

Unidad 6: Programación Orientada a Objetos

**Fundamentos de Programación.
1º de ASI**



Esta obra está bajo una licencia de Creative Commons.
Autor: Jorge Sánchez Asenjo (año 2010) <http://www.jorgesanchez.net>
e-mail: info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons
Para ver una copia de esta licencia, visite:
<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>
o envíe una carta a:
Creative Commons, 559 Nathan Abbot



Reconocimiento-NoComercial-CompartirIgual 2.5 España

Usted es libre de:



copiar, distribuir y comunicar públicamente la obra



hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciadador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
Esto es un resumen legible por humanos del texto legal (la licencia completa) disponible en los idiomas siguientes:

Catalán Castellano Euskera Gallego

Para ver una copia completa de la licencia, acudir a la dirección
<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

(6)

programación orientada a objetos

esquema de la unidad

(6.1) introducción a la programación orientada a objetos	6
(6.1.1) qué es la programación orientada a objetos	6
(6.1.2) UML	7
(6.1.3) propiedades de la POO	7
(6.2) introducción al concepto de objeto	8
(6.3) clases	8
(6.4) objetos	11
(6.4.1) creación de objetos	11
(6.4.2) acceso a las propiedades del objeto	12
(6.4.3) métodos	12
(6.4.4) herencia	12
(6.4.5) representación de objetos	12
(6.5) modificadores de acceso	14
(6.6) creación de clases	15
(6.6.1) definir propiedades	15
(6.6.2) métodos	16
(6.6.3) sobrecarga de métodos	21
(6.6.4) métodos recursivos	21
(6.7) la referencia this	24
(6.8) constructores	26
(6.8.1) comentarios Javadoc	28
(6.9) métodos y propiedades genéricos (<i>static</i>)	29
(6.10) el método main	32
(6.11) destrucción de objetos	32
(6.11.1) el método finalize	34

(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 divide 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

las clases de objetos. Una clase podría ser la clase **coche**, que representa un automóvil. Cuando se defina esta clases indicaremos las propiedades (como el color, modelo, marca, velocidad máxima,...) y los métodos (arrancar, parar, repostar,...). Todos los coches (es decir todos los objetos de clase coche) tendrán esas propiedades y esos métodos.

Usando el mismo ejemplo del coche anterior para explicar la diferencia entre clase y objeto; la clase coche representa a todos los coches. Sin embargo cuando me fijo en un coche concreto, entonces me estoy fijando en un objeto de clase coche (es decir, un ejemplar de una clase es un objeto).

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.

(6.1.2) UML

Como ya se comentó en temas anteriores UML es una notación estándar que define diagramas para utilizar en las fases de análisis y diseño de aplicaciones.

La cuestión es que UML siempre ha tenido una buena relación con Java. Por ello ha tenido mucho éxito entre los analistas de aplicaciones en Java. Por supuesto UML es una notación orientada a objetos.

Debido a su clara aceptación como estándar en el mundo de la Ingeniería del Software, en estos apuntes se utilizan sus diagramas para modelar clases, objetos y aspectos de la aplicación. En especial Java y UML siempre han tenido una excelente relación.

Actualmente la última versión de UML es la 2.2 (ISO aceptó como estándar la especificación **UML 1.4.2**).

(6.1.3) propiedades de la POO

- ◆ **Encapsulamiento**. Indica el hecho de que los objetos encapsulan datos y métodos. 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, en los programas orientados a objetos no hay **variables globales**).
- ◆ **Ocultación**. Durante la creación de las clases de objetos, hay métodos y propiedades que se crean de forma **privada**. Es decir, hay métodos y propiedades que sólo son visibles desde la propia clase, pero que no son accesibles desde otras clases. Cuando la clase está ya creada y compilada, esta zona privada permanece oculta al resto de clases. De esta forma se garantiza la independencia entre clases.
- ◆ **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`, 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

nombre sólo puede tener caracteres alfabéticos o números, pero siempre empezar con letra mayúscula. Como consejos al elegir nombre (deberían ser normas obligatorias a seguir):

