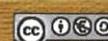


Unidad 6: Programación Orientada a Objetos

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 licenciador (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

las clases de objetos. Una clase podría ser la clase **coche**, que representa un automóvil. Cuando se defina esta clase 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)**

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).

Nombre
Propiedades
Métodos

Ilustración 6-1, Clase 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();

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

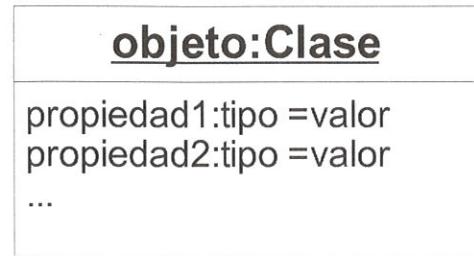


Ilustración 6-3, Sintaxis de un diagrama de objetos UML

Por ejemplo:

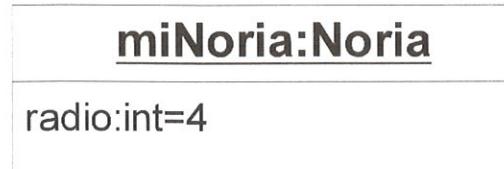


Ilustración 6-4, Diagrama que representa al objeto *miNoria* de clase Noria

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
```

1. Métodos, declaración y utilización

Objetivos:

- a) Describir el funcionamiento de la sentencia return
- b) Interpretar el resultado de una sentencia return en el código fuente de una aplicación Java
- c) Codificar una tarea sencilla convenientemente especificada utilizando la sentencia return

La sentencia `return` se emplea para salir de la secuencia de ejecución de las sentencias de un *método* y, opcionalmente, devolver un valor. Tras la salida del método se vuelve a la secuencia de ejecución del programa al lugar de llamada de dicho método.

Sintaxis:

```
return expresion;
```

1.1. Declaración y uso de métodos

Un *método* es un trozo de código que puede ser llamado o invocado por el programa principal o por otro método para realizar alguna tarea específica. El término *método* en Java es equivalente al de subprograma, rutina, subrutina, procedimiento o función en otros lenguajes de programación. El método es llamado por su nombre o identificador seguido por una secuencia de parámetros o argumentos (datos utilizados por el propio método para sus cálculos) entre paréntesis. Cuando el método finaliza sus operaciones, devuelve habitualmente un valor simple al programa que lo llama, que utiliza dicho valor de la forma que le convenga. El tipo de dato devuelto por la sentencia `return` debe coincidir con el tipo de dato declarado en la cabecera del método.

Sintaxis de declaración de un método:

```
[modificadores] tipoDeDatos identificadorMetodo (parametros formales) {
    declaraciones de variables locales;
    sentencia_1;
    sentencia_2;
    ...
    sentencia_n;
    // dentro de estas sentencias se incluye al menos un return
}
```

La primera línea de código corresponde a la cabecera del método. Los *modificadores* especifican cómo puede llamarse al método, el *tipo de dato* indica el tipo de valor que devuelve la llamada al método y los *parámetros* (entre paréntesis) introducen información para la ejecución del método. Si no existen parámetros explícitos se dejan los paréntesis vacíos. A continuación, las sentencias entre llaves componen el cuerpo del método. Dentro del cuerpo del método se localiza, al menos, una sentencia `return`.

Un ejemplo sencillo

Seguidamente se muestra un ejemplo de declaración y uso de un método que devuelve el cubo de un valor numérico real con una sentencia `return`:

```

* Demostracion de la funcion factorial
* A. Garcia-Beltran - marzo, 2004
*/
public class PruebaFactorial {
    public static void main (String [] args){
        System.out.println("El factorial de 10 es: " + factorial(10));
    }
    public static int factorial (int n) { // declaracion del metodo
        int aux = 1; // declaracion local
        for (int i = 2; i<=n; i++)
            aux *= i; // similar a aux = aux * i
        return aux;
    }
}

```

Ejemplo de ejecución y salida correspondiente por pantalla:

```
$>java PruebaFactorial
El factorial de 10 es: 3628800
```

Una declaración, innumerables llamadas

La declaración del método se realiza una única vez en el código fuente, mientras que el número de llamadas al método puede ser cualquiera. Por ejemplo, el siguiente programa *intenta* realizar una tabla con el valor del factorial de los veinte primeros números naturales:

```

/**
 * Demostracion de la funcion factorial
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaTablaFactorial {
    public static void main (String [] args){
        for (int k=1; k<=20; k++)
            System.out.println(k + "!" = " + factorial(k));
    }
    public static int factorial (int n) {
        int 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 PruebaTablaFactorial
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! = 1932053504
14! = 1278945280
```

En este caso, el error se produce al tratar de calcular el factorial de un valor superior a 20.

Más parámetros

Por otro lado, el número de parámetros o argumentos de los métodos puede ser 0, 1, 2... En el siguiente ejemplo, el método `producto` devuelve el producto de dos valores enteros, `a` y `b`, dados como parámetros o argumentos:

```
/***
 * Demostracion de la funcion producto
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaProducto {
    public static void main (String [] args) {
        System.out.println("Tabla de multiplicar del 5");
        for (int i=0; i<=10; i++)
            System.out.println("5 x " + i + " = " + producto(5,i));
    }
    public static int producto (int a, int b) {
        return a*b;
    }
}
```

Ejemplo de ejecución y salida correspondiente por pantalla:

```
$>java PruebaProducto
Tabla de multiplicar del 5
5 x 0 = 0
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

1.2. Return y void

En algunas ocasiones, no es necesario que el método estático tenga que devolver un valor al finalizar su ejecución. En este caso, el tipo de dato que debe indicar en la cabecera de declaración del método es el tipo `void` y la sentencia `return` no viene seguida de ninguna expresión.

Sintaxis:

```
return;
```

En el siguiente código se incluye un ejemplo de método que no devuelve un valor (de tipo `void`):

```
/***
 * Demostracion del metodo tabla
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaTabla {
```

```
public static boolean esPositivo(int x) {
    if (x<0) return false;
    if (x>0) return true;
    // Error: retorno perdido si x es igual a cero.
}
}
```

Ejemplo de intento de compilación del código anterior:

```
$>javac PruebaPositivo.java
pruebaPositivo.java:14: missing return statement
    }
^
```

Un ejemplo de código correcto sería:

```
/**
 * Demostracion de la funcion esPositivo
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaPositivo2 {
    public static void main (String [] args){
        for (int i=5; i>=-5; i--)
            System.out.println(i + " es positivo: " + esPositivo(i));
    }
    public static boolean esPositivo(int x) {
        if (x<0) return false;
        else return true;
    }
}
```

Ejemplo de ejecución y salida correspondiente por pantalla:

```
$>java PruebaPositivo2
5 es positivo: true
4 es positivo: true
3 es positivo: true
2 es positivo: true
1 es positivo: true
0 es positivo: true
-1 es positivo: false
-2 es positivo: false
-3 es positivo: false
-4 es positivo: false
-5 es positivo: false
```

1.3. Recursión o recurrencia

Java permite la *recursión* o *recurrencia* en la programación de métodos. La recursión consiste en que un método se llame a sí mismo. Un ejemplo muy típico de empleo de la recursión puede verse en la construcción de un método que devuelva el factorial de un entero. Se basa en el hecho de que $n!$ es igual a $n \cdot (n-1)!$ si n es mayor que 1. Por ejemplo:

```
/**
 * Demostracion de la funcion recursiva factorial
 * A. Garcia-Beltran - marzo, 2004
 */
public class PruebaFactorialR {
    public static void main (String [] args){
```

```
int b=12;
int c=56;
System.out.println("a = " + a + "; b = " + b + "; c = " + c);
System.out.println("El mayor de a y b es: " + mayor(a,b)); // El primero
System.out.println("El mayor de a, b y c es: " + mayor(a,b,c)); // El 2º
}
// Definicion de mayor de dos numeros enteros
public static int mayor (int x, int y) {
    return x>y ? x : y;
}
// Definicion de mayor de tres numeros enteros
public static int mayor (int x, int y, int z) {
    return mayor(mayor(x,y),z);
}
```

Ejemplo de salida por pantalla:

```
$>java PruebaSobrecarga
a = 34; b = 12; c = 56
El mayor de a y b es: 34
El mayor de a, b y c es: 56
```

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

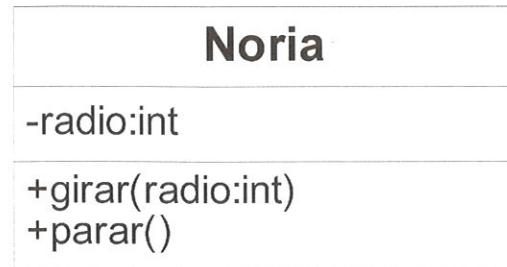


Ilustración 6-5, Diagrama UML de la clase Noria con modificadores de acceso

(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).

