4 Programación orientada a objetos. Clases

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender el concepto de recursividad y saber aplicarlo en la resolución de problemas.
- ✓ Agrupar los programas y clases generadas en paquetes para crear una estructura más lógica y útil.
- ✓ Trabajar en profundidad con el concepto de clase.
- ✓ Diseñar e implementar la estructura y miembros de una clase.
- ✓ Estudiar y comprender el concepto de constructor y finalizador.
- ✓ Aplicar el concepto de herencia en la resolución de problemas.
- ✓ Comprender el concepto de interface y su aplicación en Java.

4.1 CREACIÓN DE PAQUETES

El concepto de paquete y CLASSPATH se ha visto en el capítulo 2. Sobre los paquetes hay que tener en cuenta los siguientes conceptos:

- Un paquete es un conjunto de clases relacionadas entre sí.
- Un paquete puede contener a su vez subpaquetes.
- Java mantiene su biblioteca de clases en una estructura jerárquica.
- Cuando nos referimos a una clase de un paquete (salvo que se haya importado el paquete) hay que referirse a la misma especificando el paquete (y subpaquete si es necesario) al que pertenece (por ejemplo: java.io.File).
- Los paquetes permiten reducir los conflictos con los nombres puesto que dos clases que se llaman igual, si pertenecen a paquetes distintos, no deberían de dar problemas.
- Los paquetes permiten proteger ciertas clases no públicas al acceso desde fuera del mismo.

Para la creación de un paquete muy sencillo vamos a seguir los siguientes pasos:

La idea es crear un paquete con dos clases y llamar a dichas clases desde un programa aparte.

1 Lo primero que hay que hacer es crear en el directorio donde estamos compilando los programas un subdirectorio con nombre, por ejemplo, Utilidades. En este subdirectorio vamos a tener varios paquetes (serán subpaquetes del paquete utilidades). El primero de ellos se va a llamar educación y vamos a tener dentro de él dos clases ya compiladas llamadas saludar y despedirse (saludar.class y despedirse.class).

Figura 4.1. Estructura de directorios de Geany

Cada subpaquete estará situado en un subdirectorio aparte del subdirectorio **Utilidades**. Como se puede observar en la imagen anterior, se está utilizando el compilador Geany. En este directorio (Geany) se guardan los programas y las clases, y es aquí donde crearemos estos subdirectorios.

2 Las clases a crear son las siguientes:

```
package Utilidades.educacion;
import java.io.*;
```

```
public class saludar{
    public void saludo(){
          System.out.println("Hola");
    }
}
//** Fin código****

//** Inicio código****
package Utilidades.educacion;
import java.io.*;
public class despedirse{
    public void despedida(){
          System.out.println("Adios");
    }
}
```

package Utilidades.educacion;

Nótese que, con la sentencia anterior, ambas clases indican que pertenecen al paquete educación.

Una vez que he hecho eso, el siguiente paso será importar el paquete con la sentencia *import*.

```
import Utilidades.educacion.*;
public class test {
    public static void main(String[] args) {
                        saludar s=new saludar();
                        despedirse d=new despedirse();
                        s.saludo();
                        d.despedida();
    }
}
```

Se crearán dos objetos, uno de cada clase (*saludar* y *despedirse*) y se hace una llamada a un método de cada clase para verificar que el paquete funciona correctamente. Si se han realizado estos pasos correctamente la compilación no debería dar ningún error.

4.2 CONCEPTO DE CLASE

En la programación orientada a objetos las clases permiten a los programadores abstraer el problema a resolver ocultando los datos y la manera en la que estos se manejan para llegar a la solución (se oculta la implementación). En un programa orientado a objetos es impensable que desde el mismo programa se acceda directamente a las variables internas de una clase si no es a través de métodos *getters* y *setters* (por ejemplo *getEdad()* o *setEdad()*).

Importante
La abstracción es importante en el análisis y diseño de aplicaciones orientadas a objetos. La finalidad del A&D es crear un conjunto de clases que resuelvan el problema que se está abordando.

Por lo tanto, en la definición de nuestras clases deberemos de cuidar lo siguiente:

- No se deberá tener acceso **directo** a la estructura interna de las clases. El acceso a los atributos será a través de *getters* y *setters*.
- En el supuesto que haya que modificar el código sin modificar el interfaz con otras clases o programas, esto debería poder hacerse sin tener ninguna repercusión con otras clases o programas. Se busca que las clases tengan un alto grado de cohesión (independencia).

En Java hay varios niveles de acceso a los miembros de una clase:

- **public** (acceso publico).
- **protected** (acceso protegido).
- **private** (acceso privado).
- **no especificado** (acceso en su paquete).

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

Recuerda
Una subclase es una clase que hereda ciertas características de la clase padre aunque puede añadir algunas propias. Las subclases se estudiarán en profundidad más adelante.

Dependiendo de la finalidad de la clase, utilizaremos un tipo de acceso u otro.

- **Acceso público (public)**. Un miembro público puede ser accedido desde cualquier otra clase o subclase que necesite utilizarlo. Una interfaz de una clase estará compuesta por todos los miembros públicos de la misma.
- **Acceso privado (private)**. Un miembro privado puede ser accedido solamente desde los métodos internos de su propia clase. Otro acceso será denegado.
- **Acceso protegido (protected)**. El acceso a estos miembros es igual que el acceso privado. No obstante, para las subclases o clases del mismo paquete (*package*) a la que pertenece la clase, se considerarán estos miembros como públicos.

- **Acceso no especificado (paquete)**. Los miembros no etiquetados podrán ser accedidos por cualquier clase perteneciente al mismo paquete.

Consejo
Para un mayor control de acceso se recomienda etiquetar los miembros de una clase como *public*, *private* y *protected*.

A modo de resumen se especificarán los niveles de acceso vistos anteriormente en la siguiente tabla:

Tabla 4.1. Modificadores de acceso en Java

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

(*) Este caso no se suele dar con frecuencia. Se podría acceder al atributo o método desde objetos de la subclase pero no así por objetos de la superclase.

4.2.1 CONTROL DE ACCESO A UNA CLASE

Cuando creamos una clase en Java es posible definir la relación que esa clase tiene con otras clases o la relación que tendrá esa clase con las clases de su mismo paquete.