- **Evitar abreviaturas.** A favor de la legibilidad del código. Es muy importante que el nombre de las clases sea claro y simbolice perfectamente al tipo de objetos que simboliza.
- **Evitar nombres excesivamente largos.** Aunque parece que se contradice con la norma anterior, se trata de que los nombres sean concisos. No es conveniente que sean descripciones de clase, para eso ya están los comentarios **javadoc**.
- **Utilizar nombres ya reconocidos.** Hay abreviaturas reconocidas como por ejemplo **TCP**, por eso el nombre de clase **ManejadorTCP** es mejor que **ManejadorProtocoloTransporte**
- ◆ **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). A los atributos también se les llama **propiedades** y **campos**. Los atributos son valores que poseerá cada objeto de la clase y por lo tanto marcarán el **estado** de los mismos.
- ◆ **Sus métodos.** Las **funciones miembro** de la clase. Son las acciones (u **operaciones**) que puede realizar la clase. Sin duda es lo más complicado de programar y delimitar. Las clases se comunican entre sí invocando a métodos (a esto se le llama **enviar mensajes**).
- ◆ **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).

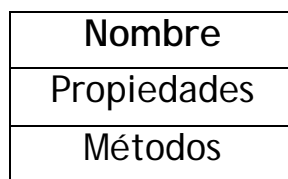


Ilustración 6-1, Clase en notación UML de diagrama de clases

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(){...  
}
```

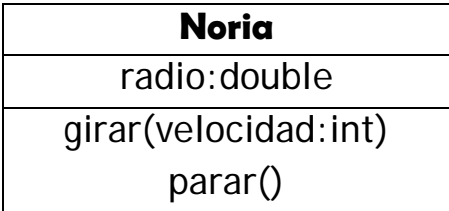


Ilustración 6-2, La clase Noria en notación UML de diagrama de clases

En Java cada clase debe ocupar un archivo, que además debe de tener el mismo nombre. Hasta ahora identificábamos archivo con programa; bien, en realidad no es así. En un archivo se declara y define una clase, que tiene que tener exactamente el mismo nombre que el archivo.

El método **main**, no se declara en cada archivo. La mayoría de clases no tienen métodos **main**, sólo disponen de él las clases que se pueden ejecutar (normalmente en cada proyecto real sólo hay una).

(6.4) objetos

Se les llama también en muchos manuales **instancias de clase**. Este término procede del inglés *instance* que realmente significa *ejemplar*. Así un objeto sería un ejemplar de una clase. Sin embargo el éxito de la mala traducción de la palabra *instance* ha conseguido que de manera más habitual los programadores usen términos como **instancia** o **instanciar** (como se verá más adelante), el problema es que en español no hay un término que realmente represente su significado en .

Los objetos son entidades 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.

Como se comentó antes, el objeto realmente es la información que se guarda en memoria, acceder a esa información es posible realmente gracias a una referencia al objeto.

(6.4.1) creación de objetos

Previamente se tiene que haber definido la clase a la que pertenece el objeto. Para ello primero necesitamos declarar una referencia al objeto lo cual se hace igual que al declarar una variable, es decir se indica la clase de objeto que es y se indica su nombre, es decir la sintaxis es:

```
Clase objeto;
```

Ejemplo:

```
Noria miNoria;
```

Esa instrucción declara que *miNoria*, será una referencia a un objeto de tipo *Noria*. La creación del objeto se hace con el operador **new**:

```
objeto=new Clase();
```

Ejemplo:

```
miNoria=new Noria();
```

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 *stantiate*). Por eso se dice que entre las clases y los objetos hay una relación de tipo *stantiate*.

En UML la forma habitual de representar objetos es la siguiente:

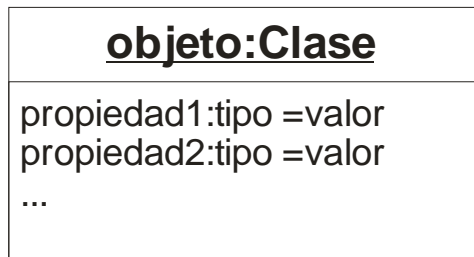


Ilustración 6-3, Sintaxis de un diagrama de objetos UML

Por ejemplo:

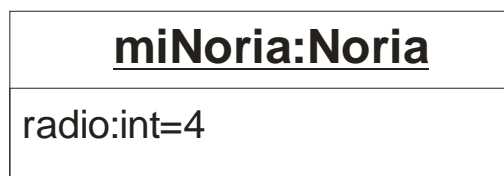


Ilustración 6-4, Diagrama que representa al objeto *miNoria* de clase Noria

A veces se representan de forma abreviada:



E incluso sin indicar el nombre del objeto:



Este último dibujo representa a un objeto cualquiera de clase Noria.

(6.4.6) asignación de referencias

El operador de asignación cuando se usa con referencias a objetos, no tiene el mismo significado que cuando se asignan variables simples o Strings. Ejemplo:

```
Noria n1=new Noria();
Noria n2;
n1.radio=9;
n2=n1;
n2.radio=7;
System.out.println(n1.radio); //sale 7
```

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
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

en detalle en el tema siguiente) con el signo #. Para friendly no se usa ningún signo. Es decir para el código anterior, el diagrama de la clase sería:

En UML se simboliza el modificador **public** con el signo +, el **private** con el signo - y el

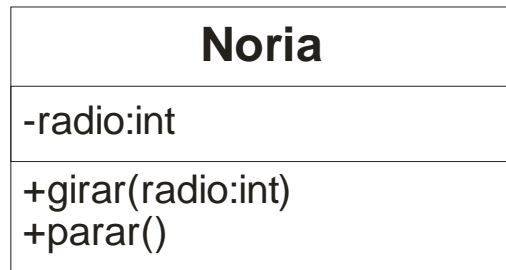


Ilustración 6-5, Diagrama UML de la clase Noria con modificadores de acceso

(6.6) creación de clases

(6.6.1) definir propiedades

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**,...). Ejemplo:

```
public 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.

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

propiedades finales

Los atributos de una clase pueden utilizar el modificador **final**, para que se conviertan en valores no modificables en el objeto. De ser así, se debe iniciar el valor del atributo en la construcción del objeto (más adelante se explica el uso de constructores).

(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 invocarles 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

- (4) **Los parámetros.** Los métodos pueden necesitar datos para realizar su tarea. Dichos parámetros en realidad son una lista de variables u objetos y los tipos o clases de los mismos. La existencia de esas variables u objetos está ligada a la del propio método; es decir, cuando el método finaliza, los parámetros se eliminan.
- (5) **El cuerpo del método.** Es decir el código que permite al método realizar su tarea. Es lo más complicado. Dentro de ese código se pueden declarar variables, objetos y utilizar cualquier conjunto de instrucciones de Java, así como invocar a métodos de otras clases y objetos (si disponemos de visibilidad para ello). El valor resultado del método se realiza mediante la instrucción **return**.

Ejemplo:

```
public class Vehiculo {
    public int ruedas;
    private double velocidad=0;
    String nombre;
    public void acelerar(double cantidad) {
        velocidad += cantidad;
    }
    public void frenar(double cantidad) {
        velocidad -= cantidad;
    }
    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 debe de ser devuelto mediante sentencia **return**.

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 **método1** finaliza, la variable **entero** desaparece, en todo este proceso la variable **x** no ha sido afectada, seguirá valiendo 24.

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 la variable `x` es un array. En este caso el parámetro `entero`, recibe una **referencia** a `x`, no una copia. Por lo que los cambios que se hacen sobre la variable `entero`, en realidad se están haciendo sobre `x` (la forma más propia de indicar esto, es que la acción se hace sobre el array al que `x` y `entero` hacen referencia, que es el mismo). Por eso cuando se ejecuta la asignación `entero[0]=18` el array ha sido modificado y aunque al finalizar la función, `entero` desaparecerá de memoria, el array ha sido modificado y por eso `x[0]` vale 24.

La cuestión es **qué se pasa por referencia y qué por valor**. La regla es simple:

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

El problema es que en Java no podemos pasar variables simples por referencia y esto genera algunos problemas (especialmente para los programadores que proceden de lenguajes como C ó C++ donde sí es posible hacerlo).

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;
    }
}
```

```
public int getEdad() {  
    return edad;  
}  
public void setEdad(int edad) {  
    this.edad=edad;  
}  
}
```

