

Artículo publicado originalmente en la revista [S ólo Programadores](#)

En el último artículo de la serie comenzamos a presentar el soporte que Java proporciona para la programación orientada a objetos, un paradigma de programación bastante diferente de la programación estructurada. En este artículo terminaremos de cubrir la programación orientada a objetos en Java describiendo en mayor profundidad las implicaciones de usar herencia, y presentando las interfaces y los paquetes.

1Algunos apuntes más sobre la herencia

En el capítulo anterior de esta serie vimos cómo la herencia permite a una clase heredar el código de su clase padre. También vimos cómo, sobrescribiendo los métodos del padre, la clase hija puede modificar el comportamiento del padre.

En ocasiones, cuando se diseña un código el programador no quiere que una de sus clases sea modificada; dado el diseño interno de la clase ésta debe emplearse tal cual, o no emplearse en absoluto (ya conocemos una clase que ha sido diseñada de este modo: la clase `Math`). Es posible impedir que el comportamiento de nuestra clase sea extendido o modificado mediante la herencia empleando el modificador `final`. Del mismo modo que cuando este modificador se emplea al definir una variable viene a indicar que dicha variable nunca va a cambiar de valor, cuando se aplica a una clase su significado es que "la clase no puede modificarse". Y el mecanismo para modificar una clase es la herencia; por tanto, será imposible heredar de una clase `final`. Así, por ejemplo, si tenemos la clase:

```
public final class AlgunaClase {...}
eintentamosdefinirunaclasequeherededeella:
public class OtraClase extends AlgunaClase {...}
```

el compilador producirá un error, ya que la clase que estamos intentando extender es una clase `final`. En ocasiones no queremos eliminar completamente la posibilidad de extender o modificar algunas partes de nuestra clase. Pero sí existe alguna funcionalidad que no queremos que se modifique bajo ningún concepto; esto es, existen algunos métodos que no queremos que sean modificados (sobrescritos) por las clases hijas. Este efecto puede conseguirse sin declarar la clase `final`, pero indicando que los métodos que no queremos que sean sobrescritos son finales:

```
publicfinalStringalgunMetodo(){...}
```

Si una clase hija define otro método con el mismo nombre y parámetros que el nuestro (es decir, si define un método que sobrescriba a nuestro método) el compilador producirá un error.

1.1 Clases abstractas

El último punto que trataremos (por lo de ahora) referente a la herencia es la utilidad de las clases abstractas. En el número anterior lo vimos a nivel teórico: una clase abstracta representa una categoría de objetos "abstractos" del mundo real. Por ejemplo, podría representar los "seres vivos". Esa es una categoría abstracta, en el sentido de que no existe ningún ser vivo como un ente puro. Existen perros, gatos, personas, plantas... y todos ellos son seres vivos. Pero no hay ningún ente que sea un ser vivo y que no pertenezca a otra categoría más específica.

Estas categorías de objetos abstractos juegan un papel muy importante para ayudarnos a organizar la información dentro de nuestro cerebro. De un modo similar, las clases abstractas pueden ayudarnos a organizar nuestro código fuente. Una clase abstracta representa una entidad que contiene ciertas propiedades y funcionalidad comunes a un conjunto de clases. Podemos decir que la clase abstracta constituye una abstracción (valga la redundancia) de las clases concretas que derivan de ella. Sin embargo, por sí sola no tiene una funcionalidad completa; no tiene sentido. Pero puede ser útil para reutilizar, mediante la herencia, sus propiedades y su funcionalidad.

Supongamos que en un programa tenemos que representar un conjunto de funciones matemáticas. Estas funciones matemáticas deberán ser capaces de hacer tres cosas: deben poder recibir un valor de x y devolver el valor de $f(x)$ correspondiente; deben permitir devolver una representación textual de la función que representan (por ejemplo, " $3.6x^2 + 5.0x + 2$ "); y deben contar con un método al cual se le pasa un valor de x , y muestra por consola la representación textual de la función, y su valor en dicho punto. Tenemos que representar varios tipos de funciones: lineales, cuadráticas, exponenciales... cada una de estas funciones va requerir de operaciones diferentes para evaluar la función en x . La forma de generar su representación textual también va a depender de cada función. Sin embargo, la operación de mostrar dicha representación textual en la consola y el resultado de evaluar la función en un punto puede implementarse una sola vez y reutilizarse para todas las funciones. Para ello necesitamos emplear una clase abstracta, como la que se muestra en el listado 1.

```
//LISTADO 1: Clase que representa cualquier función matemática. Es imposible crear instancias de ella, ya que la clase es abstracta
public abstract class FuncionAbstracta {
    public void mostrarResultadoEvaluar(float x){
        System.out.println("El valor de la función " +
            getRepresentacion() + " en "+x+ " es: "+ evalua(x));
    }

    public abstract float evalua(float x);
    public abstract String getRepresentacion();
}
```