Recuerda
Una clase definida como pública puede ser utilizada por las clases de su paquete y otros paquetes mientras que una clase no definida como pública solamente podrá ser utilizada por las clases de su propio paquete.

Clase pública	Clase NO definida como pública

`public class miClase {`
`.....`
`}`
	`class miClase {`
	`.....`
	`}`

Puede ser utilizada por cualquier clase.	Puede ser utilizada SOLO por clases de su propio paquete.

4.2.2 REFERENCIA AL OBJETO THIS

Java, al igual que C++, proporciona una referencia al objeto con el que se está trabajando. Esta referencia se denomina **this**, que no es ni más ni menos que el objeto que está ejecutando el método. En los ejemplos que hemos estado utilizando en muchas ocasiones se obviaba esta referencia puesto que se sobreentiende que el objeto está invocando al método. En algunas ocasiones nos va a servir para resolver ambigüedades o para devolver referencias al propio objeto. En el siguiente ejemplo se ve claramente el uso del **this**.

Observa
En el siguiente código vas a poder apreciar que la referencia this en ocasiones se puede omitir. Observa también cómo se devuelve una referencia al propio objeto en los métodos *incrementarAncho()* e *incrementarAlto()*.

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

```
    public rectangulo incrementarAncho(){
          ancho++; //se puede omitir el this
          return this;
    }
    public rectangulo incrementarAlto(){
          this.alto++;
          return this;
    }
}
```

4.2.3 LA CLASE OBJECT

Importante
La clase object es la raíz jerárquica de Java.

Cualquier clase implementada en Java siempre va a ser una subclase de la clase **object**. Eso quiere decir que va a heredar todos los métodos de **object**. De todos los métodos de la clase **object** vamos a ver con más profundidad los siguientes:

Tabla 4.2. Métodos de la clase Object

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

Método clone()

El método **clone** nos permite copiar un objeto en otro. Utilizar este método equivaldría a utilizar un constructor de copia. La clase base **object** tiene el método *clone()* que es el mecanismo que utiliza Java para clonar objetos. Es posible

y en muchos casos necesario implementar un método **clone**, el cual sobrescribirá al método **clone** de su superclase y podrá actuar de una forma más específica que el método genérico **clone**().

El método genérico *clone()* hace una copia superficial del objeto.

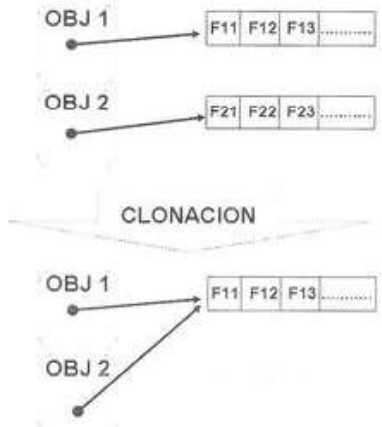

Figura 4.2. Copia superficial

Como se puede ver en la figura anterior, la copia superficial únicamente hace una copia del contenido de un objeto en otro, lo que en algunas ocasiones provoca que la modificación del contenido de un objeto implique el cambio en el clonado y viceversa.

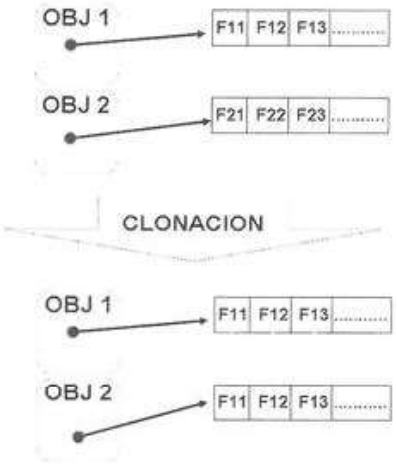

Figura 4.3. Copia en profundidad

Por el contrario, las copias en profundidad pueden hacer una copia selectiva del contenido de un objeto en otro. En este caso ambos objetos vivirán "vidas independientes".

Un ejemplo de realizar una clonación de un objeto en Java sería el siguiente:

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

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

Como se puede observar, la clase objeto de la clonación deberá de implementar la interfaz **cloneable**. Si no se implementa esta interfaz, el programa lanzará una excepción del tipo *CloneNotSupportedException*. También se ha implementado el método *clone()*, el cual hace una llamada al método *clone()* de su clase base.

Importante
Muchas veces es más cómodo para el programador utilizar el constructor de copia que el método *clone()*.

Método equals()

El método **equals** permite realizar una comparación entre un objeto y otro. Lo que hace es comprobar que ambas referencias sean iguales, con lo cual no obtenemos más ventaja que con el operador = =. No hace una comparación en profundidad sino que se limita a comprobar las referencias de los objetos. Si se quiere realizar una comprobación en profundidad habrá que reescribir este método.

Un ejemplo de utilización de este método es el siguiente:

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

El resultado en pantalla de ejecutar el código anterior será: "Iguales r1 y r3(equals)". Para que la primera comprobación sea verdadera habrá que reescribir el método **equals**.

Método toString()

El método **toString**() permite obtener el nombre de la clase desde el cual fue invocado. Además del nombre de la clase, devuelve el carácter '@' y la representación hexadecimal del código **hash** del objeto. Un ejemplo de la llamada a este método es el siguiente:

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

Este código devolverá por pantalla lo siguiente:

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

Como podemos observar en el código, al hacer r3 = r1 lo que hacemos es que ambas referencias apunten al mismo objeto con lo cual al invocar al método *toString()* el resultado será el mismo.

Método finalize()

Cuando el recolector de basura de Java (*garbage collector*) tiene constancia de que no existen más referencias a un objeto concreto, invoca a este método y se encarga de liberar su memoria ocupada. Si el programador necesita realizar una acción una vez destruido un objeto deberá reescribir este método.

4.3 ESTRUCTURA Y MIEMBROS DE UNA CLASE

En esta sección se va a trabajar en profundidad con los miembros *static*, así como con los métodos de instancia y de clase. Es importante que el alumno comprenda y sepa diferenciar estos miembros y ambos métodos.

4.3.1 MIEMBROS ESTÁTICOS (STATIC) DE UNA CLASE / MIEMBROS DE CLASE

En Java no existen variables globales, por lo tanto, si queremos utilizar una variable única y que puedan utilizar todos los objetos de una clase deberemos de declararla como estática (*static*).