(6.6.3) 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:

```
public 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 números decimales. Desde el código se puede utilizar cualquiera de las tres versiones según convenga.

En definitiva, el método *suma* es polimórfico.

(6.6.4) métodos recursivos

¿qué es la recursividad?

La recursividad es una técnica de escritura de métodos o funciones, pensada para problemas complejos. La idea parte de que un método puede invocarse a sí mismo.

Esta técnica es peligrosa ya que se pueden generar fácilmente llamadas infinitas (la función se llama a sí misma, tras la llamada se vuelve a llamar a sí misma, y así sucesivamente sin freno ni control). Hay que ser muy cauteloso con ella (incluso evitarla si no es necesario su uso); pero permite soluciones muy originales y abre la posibilidad de solucionar problemas muy complejos. De hecho ciertos problemas (como el de las torres de Hanoi, por ejemplo) serían casi imposibles de resolver sin esta técnica.

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 $2 \cdot 1$, es decir 2
- (7) Eso hace que la llamada **factorial(3)** devuelva $3 \cdot 2$, es decir 6
- (8) Por lo que la llamada **factorial(4)** devuelve $4 \cdot 6$, es decir 24 Y ese es ya el resultado final

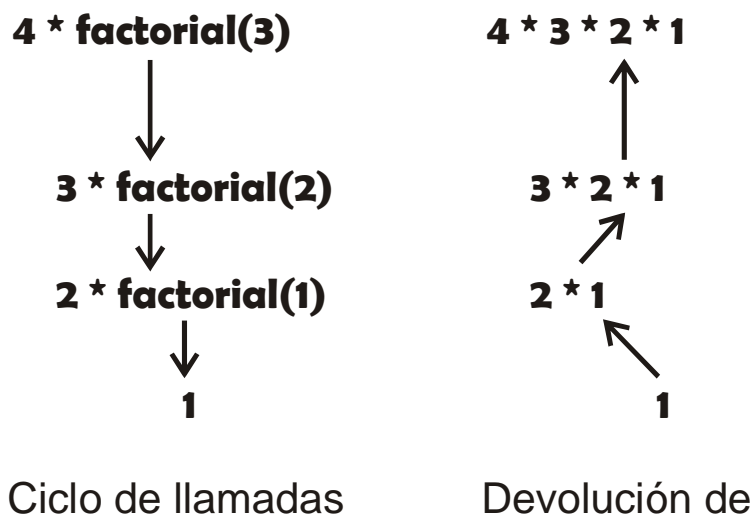


Ilustración 6, Pasos en la ejecución del método recursivo del cálculo del factorial

¿recursividad o iteración?

Hay otra versión del factorial resuelto mediante un bucle **for** (solución iterativa) en lugar de utilizar la recursividad. La cuestión es ¿cuál es mejor?

Ambas implican sentencias repetitivas hasta llegar a una determinada condición. Por lo que ambas pueden generar programas que no finalizan si la condición nunca se cumple. En el caso de la iteración es un contador o un centinela el que permite determinar el final, la recursividad lo que hace es ir simplificando el problema hasta generar una llamada al método que devuelva un único valor.

Para un ordenador es más costosa la recursividad ya que implica realizar muchas llamadas a funciones en cada cual se genera una copia del código de la misma, lo que sobrecarga la memoria del ordenador. Es decir, **es más rápida y menos voluminosa la solución iterativa de un problema recursivo.**

¿Por qué elegir recursividad? De hecho si poseemos la solución iterativa, no deberíamos utilizar la recursividad. **La recursividad se utiliza sólo si:**

- ◆ **No encontramos la solución iterativa a un problema**
- ◆ **El código es mucho más claro en su versión recursiva**

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.

(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;
    }
}
```

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:

```
public class Punto {
    int posX;
    int 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*, hace que cada coordenada del punto valga el doble. Como ya tenemos una función que suma coordenadas, es posible invocar a esta para que calcule el doble sumando las coordenadas del punto consigo mismas, para ello invoca a la función *suma* enviando el propio punto mediante la palabra **this**.

En definitiva, 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.

En realidad cuando desde un método invocamos a un método de la propia clase o a una propiedad de la misma, se usa **this** de forma implícita, es decir que aunque no escribamos **this**, el compilador lo sobreentiende. Por eso en la práctica sólo se indica si es imprescindible.

(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 creamos 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.

Aún más interesante es que los constructores pueden tener parámetros como los métodos y eso permite crear objetos con diferentes valores para sus propiedades (que es lo habitual en la práctica):

```
public class Ficha {
    private int casilla; //Valor inicial de la propiedad
    public 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*. Cuando hay un constructor específico con parámetros, entonces no se puede invocar al constructor por defecto. Es decir en el ejemplo anterior la llamada **new Ficha()** lanzaría un error.