En el listado 1 podemos observar como la clase que hemos declarado emplea el modificador `abstract`. También hay dos métodos que tienen ese modificador. Cuando este modificador se emplea en un método quiere decir que vamos a declarar el método, pero que no vamos a proporcionar ninguna implementación para él. Una clase que contenga métodos abstractos obligatoriamente tiene que ser abstracta. Si heredamos de una clase abstracta y queremos que la clase hija no sea abstracta, obligatoriamente tendremos que sobrescribir todos los métodos abstractos de la clase padre y proporcionar una implementación para ellos.

En nuestra clase abstracta existe un método que sí que tiene implementación: `mostrarResultadoEvaluar(float x)`. Es el método que muestra la función y el resultado de evaluar la función por consola. Observa que para ello emplea los otros dos métodos abstractos!. No hay ningún problema en ello: nunca nadie va a poder crear un objeto de esta clase, ya que es abstracta. Y si alguna clase no abstracta hereda de nuestra clase, obligatoriamente va a tener que proporcionar una implementación para los métodos abstractos. Por tanto, cuando se cree un objeto de la clase hija tenemos garantizado que los métodos `evalua(float x)` y `getRepresentacion()` habrán sido definidos.

En el listado 2 podemos ver dos clases que heredan de la clase `FuncionAbstracta`. La primera implementa una función lineal ($ax + b$), mientras que la segunda implementa una función cuadrática ($ax^2 + bx + c$). Empleando BlueJ puedes comprobar como es posible crear objetos de la clase `FuncionLineal` o de la clase `FuncionCuadratica`, aunque no de la clase `FuncionAbstracta`. También puedes comprobar como el método `mostrarResultadoEvaluar(float x)` de la clase `FuncionLineal` y de la clase `FuncionCuadratica` funcionan perfectamente, y hacen dos cosas

diferentes (uno muestra y evalúa la función lineal, y el otro muestra y evalúa la función cuadrática). A pesar de ello, sólo tuvimos que escribir una vez su código, en la clase padre. La herencia hizo el resto de la magia.

//LISTADO 2: Dos clases que heredan de la clase FuncionAbstracta e implementan una función lineal y una función cuadrática

```
public class FuncionLineal extends FuncionAbstracta {
    float a,b;

    public FuncionLineal(float a,float b){
        this.a = a;
        this.b = b;
    }

    public float evalua(float x){
        return a*x+b;
    }

    public String getRepresentacion(){
        return a+"x + "+b;
    }
}

public class FuncionCuadratica extends FuncionAbstracta {
    float a,b,c;

    public FuncionCuadratica(float a,float b,float c){
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public float evalua(float x){
        return (float)(a*Math.pow(x,2))+b*x+c;
    }

    public String getRepresentacion(){
        return a+"x^2 + " + b + "x + " + c;
    }
}
```

2 Las interfaces

En Java no está soportada la herencia múltiple, esto es, no está permitido que una misma clase pueda heredar de varias clases padres. En principio esto pudiera parecer una propiedad interesante que le daría una mayor potencia al lenguaje de programación. Sin embargo los creadores de Java decidieron no implementar la herencia múltiple por considerar que ésta añade al código una gran complejidad que no se ve compensada con la potencia que proporciona (lo que hace que muchas veces los programadores que emplean lenguajes que sí la soportan no lleguen a usarla).

Sin embargo, para no privar completamente a Java de la potencia de la herencia múltiple, sus creadores introdujeron un nuevo concepto: el de interface sin código sobre madres. Una interfaz es similar a una clase, pero tiene dos diferencias: sus métodos están vacíos, no hacen nada, y a la hora de definirla en vez de utilizar la palabra clave class se utiliza interface.