Recuerda
A diferencia de los miembros normales o miembros de instancia, los miembros de clase tienen la cláusula static y todos los objetos de la misma clase compartirán dichos miembros.

Veamos como funcionan los atributos estáticos de una clase con el siguiente ejemplo:

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

Tenemos una clase la cual tiene un miembro estático. Esta variable *numcohetes* almacenará el número de objetos cohete que se van creando.

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

Cuando desde otra clase, por ejemplo la anterior, se crean varios objetos de la clase cohete (3 objetos) y se llama al método *getcohetes()* ¿qué valores devolverá dicho método?

```
System.out.println(c1.getcohetes());
System.out.println(c3.getcohetes());
```

La solución es 3 en ambas llamadas. La variable *numcohetes* se inicializa a 0 solo una vez (cuando se crea el objeto c1). Cuando se crean los objetos c2 y c3 no se vuelve a inicializar pues ya existe y es estática, solo se incrementa.

Al haber definido *numcohetes* como *private*, no es posible desde nuestra clase *testestaticos* acceder a *c1.numcohetes*.

Recuerda
Los miembros o atributos de instancia son aquellos que no son *static*.

4.3.2 MÉTODOS DE INSTANCIA Y DE CLASE

Los métodos de una clase son una abstracción del comportamiento de la misma. Los algoritmos formarán parte de los métodos y contendrán la lógica de la aplicación que queramos desarrollar.

Podemos dividir los métodos en dos bloques:

- Métodos de **instancia**. Son aquellos utilizados por la instancia.
- Métodos de **clase**. Son aquellos comunes para una clase. Un método por clase.

Recuerda
Miembros o atributos de instancia y de clase son análogos a métodos de instancia y de clase.

4.3.3 MÉTODOS DE INSTANCIA

Los métodos de instancia son, por así decirlo, los llamados métodos comunes. Cada instancia u objeto tendrá sus propios métodos independientes del mismo método de otro objeto de la misma clase.

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

En el anterior ejemplo podemos ver un método de instancia.

Recuerda
Los métodos de instancia pueden acceder a los miembros de instancia y también a los miembros de clase.

La siguiente clase, atendiendo a la regla anterior compilará sin problemas:

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

No obstante en vez de utilizar la línea:

```
var = 3;
```

Quizás hubiese sido más correcto utilizar la siguiente:

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

La llamada a un método de instancia sería la siguiente:

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

4.3.4 MÉTODOS ESTÁTICOS O DE CLASE

Recuerda las siguientes reglas
1. Los métodos *static* no tienen referencia *this*.
2. Un método *static* no puede acceder a miembros que no sean *static*.
3. Un método *no static* puede acceder a miembros *static* y *no static*.

Veamos alguna de estas reglas en un pequeño programa:

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

Veremos las razones por las cuales las líneas resaltadas en negrita dan error de compilación.

```
this.datostatico++; // Esto da error al compilar
```

La sentencia anterior produce un error debido a la regla 1 (los métodos *static* no tienen referencia *this*).

```
dato++;  // Esto da error al compilar
```

La sentencia anterior produce un error debido a la regla 2 (un método *static* no puede acceder a miembros que no sean *static*) dado que dato no es *static*.

```
metodo();  // Esto da error al compilar
```

La sentencia anterior produce un error debido a la regla 2 (un método *static* no puede acceder a miembros que no sean *static*) dado que el método *metodo()* no es *static*. Sin embargo según la regla 3, el método *metodo()* puede acceder al dato datostatico dado que éste método no es estático.

Recuerda
Los métodos de clase o *static* NUNCA pueden acceder a los miembros de instancia.

Tabla resumen:

Tabla 4.3. Tabla resumen

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

Un ejemplo de los métodos de clase son las funciones de la librería java.lang.Math las cuales pueden ser llamadas anteponiendo el nombre de la clase Math. Un ejemplo de llamada a una de estas funciones son por ejemplo:

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

Como se puede observar se antepone el nombre de la clase (Math) al del método cos.

Algunos métodos estáticos (los más utilizados) de la clase Math son los siguientes:

Tabla 4.4. Métodos de la clase Math

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

Además de los métodos vistos, la clase Math tiene un sinfín de funciones trigonométricas además de muchas otras funciones.

4.4 TRABAJANDO CON MÉTODOS

4.4.1 PASO DE PARÁMETROS POR VALOR Y POR REFERENCIA

Generalmente es común pasar parámetros a los métodos salvo que sean métodos para inicializar o finalizar el objeto. Existen ocasiones en las que necesitamos que estas variables que pasamos como parámetros cambien su valor una vez ejecutado el método si este las ha modificado. Si es una variable se puede solucionar con la sentencia *return*, pero imagínate que queremos pasar 5 variables a un método y conservar los valores si éste los modifica. En ese caso con *return* solamente podríamos obtener una variable modificada. Por lo tanto, deberemos utilizar paso de parámetros por referencia.

Resumiendo:

- Paso de parámetros **por valor**. Los parámetros se copian en las variables del método. Las variables pasadas como parámetro no se modifican.
- Paso de parámetros **por referencia**. Las variables pasadas como parámetro se modifican puesto que el método trabaja con las direcciones de memoria de los parámetros.

Un ejemplo de esto explicado es el siguiente:

```
public class testparam {
    public static void cambiar(int x){
          x++;
    }
    public static void cambiar2(int[] par){
          par[0]++;
    }
    public static void main(String[] args) {
          int x = 3;
          int []arrx={3};
          cambiar(x);
          System.out.println(x);
          cambiar2(arrx);
          System.out.println(arrx[0]);
    }
}
```

Este programa dará como salida los valores 3 y 4. En la función cambiar2 se pasa un array en vez de una variable. La diferencia entre un array de enteros y una variable entera es que un array de enteros es una dirección de memoria donde de manera consecutiva se almacenarán una serie de valores enteros. Los **arrays** o **vectores** se estudiarán más adelante en profundidad en el **capítulo 6**.