- (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 su 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**.

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).

```
1 import java.util.*;
2 public class Factorial10 {
3     public static void main (String args[]) {
4         for (int num1=1;num1<=10;num1++)
5             System.out.printf("El factorial de %2d es: %8d%n",num1,factorial(num1));
6     }
7     static int factorial(int num1){
8         if (num1==0) return 1;
9         return num1*factorial(num1-1);
10    }
11 }
12
13 }
```

```
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.

class's. Notas básicas

NOTA 1 BÁSICA

Un código fuente Java sólo puede tener una clase con el modificador de acceso public. Si el modificador de la clase es público, el código fuente debe tener el mismo nombre que la clase.

Ejemplo 1:

```
1 public class ClasePublica{  
2     public static void main(String args[]){  
3         System.out.println("Estoy en una clase publica");  
4     }  
5 }
```

Si se guarda el código Java en un fichero con un nombre distinto al de la clase y se intenta compilar, se produce error.

Ejemplo 2:

```
1 public class ClasePublicaPrimera{  
2     public static void main(String args[]){  
3         System.out.println("Estoy en la clase publica primera");  
4     }  
5 }  
6 public class ClasePublicaSegunda{  
7     public static void main(String args[]){  
8         System.out.println("Estoy en la clase publica segunda");  
9     }  
10 }
```

Si se guarda el código Java en un fichero con el nombre de una de las clases y se intenta compilar, se produce error.

NOTA 2 BÁSICA

Si un código fuente Java tiene varias clases, lo habitual es que sólo una de ellas cuente con método main. Se verá en el tema de sockets que, a veces, puede resultar cómodo contar con dos métodos main en dos clases de un mismo código fuente. Por el momento, se trabajará con un sólo método main. Si hubiera varios definidos en clases del mismo código fuente, sólo se ejecuta uno de ellos; el resto no se ejecuta.

La pregunta es: ¿qué método main va a ejecutarse?

La respuesta es: el asociado a la clase principal; esta clase principal la define el programador cuando ejecuta el código con: java <clase_principal>.

Ejemplo:

NOTA 4 BÁSICA

Una clase Java puede comunicarse con clases que no se encuentran en su código fuente mediante la creación de objetos adecuados.

Lo que debe hacerse es crear un objeto de la clase a la que se quiere acceder utilizando el nombre de la clase y la palabra reservada new.

Mediante este objeto se tiene acceso a los métodos y variables de instancia no privadas de esa clase. Se verá en los siguientes temas las características de una "variable de instancia" y el modificador de acceso aplicable a variables y métodos "private". Por el momento, sobre las variables de instancia, bastará con saber que se declaran después de la clase y fuera de cualquier método, y que tienen alcance global.

Ejemplo: cada clase se guardará en un código fuente distinto.

```
1 public class Mensaje{  
2     void mostrarMensaje(){  
3         System.out.println("Has calculado el cuadrado de un numero"  
4     }  
5 }  
  
1 public class SinMetodoMain{  
2     int cuadrado;  
3     int calcularCuadrado(int numero){  
4         cuadrado=numero*numero;  
5         return cuadrado;  
6     }  
7 }  
  
