemáticas posee tres versiones del método suma. una versión que suma dos números double, otra que suma tres y la última que suma todos los miembros de un array de doubles. Desde el código se puede utilizar cualquiera de las tres versiones según convenga.

la referencia *this*

La palabra **this** es una referencia al propio objeto en el que estamos. Ejemplo:

```
class punto {
  int posX, posY;//posición del punto
  punto(posX, posY){
        this.posX=posX;
        this.posY=posY;
  }
```

En el ejemplo hace falta la referencia **this** para clarificar cuando se usan las propiedades *posX y posY,* y cuando los argumentos con el mismo nombre. Otro ejemplo:

```
class punto {
  int posX, posY;
  ...
        /**Suma las coordenadas de otro punto*/
        public void suma(punto punto2){
              posX = punto2.posX;
              posY = punto2.posY;
        }

  /** Dobla el valor de las coordenadas del punto*/
        public void dobla(){
              suma(this);
        }
```

En el ejemplo anterior, la función dobla, dobla el valor de las coordenadas pasando el propio punto como referencia para la función suma (un punto sumado a sí mismo, daría el doble).

Los posibles usos de **this** son:

- **this.** Referencia al objeto actual. Se usa por ejemplo pasarle como parámetro a un método cuando es llamado desde la propia clase.
- **this.***atributo*. Para acceder a una propiedad del objeto actual.
- **this.***método***(***parámetros***).** Permite llamar a un método del objeto actual con los parámetros indicados.
- **this(***parámetros***).** Permite llamar a un constructor del objeto actual. Esta llamada sólo puede ser empleada en la primera línea de un constructor.

creación de constructores

Un constructor es un método que es llamado automáticamente al crear un objeto de una clase, es decir al usar la instrucción **new**. Sin embargo en ninguno de los ejemplos anteriores se ha definido constructor alguno, por eso no se ha utilizado ningún constructor al crear el objeto.

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:

```
class Ficha {
  private int casilla;
  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. Pero el constructor puede tener parámetros:

```
class Ficha {
  private int casilla; //Valor inicial de la propiedad
  Ficha(int n) { //constructor
        casilla = n;
  }
  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(6);
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 9
```

En este otro ejemplo, al crear el objeto *ficha1*, se le da un valor a la casilla, por lo que la casilla vale al principio **6**.

Hay que tener en cuenta que puede haber más de un constructor para la misma clase. Al igual que ocurría con los métodos, los constructores se pueden sobrecargar.

De este modo en el código anterior de la clase *Ficha* se podrían haber colocado los dos constructores que hemos visto, y sería entonces posible este código:

```
Ficha ficha1= new Ficha(); //La propiedad casilla de la
                           //ficha valdrá 1
Ficha ficha1= new Ficha(6); //La propiedad casilla de la
                            //ficha valdrá 6
```

Cuando se sobrecargan los constructores (se utilizan varias posibilidades de constructor), se pueden hacer llamadas a constructores mediante el objeto **this**

métodos y propiedades genéricos (*static*)

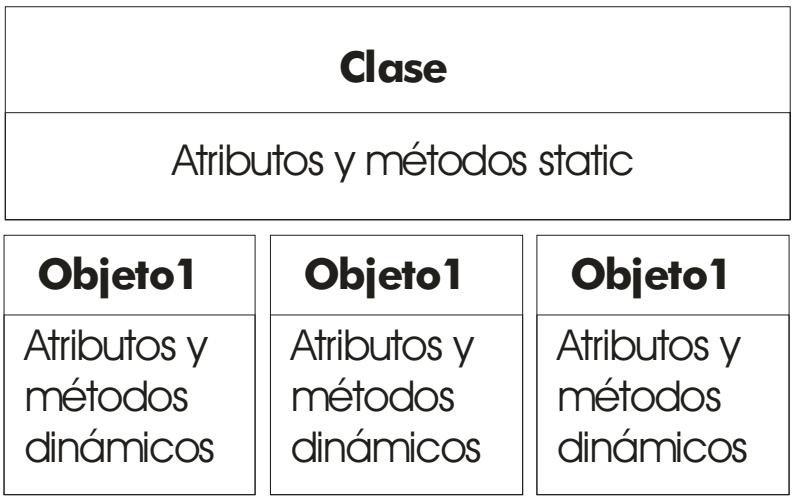

Ilustración 10, *Diagrama de funcionamiento de los métodos y atributos static*

Hemos visto que hay que crear objetos para poder utilizar los métodos y propiedades de una determinada clase. Sin embargo esto no es necesario si la propiedad o el método se definen precedidos de la palabra clave **static**. De esta forma se podrá utilizar el método sin definir objeto alguno, utilizando el nombre de la clase como si fuera un objeto. Así funciona la clase **Math** (véase la clase Math, página 23). Ejemplo:

```
class Calculadora {
   static public int factorial(int n) {
         int fact=1;
         while (n>0) {
               fact *=n--;
         }
         return fact;
   }
}
public class app {
   public static void main(String[] args) {
         System.out.println(Calculadora.factorial(5));
   }
}
```

En este ejemplo no ha hecho falta crear objeto alguno para poder calcular el factorial.

Una clase puede tener métodos y propiedades genéricos (static) y métodos y propiedades dinámicas (normales).

Cada vez que se crea un objeto con **new**, se almacena éste en memoria. Los métodos y propiedades normales, gastan memoria por cada objeto que se cree, sin embargo los métodos estáticos no gastan memoria por cada objeto creado, gastan memoria al definir la clase sólo. Es decir los métodos y atributos static son los mismos para todos los objetos creados, gastan por definir la clase, pero no por crear cada objeto.

Hay que crear métodos y propiedades genéricos cuando ese método o propiedad vale o da el mismo resultado en todos los objetos. Pero hay que utilizar métodos normales (dinámicos) cuando el método da resultados distintos según el objeto. Por ejemplo en un clase que represente aviones, la altura sería un atributo dinámico (distinto en cada objeto), mientras que el número total de aviones, sería un método static (es el mismo para todos los aviones).

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){
   ...instruccionesejecutables....
}
```

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

Los argumentos del método main son un array de caracteres donde cada elemento del array es un parámetro enviado por el usuario desde la línea de comandos. A este argumento se le llama comúnmente *args*. Es decir, si se ejecuta el programa con:

```
java claseConMain uno dos
```

Entonces el método main de esta clase recibe un array con dos elementos, el primero es la cadena "uno" y el segundo la cadena "dos" (es decir args[0]="uno"; args[1]="dos").

destrucción de objetos

En C y C++ todos los programadores saben que los objetos se crean con **new** y para eliminarles de la memoria y así ahorrarla, se deben eliminar con la instrucción **delete.** 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 una función 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 eliminarles de memoria. Este proceso es automático e impredecible y trabajo en un hilo (***thread***) de baja prioridad.

Por lo general ese proceso de recolección de basura, trabaja cuando detecta que un objeto hace demasiado tiempo que no se utiliza en un programa. Esta eliminación depende de la máquina virtual, en casi todas la recolección se realiza periódicamente en un determinado lapso de tiempo. La implantación de máquina virtual conocida como HotSpot[1] suele hacer la recolección mucho más a menudo

Se puede forzar la eliminación de un objeto asignándole el valor **null**, pero teniendo en cuenta que eso no equivale al famoso **delete** del lenguaje C++. Con null no se libera inmediatamente la memoria, sino que pasará un cierto tiempo (impredecible, por otro lado) hasta su total destrucción.

Se puede invocar al recolector de basura desde el código invocando al método estático `System.gc()`. Esto hace que el recolector de basura trabaje en cuanto se lea esa invocación.

Sin embargo puede haber problemas al crear referencias circulares. Como:

```
class uno {
  dos d;
  uno() { //constructor
        d = new dos();
  }
}

class dos {
  uno u;
  dos() {
        u = new uno();
  }

public class app {
```

[1] Para saber más sobre HotSpot acudir a java.sun.com/products/hotspot/index.html.

```
  public static void main(Stgring[] args)  {
        uno prueba = new uno();//referencia circular
        prueba = null; //no se liberará bien la memoria
  }
}
```

Al crear un objeto de clase uno, automáticamente se crea uno de la clase dos, que al crearse creará otro de la clase uno. Eso es un error que provocará que no se libere bien la memoria salvo que se eliminen previamente los objetos referenciados.

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:

```
class uno {
  dos d;
  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).