Gráficamente el comportamiento del programa es el siguiente:

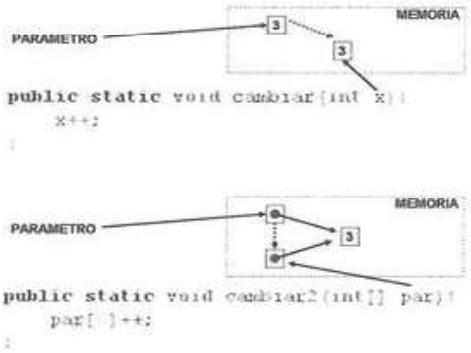

Figura 4.4. Parámetros por valor y por referencia

Como puedes ver, en la función *cambiar*, lo que se hace es que se copia el contenido del parámetro a la variable x del método. Sin embargo, en la función cambiar2, aunque se hace lo mismo, lo que cambia es que el valor que contiene el parámetro es a su vez una dirección de memoria. Con lo cual, cada cambio en la variable par del método repercutirá en un cambio del parámetro pasado por referencia.

4.4.2 LOS MÉTODOS RECURSIVOS

A FONDO

LOS MÉTODOS RECURSIVOS

Un método se llama **recursivo** cuando **se llama a sí mismo**.

¿Cuándo utilizar la recursividad?

- Cuando la resolución de un problema es más sencilla.
- Cuando no es infinita, es decir, hay un caso resoluble más básico o más sencillo.

Generalmente, cuando se va a resolver un problema recursivo vemos que en cada llamada sucesiva al método recursivo nos vamos acercando cada vez más a la solución.

¿Es eficiente la recursividad?

NO. La recursividad **no** es eficiente pero es sencilla de programar y de entender. Hay que tener siempre en cuenta que para un método recursivo siempre hay uno equivalente iterativo.

Ejemplo de recursividad

Vamos a ver la recursividad con un ejemplo sencillo. El método que vamos a escoger es la potencia de

un número. Nuestro método será el siguiente:

`potencia(x,y)` → x^y

Para resolver un caso recursivo generalmente debemos encontrar:

1. Una fórmula o proceso que reduzca la complejidad y nos vaya acercando a la solución.
2. Un caso base que hace que nuestra recursividad no sea infinita.

En el caso de la potencia:

- Sabemos que $x^y = x * x^{y-1}$ (ésta es la fórmula que reduce la complejidad).
- Y que $x^0 = 1$ (caso base).

Este problema, como se puede observar, es un claro caso de método recursivo.

La programación del método sería la siguiente:

```
public static int potencia(int x, int y){
   if (y == 1){ //caso base
   return x;
   }else{ //reducción de la complejidad
   return x * potencia(x,y-1);
   }
   }
```

Veamos como funciona una llamada al método cuando queremos realizar la operación 2^3.

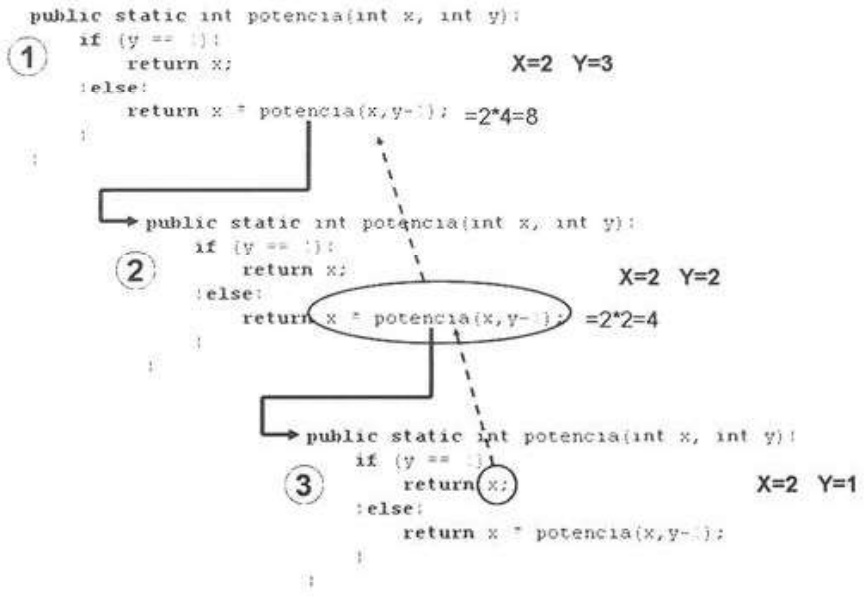

Figura 4.5. Estructura de la llamada a un método recursivo

Como se puede observar, cuando se realiza una llamada al método potencia(2,3), éste va a estar llamandose recursivamente hasta que el segundo parámetro(y) valga 1. Luego, en nuestra llamada al método se producirán dos subllamadas (potencia(2,2) y potencia(2,1)). En la última llamada (cuando y = 1) se produce el fin de las llamadas recursivas y se procede a recuperar los valores obtenidos en las subllamadas (retorno de los valores 2 y 4). Al final, el método devuelve 8 (2 * el valor retornado que es 4).

Para entender más sobre la recursividad se recomienda trabajar en profundidad los ejercicios resueltos.

4.5 LOS CONSTRUCTORES

Java, al igual que hace con las variables, cuando va a crear un objeto, lo que hace es reservar espacio en memoria para dicho objeto. En esta fase de construcción del objeto, Java crea un constructor público por defecto del objeto. No obstante, si el programador lo cree oportuno se puede un constructor diferente que satisfaga las necesidades de la clase.

Recuerda
El constructor se llama de forma automática siempre que se crea un objeto de una clase.

Por lo tanto tenemos dos tipos de constructores:

- **Constructor por defecto**. Cuando no se especifica en el código. Se ejecuta siempre de manera automática e inicializa el objeto con los valores especificados o predeterminados del sistema.
- **Constructor definido**. Puede ser más de uno. Tiene el **mismo nombre** de la clase. Nunca devuelve un valor y no puede ser declarado como *static*, *final*, *native*, *abstract* o *synchronized*. Por regla general se declaran los constructores como públicos (**public**) para que puedan ser utilizados por cualquier otra clase.

Recuerda
Cuando existe más de un constructor para una clase se dice que este está sobrecargado.

¿Qué hace Java cuando tiene que cargar una clase?