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**. Pero entonces tiene que ser la primera línea del código.

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) comentario 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

Ejemplo:

```
/**
 * Representa muy simbólicamente un vehículo
 * @author Jorge Sánchez
 */
public class Vehiculo {
    /** n° de ruedas del vehículo */
    public int ruedas;
    /** almacena la velocidad a la que marcha el vehículo */
    private double velocidad=0;
    String nombre;
    /** Aumenta la velocidad
     * @param cantidad Cuánto aceleramos el vehículo
     */
    public void acelerar(double cantidad) {
        velocidad += cantidad;
    }
    /**
     * Disminuye la velocidad
     * @param cantidad Cuánto deceleramos el vehículo
     */
    public void frenar(double cantidad) {
        velocidad -= cantidad;
    }
    /**
     * Devuelve la velocidad a la que marcha el vehículo
     * @return la velocidad*/
    public double obtenerVelocidad(){
        return velocidad;
    }
}
```

(6.10) arrays de objetos

En Java es posible crear arrays de objetos. Los propios arrays son objetos, pero dadas sus capacidades para manipular listas de datos, es muy frecuente utilizarles para hacer referencia a conjuntos de objetos. Por ejemplo:

```
Punto a[]=new Punto[4];
```

Con esa instrucción declaramos la existencia de un array que contiene cuatro referencias a Puntos. En ese instante aún no se ha creado ningún punto (los objetos se crean con **new**). Cada elemento del array puede ser una referencia a un nuevo punto o señalar a un punto existente.

Ejemplo:

```
a[0]=new Punto(3,4);  
a[1]=p; //suponiendo que p es una referencia válida aun punto,  
        //a[1] haraá tamboén 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 usarles 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;  
    }  
}
```

Para utilizar el método factorial de la clase Calculadora:

```
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 sólo al definir la clase. Es decir los métodos y atributos **static** son los mismos para todos los objetos creados se almacenan en la zona común de la memoria de la clase.

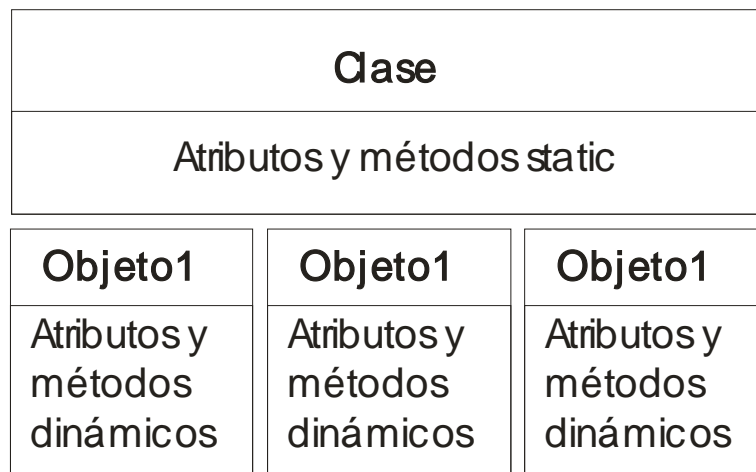


Ilustración 6-7, Funcionamiento de los métodos estáticos

Hay que crear métodos y propiedades genéricos cuando ese método o propiedad vale lo mismo 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 que hemos construido, sería un método **static** (es el mismo para todos los aviones).

Una de las grandes sorpresas para los programadores en lenguaje C que pasan a Java es el hecho de que Java no posee variables ni funciones globales. Las variables globales de C y otros lenguajes crean malas mañas al programar ya que no independizan los objetos del sistema.

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

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 HotSpot¹ 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:

```
public class uno {
    dos d;
    public uno() { //constructor
        d = new dos();
    }
}
public class dos {
    public uno u;
    public dos() {
        u = new uno();
    }
}
public class App {
    public static void main(String[] 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.

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

(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.