Aunque en este momento no le veamos demasiado sentido, podríamos hacer que la clase abstracta que representa una función genérica del apartado anterior implementarse una interfaz que defina todas las operaciones que deben ser comunes para todas las funciones. Esto nos proporcionaría un nivel de indirección adicional que puede resultar muy útil en ciertos casos. Elaboraremos más sobre este punto a lo largo de esta serie de artículos. Por lo de ahora simplemente vamos a crearnos que es una buena idea hacer que la clase FuncionAbstracta herede de la interfaz que se muestra en el listado 3.

//LISTADO 3: Interfaz que representa una función cualquiera

```
public interface Funcion{  
public float evalua(float x);  
public void mostrarResultadoEvaluacion(float x);  
public String getRepresentacion();  
}
```

Cabe preguntarnos cuál es el uso de una interfaz si sus métodos están vacíos. Cuando una clase implementa una interfaz lo que estamos haciendo es una promesa de que esa clase va a implementar todos los métodos de la interfaz en cuestión. Si la clase que implementa la interfaz no sobrescribiese alguno de los métodos de la interfaz automáticamente esta clase se convertiría en abstracta y no podríamos crear ningún objeto de ella.

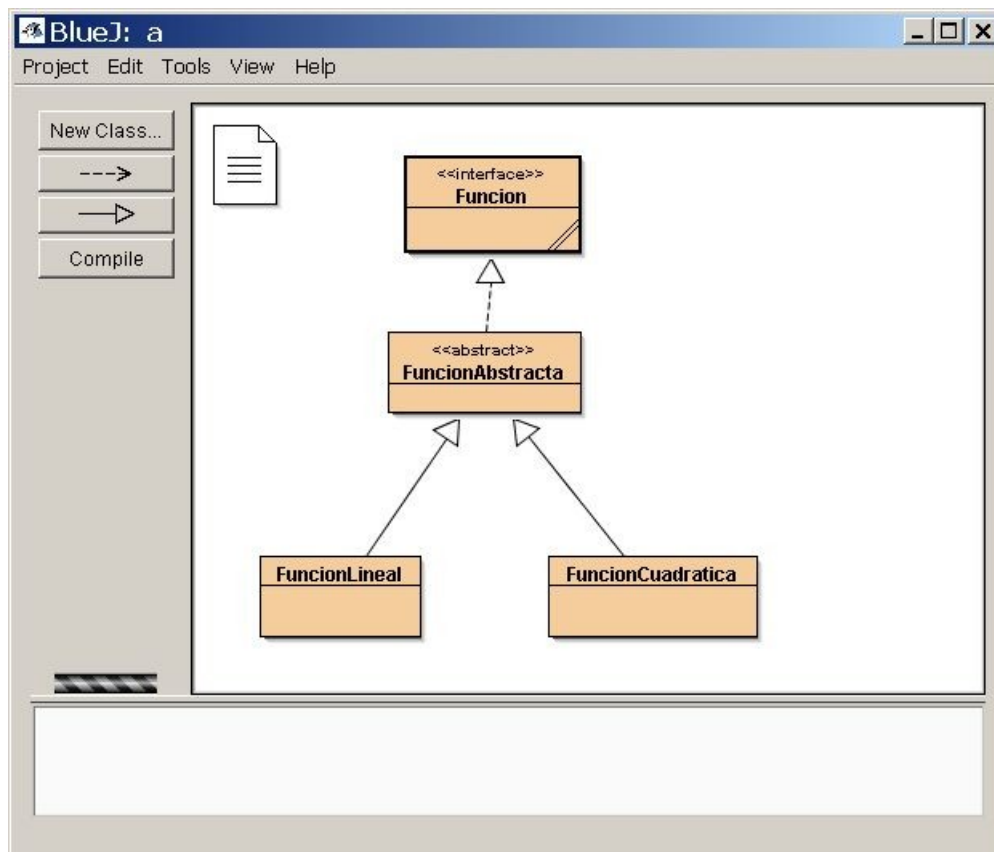


FIGURA 1: Jerarquía de clases del ejemplo de las funciones incluyendo la interfaz Funcion

Si ahora queremos que la clase FuncionAbstracta implemente la interfaz Funcion debemos definirla como se muestra en el listado 4. La clase tiene que ser obligatoriamente abstracta, ya que no sobrescribe dos métodos de la interfaz. Las clases que deriven de ella (como, por ejemplo, la clase FuncionLineal) deberán sobrescribir los dos métodos que quedan por sobrescribir de la interfaz.