1. Antes de crear el primer objeto, Java localiza el fichero de la clase en disco (recuerda el fichero **.class**) y lo carga en memoria.
2. Se ejecutarán los inicializadores *static* de la clase (se explican en la sección a fondo al final de este apartado).
3. Se crea el objeto.

¿Qué hace Java cuando se crea un objeto?

1. Crea memoria para el objeto mediante el operador *new*.
2. Inicializa los atributos del objeto (solamente los que no fueron inicializados).
3. Se ejecutan los inicializadores de objeto.
4. Llama al constructor adecuado.

Tabla 4.5. Inicialización predeterminada de variables

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

Figura 4.6. Pasos enla creación de los objetos

4.5.1 SOBRECARGA DEL CONSTRUCTOR

¿Cuándo necesitamos sobrecargar o definir múltiples constructores para una clase?

Se definen múltiples constructores para una clase cuando el objeto pueda ser inicializado de múltiples formas. Al sobrecargar un constructor variaremos el tipo y número de parámetros que recibe.

La siguiente clase muestra un ejemplo de sobrecarga de constructores:

```
public class rectangulo
{
    private int ancho;
    private int alto;
    rectangulo(int an, int al){
          this.ancho = an;
          this.alto = al;
    }
    rectangulo(){
          ancho=alto=0;
    }
    rectangulo(int dato){
          ancho=alto=dato;
    }
.........
}
```


Recuerda
Cuando existe más de un constructor para una clase se dice que este está sobrecargado. Cuando creamos un objeto con *new*, Java elige el constructor más adecuado dependiendo de los parámetros utilizados.

Como según el ejemplo de la clase rectángulo tenemos tres constructores, la creación de cada uno de los objetos siguientes se realizará con un constructor diferente:

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

4.5.2 ASIGNACION DE OBJETOS

Importante
Cuando trabajamos con objetos estamos trabajando con referencias. Una referencia es una localización de la memoria donde se encuentra el objeto.

Como ya se ha dicho, hay que tener en cuenta la utilización de estas referencias cuando se trabaja con objetos. De forma sencilla se va a comprender qué es este enigma de las referencias:

Imaginemos que tenemos la siguiente clase rectángulo:

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

Como esta clase es pública, desde el método *main* de otra clase ejecuto el siguiente código:

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

La pregunta es la siguiente: ¿Qué mostrará el programa por pantalla? Tenemos dos opciones:

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

La respuesta es la **opción 2**. Esto es así porque cuando se hace **r2 = r1** se copia la referencia al objeto y no el contenido de un objeto en otro. En la siguiente figura se puede ver esto de forma gráfica:

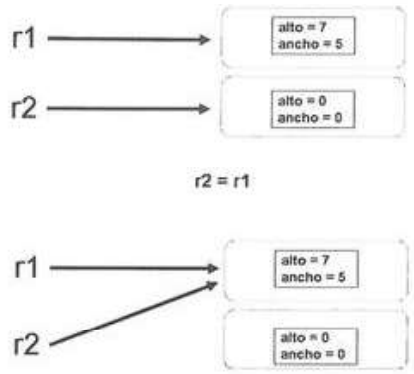

Figura 4.7. Asignación de referencias

Entonces, ¿cómo se copia el contenido de un objeto a otro?

Una solución sencilla es utilizar un constructor de copia. En la siguiente sección aprenderás a utilizarlo.

4.5.3 CONSTRUCTOR COPIA

Con un constructor de copia se inicializa un objeto asignándole los valores de otro objeto diferente de la misma clase. Este constructor de copia tendrá solo un parámetro: un objeto de la misma clase.

Un constructor de copia para nuestra clase anterior rectángulo sería el siguiente:

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

En el caso de que tenga este nuevo constructor, puedo hacer uso del mismo cuando cree un objeto r2 de la misma clase:

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

La utilización del constructor copia "copiará" los miembros del objeto r1 al objeto r2. El programa anterior ahora mostrará por pantalla lo siguiente:

Alto: 7

Ancho: 5

A FONDO

INICIALIZADORES STATIC

Los inicializadores *static* son un bloque de código que se ejecutará una vez solamente cuando se utilice la clase.

Importante: A diferencia del constructor que se llama cada vez que se crea un objeto de dicha clase, el inicializador solamente se ejecuta la primera vez que se utiliza la clase.

Los inicializadores *static* siguen las siguientes reglas:

- No devuelven ningún valor.
- Son métodos sin nombre.
- Ideal para inicializar objetos o elementos complicados.
- Permiten gestionar excepciones con try...catch.
- Se puede crear más de un inicializador *static* y se ejecutarán según el orden en el que se han definido.
- Se pueden utilizar para invocar métodos nativos o inicializar variables static.
- A partir de Java 1.1 existen los inicializadores de objeto utilizados en las clases anónimas y no tienen el modificador *static*.

Un ejemplo muy sencillo de clase con varios inicializadores es el siguiente:

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

Desde el siguiente programa vamos a crear tres objetos de la clase testInicializador:

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

Este programa mostrará por pantalla lo siguiente:

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

4.6 LOS DESTRUCTORES

En Java no existen los destructores

Aunque parezca raro empezar un apartado de destructores diciendo que no existen los destructores, en realidad es así. Al contrario que en C++ y otros lenguajes OO, en Java no existen los destructores. En un intento de simplificar las cosas y mejorar la gestión de memoria, es el propio sistema el que se encarga de eliminar definitivamente los objetos de la memoria cuando le asignamos el valor *null* a la referencia (referencia = null;), le asignamos a la referencia un objeto diferente o bien termina el bloque donde está definida la referencia.

El sistema de liberación de memoria en Java se llama *garbage collector* (recolector de basura), este recolector trabaja de forma automática. Como se vio en apartados anteriores se le puede sugerir que se active realizando la llamada *System.gc()*, pero esta sola no es la única razón para que se active el recolector de basura (se activará cuando él lo decida, generalmente cuando falta memoria).

4.6.1 LOS FINALIZADORES

Cuando se va a liberar automáticamente la memoria de objetos inservibles, el sistema ejecuta el finalizador de los objetos. El finalizador se caracteriza por no tener valor de retorno ni argumentos, no puede ser *static* y denominarse *finalize()*. Un ejemplo de finalizador es el siguiente:

```
protected void finalize(){System.out.println("Adioossss");}
```

Generalmente, los finalizadores se utilizan para liberar memoria, cerrar ficheros, conexiones, etc. Como no se sabe a ciencia cierta cuándo se van a ejecutarlos finalizadotes es el recolector de basura el que se encarga de ello, el consejo es que las operaciones de liberación de ciertos recursos se realicen de forma explícita (a mí no se me ocurriría cerrar una conexión de una base de datos en un finalizador).