1 public class CalculoCuadrado{  
2     public static void main(String args[]){  
3         SinMetodoMain smm=new SinMetodoMain();  
4         int resultado=smm.calcularCuadrado(15);  
5         System.out.println(resultado);  
6         Mensaje m=new Mensaje();  
7         m.mostrarMensaje();  
8         System.out.println("FIN DEL PROGRAMA");  
9     }  
10 }
```

Por consola:

225

Has calculado el cuadrado de un numero

FIN DEL PROGRAMA

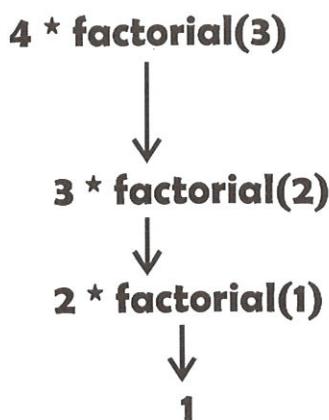
Esta última forma de programar se emplea cuando el programa es sencillo. Lo habitual es que un programa se componga de varias clases en distintos códigos fuente y que cada clase tenga varios métodos y variables de instancia. Para explicar tipos de variables, bucles, estructuras condicionales y conceptos básicos de programación se empleará una clase y sólo el método main. Cuando se hagan programas más complejos se emplearán varias clases y varios métodos en cada clase. El enfoque más orientado a objetos del programa desarrollado en las notas 4, 5 y 6 es, sin duda, el de la nota 4. La clase SinMetodoMain, mediante el método "int calcularCuadrado(int numero)" calcula un entero siempre y cuando reciba a través de su argumento otro.

Esta clase puede ser utilizada por cualquier programador Java en el momento que lo necesite el desarrollo de su programa. Es una práctica habitual en el diseño de una aplicación Java utilizar clases creadas por otros programadores.

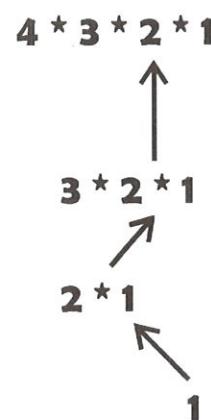
NOTA 7 BÁSICA

Reglas para los nombres de clases, métodos y variables:

- El primer carácter debe ser una letra, el carácter subrayado (_) o el símbolo \$ y no otra cosa. Tras el primer carácter, el nombre puede estar compuesto de letras o de números pero no de espacios en blanco ni tabuladores.
- En las clases, por convenio, el nombre comienza con mayúscula. Si tiene varias palabras la primera letra de cada palabra será también mayúscula.
- En los métodos y variables, por convenio, el nombre comienza con minúscula. Si tienen varias palabras la primera letra de la segunda, tercera... palabras llevará mayúscula.
- Java diferencia entre mayúsculas y minúsculas, es case-sensitive.
- No pueden usarse como nombre palabras reservadas o clave de Java (Java las emplea para los tipos de variables, instrucciones, modificadores de acceso, operadores, etc. propios del lenguaje). Tampoco se permite la ñ.
- No suelen utilizarse acentos (provocan error de ejecución).
- Los nombres deben ser lo mas representativos de las tareas que ejecutan y de los valores que almacenan, para que resulte cómoda la lectura del código a otras personas o incluso a los propios programadores creadores del código una vez transcurrido un cierto tiempo.



Ciclo de llamadas



Devolución de

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

Ejercicio nº 26

- a) Crear un programa que a través de un método recursivo sume ~~misbe~~ la suma de todos los números enteros comprendidos entre dos introducidos por teclado. El método suma recibe como parámetros los dos números límite.

Ejercicio nº 27

- b) Idee, donde el método suma reciba únicamente el menor de los dos. (Declaración el otro como propiedad de la clase).

Ejercicio nº. 29

Crear una clase Java que muestre apilados en forma piramidal (en el plano) un determinado número de botes introducido por teclado.

Se deben tener en cuenta los siguientes requisitos:

- Cada uno de los botes estará representado mediante 5 filas de 5 asteriscos cada una.
- Entre cada dos botes existirá una columna de separación en blanco.
- Un bote se apilará, situando el centro de su base en la columna de separación existente entre dos botes de una fila inferior.
- El número de botes de la base, será siempre el menor necesario para poder apilar todos los botes.
- Las filas, como ya se ha indicado, se representarán en forma piramidal, es decir, el mayor número de botes estará en la base, y el menor número de botes en la fila superior, de tal forma que no se represente una fila que no esté completa, quedando los botes sobrantes depositados a la derecha de la fila base.
- La entrada tendrá una apariencia similar a esta:
Introduce el numero de botes a apilar: 11
- La salida que represente en este caso la clase Java creada será similar a esta:

```
***** ***** ***** *****  
----|-----|-----|----  
***** ***** ***** *****  
----|-----|-----|----  
***** ***** ***** *****  
----|-----|-----|----  
***** ***** ***** *****  
----|-----|-----|----  
***** ***** ***** ***** ..|*****  
***** ***** ***** ***** ..|*****  
***** ***** ***** ***** ..|*****  
***** ***** ***** ***** ..|*****  
***** ***** ***** ***** ..|*****
```

Métodos estáticos o de clase

Se cargan en memoria en tiempo de compilación y no a medida que se ejecutan las líneas de código del programa. Van precedidos del modificador **static**. Para invocar a un método estático **no se necesita crear un objeto de la clase en la que se define**:

- Si se invoca desde la clase en la que se encuentra definido, basta con escribir su nombre.
- Si se le invoca desde una clase distinta, debe anteponerse a su nombre, el de la clase en la que se encuentra seguido del operador punto (.) <NombreClase>.metodoEstatico

Suelen emplearse para realizar operaciones comunes a todos los objetos de la clase. No afectan a los estados de los mismos (a los valores de sus variables de instancia). Por ejemplo, si se necesita un método para contabilizar el número de objetos creados de una clase, tiene sentido que sea estático ya que su función (aumentar el valor de una variable entera) se realizaría independientemente del objeto empleado para invocarle.

No conviene usar muchos métodos estáticos, pues si bien se aumenta la rapidez de ejecución, se pierde flexibilidad, no se hace un uso efectivo de la memoria y no se trabaja según los principios de la Programación Orientada a Objetos.

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

En el ejemplo anterior, la función **dobra**, hace que cada coordenada del punto valga el doble. Como ya tenemos una función que suma coordenadas, es posible invocar a esta parte 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 invitamos 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.

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:

```
/*
 * Representa muy simbólicamente un vehículo
 * @author Jorge Sánchez
 */
public class Vehiculo {
    /** nº 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.

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.

Clase		
Atributos y métodos static		
Objeto1	Objeto1	Objeto1
Atributos y métodos dinámicos	Atributos y métodos dinámicos	Atributos y métodos dinámicos

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.

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.

Ejercicio 2

Enunciado:

Desarrollar una clase llamada Asignatura que:

- Tenga dos atributos private de tipo int (el identificador) y de tipo double (la calificación).
- Tenga un constructor con un parámetro de tipo int.
- Tenga un getter para cada uno de los atributos.
- Tenga un setter para la calificación.

Desarrollar una clase llamada Alumno que:

- Tenga tres atributos private de tipo Asignatura.
- Tenga un constructor con tres parámetros de tipo Asignatura que inicialice los tres atributos.
- Tenga un constructor con tres parámetros de tipo int que inicialice los tres atributos.
- Tenga un getter para cada uno de los atributos.

Desarrollar una clase llamada Profesor que:

- Tenga un método ponerNotas que recibe un parámetro de tipo Alumno y que no devuelve nada. Pondrá una calificación aleatoria a cada una de las asignaturas del alumno.
- Tenga un método calcularMedia que recibe un parámetro de tipo Alumno y devuelve un double.

Ejercicio 3

Desarrollar una clase llamada Práctica5b que en su método main:

- Cree e inicialice tres Asignaturas.
- Cree un Alumno con las tres Asignaturas.
- Cree un Profesor que le ponga calificaciones al Alumno y muestre por pantalla la media del Alumno.

Nota: Para calcular un número aleatorio usaremos Math.random()

```
59     Alumno a=new Alumno(programacion,redes,estadistica);
60     Profesor p=new Profesor();
61     p.ponerNotas(a);
62     System.out.printf("La media del alumno es: %.2f\n",p.calcularMedia(a));
63 }
64 }
65 }
```

Ejercicio32

Crear una clase llamada **Password** que atienda a las siguientes condiciones:

Serán **propiedades** de clase:

- **longitud** y **contraseña**. Por defecto, la longitud será 8.

Contará con dos **Constructores**:

- Un constructor por defecto.
- Un constructor con la longitud que pasemos como parámetro.

Los **métodos** que implementa la clase serán:

- **esFuerte()**: devuelve un booleano si es fuerte o no, para que sea fuerte debe tener al menos: 1 mayúscula, 1 minúscula y 3 números.
- **generarPassword()**: genera la contraseña del objeto con la longitud deseada.
- Aquellos que garanticen la **encapsulación** de las propiedades.

Ahora, crea una clase **Ejercicio32** ejecutable que:

A través de un menú permita seleccionar la longitud de las passwords:

- 1.- Longitud por defecto.
- 2.- Longitud personalizada.

Paralelamente:

- Cree un array de 5 passwords con el tamaño deseado.
- Cree otro array de booleanos donde se almacene si el password del array de passwords es fuerte o no.

Finalmente:

- Muestre las contraseñas generadas acompañadas de la longitud y de la indicación de fortaleza.