Tras hacer estos cambios a la clase FuncionAbstracta , tanto la clase FuncionLineal como la clase FuncionCuadratica no necesitan ninguna modificación para seguir funcionando correctamente. En la figura 1 podemos ver cómo BlueJ representa la jerarquía de clases de nuestro ejemplo.

```
//LISTADO4:LaclaseFuncionAbstractaahoraimplementalainterfazFuncion
public abstract class FuncionAbstracta implements Funcion{
    public void mostrarResultadoEvaluar(float x){
        System.out.println("El valor de la funcion "+
            getRepresentacion() +" en "+x+ " es: "+ evalua(x));
    }
}
```

Las variables que se definen dentro de una interfaz llevan todas el atributo final (aunque nuestro código no lo indique), y es obligatorio darles un valor dentro del cuerpo de la interfaz. Además no pueden llevar modificadores private ni protected , sólo public . Su función es la de ser una especie de constantes para todos los objetos que implementen dicha interfaz.

Por último, mencionar que aunque una clase sólo puede heredar propiedades de otra clase, puede implementar cuantas interfaces se desee, recuperándose así parte de la potencia de la herencia múltiple. Para ello, basta con poner la lista de interfaces a implementar después de la palabra reservada implements , separando los nombres de las interfaces con comas. En el listado 5 mostramos como una clase puede implementar varias interfaces.

```
//LISTADO5:Unaclasepuedeimplementar cuantas interfaces se desee
interface Interfaz1{
    public void metodo1();
}

interface Interfaz2{
    public void metodo2();
}

interface Interfaz3{
    public void metodo3();
}

public class AlgunaClase implements Interfaz1, Interfaz2, Interfaz3 {
    //dentro de esta clase estamos obligados a sobrescribir
    // metodo1(),metodo2() y metodo3();
    ...
}
```

3 Los packages

A estas alturas deberías tener claro que una clase tiene una parte privada que oculta a los demás y que no es necesario conocer para poder acceder a su funcionalidad. Si hacemos cambios a la parte privada de la clase, mientras se respeta la parte pública, cualquier código cliente que emplee la clase no se dará cuenta de dichos cambios.

Imagínate que tú y un compañero vais a construir un programa complejo juntos. Os repartís el trabajo entre los dos y cada uno de vosotros implementa su parte como un montón de clases Java. Cada uno de vosotros en su código va a emplear parte de las clases del otro. Por tanto, os ponéis de acuerdo en las interfaces de esas clases. Sin embargo, cada uno de vosotros para construir la funcionalidad de esas clases probablemente se apoye en otras clases auxiliares. A tu compañero le dan igual las clases auxiliares que tú emplees. Es más, dado que el único propósito de esas clases

es servir de ayuda para las que realmente constituyen la "interfaz" de tu parte del trabajo sería contraproducente que él pudiese acceder a esas clases que son detalles de implementación: tú en el futuro puedes decidir cambiar esas clases, modificándolas o incluso eliminándolas.

Dada esta situación ¿no sería interesante poder "empaquetar" tu conjunto de clases de tal modo que ese "paquete" sólo dejase acceder a tu compañero a las clases que tú quieras y oculte las demás. Esas clases a las que se podría acceder serían la interfaz de ese "paquete"; serían la parte pública del paquete. Dentro del paquete tú puedes tener clases adicionales. Pero esas no son accesibles por tu compañero y podrás cambiarlas en cualquier momento sin que él tenga que modificar su código. Es la misma idea que hay detrás de una clase pero llevada a un nivel superior: una clase puede definir cuáles de sus partes son accesibles y no accesibles para los demás. El paquete permitiría meter dentro cuantas clases quieras pero mostraría al exterior sólo aquellas que tú indiques. Parece una buena idea ¿no

Pues esa es precisamente la utilidad de los package en Java. Agrupar un montón de clases y permitir indicar cuáles serán accesibles para los demás y cuáles no. Para empaquetar las clases simplemente debemos poner al principio del archivo donde definimos la clase, en la primera línea que no sea un comentario, una sentencia que indique a qué paquete pertenece:

```
package mipaquete;
```

Una clase que esté en el paquete mipaquete debe situarse dentro de un directorio con nombre mipaquete . En Java los paquetes se corresponden con una jerarquía de directorios. Por tanto, si para construir un programa quiero emplear dos paquetes diferentes con nombres paquete1 y paquete2 en el directorio de trabajo debo crear dos subdirectorios con dichos nombres y colocar dentro de cada uno de ellos las clases correspondientes. En la figura 1, el directorio de trabajo desde el cual deberíamos compilar y ejecutar la aplicación es Paquetes. En cada uno de los dos subdirectorios colocaremos las clases del paquete correspondiente.