Como se verá más adelante, existe una forma de sugerir a Java que ejecute el recolector de basura y es llamando al método *System.runFinalization()* y luego al recolector de basura *System.gc()*.

Un ejemplo de la llamada al finalizador es el siguiente:

```
public class rectangulo
{
    ..........
    protected void finalize(){System.out.println("Adioossss");}
    ..........
}
class testfinalize {

    public static void main(String[] args) {
          for (int i = 0; i < 20; i++) {
          rectangulo r = new rectangulo(5,5);
          }
          System.runFinalization();
          System.gc();
    }
}
```

Como se puede observar en el código se ha definido el método *finalize()* como *protected* para evitar que pueda ser invocado desde fuera de la clase.

En el código lo que se ha hecho es crear una serie de objetos cuyo *scope* o ámbito está reducido a un bucle *for*. Una vez realizado esto hay que tener en cuenta que *finalize*() no se invoca cuando termina su *scope* (cuando termina el bucle). Este método se ejecutará justo antes de ejecutarse el *garbage collector* por lo que en nuestro código hemos tenido que forzar este hecho con las siguientes líneas:

```
System.runFinalization();
System.gc();
```

Sin las anteriores líneas el programa no mostrará nada por pantalla.

Importante
El *garbage collector*, gc o recolector de basura, se ejecuta en segundo plano en un subproceso paralelo a la propia aplicación. La llamada al recolector de basura se hace ejecutando el método *gc()* de la clase *System*.

Resumiendo:

- El método *finalize()* no es el destructor de C++. En Java no existe el destructor, existe la recolección de basura.
- El método *finalize()* tiene que estar asociado a recuperar la memoria que ha sido utilizada y ya no sirve, no hay que programar otro tipo de cosas aquí.
- Es el sistema y no el programador el que decide cuándo se ejecuta el recolector de basura.
- Los objetos pueden o no ser eliminados por el recolector de basura. En algunos casos no se eliminan si no existe una necesidad de memoria.
- Generalmente, se utiliza *finalize()* cuando se hacen llamadas a métodos nativos (por ejemplo en C o C++) para reservar memoria y luego ésta necesita ser liberada.
- Visto lo anterior, es normal pensar que el método *finalize()* no se va a utilizar mucho en Java y, salvo necesidad específica, esto es siempre así.

4.7 ENCAPSULACIÓN Y VISIBILIDAD. INTERFACES

Como ya sabemos, un objeto interactúa con el mundo exterior a través de su interfaz. En el caso de un ordenador, por ejemplo, las interfaces con el mundo exterior serán la pantalla, el ratón, el teclado, etc. Cuando nosotros tecleamos en un ordenador, las teclas o la pantalla sirven para comunicarnos con la parte interna del equipo. Imaginemos que actualizamos la memoria, el procesador y la placa base del equipo conservando la parte software del mismo. La interfaz será exactamente la misma y las personas interactuarán exactamente igual con ella. Lo único que notarán es una mejora del rendimiento. Con los objetos pasa exactamente lo mismo, la interacción con el mundo exterior es a través de sus métodos. Los métodos componen la interfaz del objeto con el mundo exterior.

Una interfaz es un grupo de métodos con sus cuerpos vacíos. Por ejemplo, la interfaz figura (intfigura) podría ser el siguiente:

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

La interfaz define el método área, para su posterior desarrollo en las clases que implementen esta interfaz. Una de las clases que podría implementar esta interfaz es la clase rectángulo:

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

```
    public int area(){ return ancho*alto; }
}
```

Como se puede observar en la declaración, la clase rectángulo implementa la interfaz *intfigura*.

Recuerda
Para compilar correctamente una clase que implementa una interfaz, ésta debe contener los métodos declarados en dicha interfaz.

4.8 HERENCIA

La herencia es la base de la reutilización del código. Cuando una clase deriva de una clase padre, ésta hereda todos los miembros y métodos de su antecesor. También es posible redefinir (*override*) los miembros para adaptarlos a la nueva clase o bien ampliarlos. En general, todas las subclases no solo adoptan las variables y comportamiento de las superclases sino que los amplían.

Recuerda
En Java al contrario que en C++ no se permite la herencia múltiple. Es decir, una clase no puede heredar de varias clases.

En la siguiente figura se muestra un ejemplo de herencia:

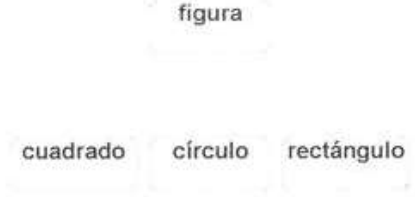

Figura 4.8. Estructura de clases descendientes de figura

Como se puede ver en la figura, en este árbol de herencia tendremos la clase *figura* de las que heredan las clases *cuadrado*, *círculo* y *rectángulo*. Como es obvio, *cuadrado*, *círculo* y *rectángulo* tienen una característica en común y

es que todas son figuras. Otra característica que presenta este árbol es que, en este caso, las figuras por sí mismas no existen, es decir, existirán pero siempre deberá de ser a través de una clase de nivel inferior (cuadrado, círculo o rectángulo).

Para indicar que una clase hereda de otra se etiqueta con la cláusula **extends** detrás del nombre de la clase. Por ejemplo, para indicar que la clase *rectángulo* hereda de la clase *figura* escribiremos lo siguiente:

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

Igual podremos hacer para las demás clases:

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


Recuerda
Todas las clases tienen una superclase o clase padre. Cuando escribas una clase, si ésta no hereda de ninguna clase concreta en realidad hereda de la clase Object (java.lang.Object).

Imaginemos que queremos realizar una estructura de clases como la que se muestra a continuación. El código de la clase *figura* y *cuadrado* se muestra junto a la siguiente imagen:

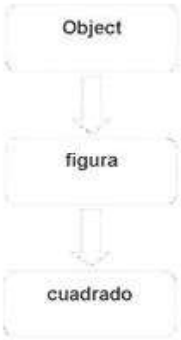

Figura 4.9. Jerarquía de clases

```
public class figura{
   String color;
   public void setColor(String s){color=s;}
   public String getColor(){return color;}
}
```

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

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

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

Recuerda
Las clases heredan el comportamiento de sus antecesores (padres) pero no lo heredan de otras subclases (hermanos).

Imaginemos que queremos testear el comportamiento de la jerarquía anterior. Para ello crearemos la siguiente clase:

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

En esta clase se puede observar cómo se llama a métodos de la superclase *figura* y de la subclase *cuadrado*. Solamente hemos tenido que crear una clase *cuadrado* puesto que los atributos y métodos de la clase *figura* los ha heredado la clase *cuadrado*.

RESUMEN DEL CAPÍTULO

En este capítulo se profundiza y se amplían los conceptos que se vieron en el Capítulo 2. Una vez estudiado este capítulo el alumno se dará cuenta que en el Capítulo 2 se vieron solamente los conceptos básicos para poder realizar nuestros primeros programas. En este capítulo se entrará a estudiar en profundidad el concepto de clase así como los miembros y estructura de una clase. Se verán los métodos recursivos. El alumno deberá de comprender a fondo el concepto de recursividad para poder realizar los ejercicios propuestos. También se verán en este capítulo algunas de las características más importantes de Java, como son las interfaces y la herencia. Se recomienda al alumno un estudio exhaustivo de todos los apartados para luego poder profundizar más a lo largo del siguiente tema.

EJERCICIOS RESUELTOS

1A. Realiza una clase con un método factorial que utilizando la recursividad genere el factorial de un número dado.

Solución:

Como se dijo antes en el desarrollo del capítulo, para resolver un caso recursivo debemos encontrar:

a. Una fórmula o proceso que reduzca la complejidad y nos vaya acercando a la solución.

En nuestro caso sabemos que por ejemplo factorial(4) es 4 * factorial(3), lo que es igual factorial(num) = num * factorial(num -1).

b. Un caso base que hace que nuestra recursividad no sea infinita.

En nuestro caso será factorial(0) = 1.

En la siguiente figura se muestra el factorial de una manera más matemática:

$$n! \quad \begin{cases} \text{Si } n = 0 \rightarrow 1 \\ \text{Si } n >= 1 \rightarrow (n-1)! \cdot n \end{cases}$$

Figura 4.10. Algoritmo factorial

El código que resuelve el factorial es el siguiente:

```
class Test {
    public static int factorial(int num)
    {
          if (num == 0) return 1;
          return num * factorial(num-1);
    }
    public static void main(String[] args) {
          System.out.println("El factorial de 0 es : "+factorial(0));
          System.out.println("El factorial de 1 es : "+factorial(1));
          System.out.println("El factorial de 2 es : "+factorial(2));
          System.out.println("El factorial de 3 es : "+factorial(3));
          System.out.println("El factorial de 4 es : "+factorial(4));
          System.out.println("El factorial de 5 es : "+factorial(5));
    }
}
```

1B. Para el programa anterior, modifica el método para generar el factorial de forma iterativa.

Solución:

```
class Test {
    public static int factorial(int num)
    {
          int factorial=1;
          while(num>0){

                factorial *= num;
                num--;
          }
          return factorial;
    }
    public static void main(String[] args) {
          System.out.println("El factorial de 0 es : "+factorial(0));
          System.out.println("El factorial de 1 es : "+factorial(1));
          System.out.println("El factorial de 2 es : "+factorial(2));
          System.out.println("El factorial de 3 es : "+factorial(3));
          System.out.println("El factorial de 4 es : "+factorial(4));
          System.out.println("El factorial de 5 es : "+factorial(5));
    }
}
```

¿Qué ventaja tiene la solución iterativa frente a la recursiva?

La solución iterativa tiene la ventaja de la rapidez de ejecución (más eficiente). La solución recursiva es más lenta pero en muchas ocasiones (en esta concretamente no) es mucho más fácil de entender y más fácil de codificar con lo cual, aunque tengamos un código menos eficiente es mucho más fácil de comprender y mantener.

Como ejercicio complementario modifica los métodos anteriores para que cuando se introduzca un número menor a 0 el método muestre un mensaje de error y no realice ningún cálculo.

2. Realiza un programa con un método recursivo que muestre por pantalla la siguiente serie:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Solución:

Esta serie es un problema ampliamente conocido como serie de Fibonacci. Se caracteriza por lo siguiente:

Fibonacci(0) = 0

Fibonacci(1) = 1

Y para todos los demás números:

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

El código que resuelve la serie es el siguiente:

```
class Test {
    public static int fibonacci(int num)
    {
          if (num == 0) return 0;
          if (num == 1) return 1;
          return fibonacci(num-1)+fibonacci(num-2);
    }
    public static void main(String[] args) {
          for (int i=0;i<10;i++){
                System.out.print(fibonacci(i)+", ");
          }
    }
}
```

3. Realiza un programa que utilizando recursividad muestre por pantalla la siguiente pirámide:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Figura 4.11. Triángulo numérico

El programa podrá generar una pirámide de cualquier número de filas.

Solución:

¿Cómo resolver el problema?

El primer paso, desde mi punto de vista, es observar el problema desde otra perspectiva. El mirar el problema como una pirámide evita poder resolverlo de forma satisfactoria. Si modificamos el programa e intentamos resolverlo como si fuese una pirámide tendría más sentido. La pirámide anterior en forma matricial sería la siguiente:

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1

De esta matriz podemos concluir lo siguiente:

- Los elementos con valor 0 no se mostrarán.
- El primer elemento de cada fila vale 1.
- Los elementos cuya columna sea menor de 1 valen 0.
- Los elementos cuya columna sea mayor que la fila valen 0.
- Para los demás elementos, el elemento(fila, columna) es igual al elemento(fila-1,columna)+elemento(fila-1,columna-1)

Una vez que tenemos claros estos principios el siguiente paso es programar el método. Observa que es la última regla la que implica que este método sea recursivo.

Con esto ya tenemos casi realizado el programa. La programación de los bucles no parece compleja a estas alturas pero lo que puede resultar más complejo es "enderezar" la pirámide introduciendo espacios en blanco antes de imprimir cualquier número. En nuestro caso se ha resuelto el problema añadiendo numfilas-filaactual o mejor dicho numfilas-i espacios en blanco en cada línea y de esta manera se imprimirán 4, 3, 2, 1 y 0 espacios en blanco en cada línea.

```
class piramide {
    public static int elemento(int fila, int columna){
          if (columna == 1 ) return 1;
          if (columna < 1 || columna > fila) return 0;
          return elemento(fila-1,columna)+elemento(fila-1,columna-1);
    }
    public static void main(String[] args) {
          int numfilas = 5;
          for (int i=1; i<(numfilas+1) ; i++){
                for (int e=0; e<(numfilas - i);e++)System.out.print(" ");
                for (int j=1; j<(numfilas+1) ; j++){
                      int dato = elemento(i,j);
                      if (dato > 0 ) System.out.print(dato+" ");
```

```
                }
                System.out.println("");
            }
        }
    }
```

Este programa funcionará con una pirámide de cualquier número de filas, bastará solamente con cambiar el valor a la variable numfilas por el número de filas deseado. Nótese que cuando se comienzan a mostrar números de dos y más cifras la alineación de la pirámide se pierde. Se pide al alumno como ejercicio complementario que solvente este problema.

4. Realiza un programa que utilizando la recursividad muestre por pantalla la siguiente pirámide:

```
          1
        1 1 1
      1 2 3 2 1
    1 3 6 7 6 3 1
```

Figura 4.12. Triángulo numérico (II)

El programa podrá generar una pirámide de cualquier número de filas.

Solución:

```
class piramide {
    public static int elemento(int fila, int columna){
       if (fila < 1 || columna < 1 ) return 0;
       if (columna == 1 ) return 1;
return elemento(fila-1,columna)+elemento(fila-1,columna-1)+elemento(fila-1,columna-2);
    }
    public static void main(String[] args) {
           int numfilas = 4;

           for (int i=1; i<(numfilas+1) ; i++){
                 for (int e=0; e<(numfilas - i);e++)System.out.print("  ");
                 for (int j=1; j<(2*numfilas+1) ; j++){
                       int dato = elemento(i,j);
                       if (dato > 0 ) System.out.print(dato+" ");
                 }
                 System.out.println("");
           }
    }
}
```

Este programa funcionará con una pirámide de cualquier número de filas, bastará solamente con cambiar el valor a la variable numfilas por el número de filas deseado. Nótese que cuando se comienza a mostrar números de dos o más cifras la alineación de la pirámide se pierde. Se pide al alumno como ejercicio complementario que solvente este problema.

5. Realiza una clase TransformaBase la cual trasforme y cambie de base números decimales.

Solución:

```
class TransformaBase {
    public static void muestraCifra(int dat) {
        if (dat<10) {
            System.out.print(dat);
        } else {
    dat-=10;
    char c = (char)('A'+ dat);
            System.out.print(c);
        }
    }
    public static void transforma(int dato, int base) {
        if (base > dato) {
            muestraCifra(dato);
        } else {
            transforma(dato/base,base);
            muestraCifra(dato%base);
        }
    }
   public static void main(String[] args) {
        transforma(8,2);
        System.out.println("");
        transforma(12,16);
        System.out.println("");
        transforma(13,8);
        System.out.println("");
   }
}
```

Como ejercicio complementario se pide al alumno que cree otro método en el cual se transformen los números de forma iterativa.

6. En la siguiente clase indica cuál es el atributo de instancia y cuál es el atributo de clase:

```
public class unaClase {
    public static int a = 20;
    public int b = 13;
}
```

Solución:

- La variable de clase es la a.
- La variable de instancia es la b.

EJERCICIOS PROPUESTOS

- **1.** Realiza un programa que muestre por pantalla el siguiente cuadrado:

```
1  1  1  1  1
1  2  3  4  5
1  3  6  10 15
1  4  10 20 35
1  5  15 35 70
```

Figura 4.13. Matriz numérica

El programa podrá generar un cuadrado de cualquier dimensión. Utiliza la recursividad para resolver el problema.

- **2.** (Ejercicio de dificultad alta) Realiza los ejercicios resueltos de forma recursiva de tal forma que la solución ahora sea iterativa.

- **3.** (Ejercicio de dificultad alta) Para los ejercicios resueltos de pirámides crea como añadido un método recursivo que muestre la suma de los valores mostrados por pantalla.

- **4.** Crea en tu equipo un paquete Utilidades.mates con dos clases sumar y potenciar. La clase sumar tendrá un método int suma(int,int) el cual devolverá la suma de los dos parámetros introducidos y la clase potenciar tendrá un método int potencia(int,int) el cual devolverá el resultado de elevar el primer parámetro al segundo parámetro. Realiza un programa que haga uso de este paquete.

- **5.** Realiza una clase *pez* la cual tendrá un miembro nombre de tipo *String* el cual podrá ser heredado por sus subclases. Realiza un método getNombre y otro setNombre. Utiliza el objeto *this* en estos métodos. Implementa en esta clase el método *clone()* así como el método *equals()* para poder hacer una comparación en profundidad. Realiza un programa que haga un testeo en profundidad de las características de esta clase.

- **6.** Para la clase *pez* anterior, crea un miembro privado entero numpeces común a todos los objetos *pez* el cual cuente el número de peces creados. Crea un programa que compruebe que esta variable se incrementa cada vez que se crea un objeto *pez*.

- 7. Para el objeto *pez* anterior crea un constructor copia. Comprueba este constructor mediante un programa.

- 8. Crea una clase *prueba* en el que tenga dos métodos (*primero* y *segundo*). El método *segundo* llamará al método *primero* dos veces, de forma normal y utilizando *this*. Verifica que ambas llamadas son equivalentes.

- 9. ¿Qué mostrará el siguiente programa por pantalla?

```
public class bebe {
    bebe (int i) {
          this("Soy un bebe consentido");
          System.out.println("Hola, tengo " + i + " meses");
    }
    bebe (String s) {
          System.out.println( s );
    }
    void berrea() {
          System.out.println("Buaaaaaaaaaa");
    }
   public static void main(String[] args) {
                new bebe (8).berrea();
   }
}
```