FIGURA 2: En Java, los paquetes se corresponden con una estructura de directorios

Cuando una clase se encuentra dentro de un paquete el nombre de la clase pasa a ser NombrePaquete.NombreClase . Así, la clase ClasePaquete1 que se encuentra físicamente en el directorio paquete1 y cuya primera línea de código es:

```
package paquete1;
```

tendrá como nombre completo paquete1.ClasePaquete1 . Si deseamos, por ejemplo, ejecutar el método main de dicha clase debemos situarnos en el directorio Paquetes y teclear el comando:

```
java paquete1.ClasePaquete1
```

Cuando en una clase no se indica que está en ningún paquete, como hemos hecho hasta ahora en todos los ejemplos de esta serie de artículos, esa clase se sitúa en el "paquete por defecto" (default package). En ese caso, el nombre de la clase es simplemente lo que hemos indicado después de la palabra reservada class , sin precederlo del nombre de ningún paquete.

Es posible anidar paquetes; por ejemplo, en el directorio paquete1 puedo crear otro directorio con nombre paquete11 y colocar dentro de él la clase OtraClase . La primera línea dentro del fichero de código fuente de dicha clase deberá ser:

```
package paquete1.paquete11;
```

y el nombre de la clase será paquete1.paquete11.OtraClase .

¿Cómo indico qué clases serán visibles en un paquete y qué clases no serán visibles. Cuando explicamos cómo definir clases vimos que antes de la palabra reservada class podíamos poner un modificador de visibilidad. Hasta ahora siempre hemos empleado el modificador public . Ese modificador significaría que la clase va a ser visible desde el exterior; es decir, forma parte de la interfaz del paquete. Si no ponemos el modificador public la clase tendrá visibilidad de paquete, es decir, no será visible desde fuera del paquete pero sí será visible para las demás clases que se encuentren en el mismo paquete que ella. Aunque hay más opciones para el modificador de visibilidad de una clase, para un curso básico como éste estas dos son suficientes.

¿Y cómo hacemos para emplear clases que se encuentren en otros paquetes diferentes al paquete en el cual se encuentra nuestra clase. Para eso es precisamente para lo que vale la sentencia import : para indicar que vamos a emplear clases de paquetes diferentes al nuestro. Así, si desde la clase MiClase , que se encuentra definida dentro de paquete1 , quiero emplear la clase OtraClase , que se encuentra en paquete2 , en MiClase debo añadir la sentencia:

```
import paquete2.OtraClase;
```

A partir de ese momento, si OtraClase era pública, podré acceder a ella y crear instancias. El importar una clase sólo será posible si dicha clase forma parte de la interfaz pública del paquete; en caso contrario el compilador dará un error. La sentencia:

```
import paquete2.*;
```

hace accesibles todas las clases públicas que se encuentren en paquete2. El importar un paquete nunca es recursivo; es decir, al escribir la sentencia anterior sólo importamos el contenido de paquete2. Si existiese otro paquete con nombre paquete2.subpaquete , esa sentencia no está importando las clases de subpaquete.

Una opción alternativa a emplear la sentencia import es emplear el nombre completo de la clase cuando vayamos a acceder a ella para crear un objeto o para invocar uno de sus métodos estáticos. Así, si no hemos importado las clases del paquete2 , para crear un objeto de una de sus clases debemos escribir:

```
paquete2.OtraClase objeto = new paquete2.OtraClase ();
```

Es posible que las clases que estén dentro de un paquete hereden de clases que forman la parte pública de otro paquete. En este caso, se aplican las normas que ya hemos presentado en el artículo anterior para la herencia: la clase hija podrá acceder a la parte pública y protegida de la clase padre. Observa que, si no está involucrada la herencia, una clase nunca podrá acceder a las partes no públicas de las clases de otro paquete.

En los listados 7 y 8 podemos ver dos clases, en cada uno de ellos, que pertenecen, respectivamente a los paquetes paquete2 y paquete1 . Las clases de paquete1 son las que van a

usar las clases del segundo paquete. En concreto, la clase ClasePaquete1 empleará a una clase del segundo paquete (creará a una instancia de ella e invocará métodos) y la clase ClasePaquete1Herencia heredará de una clase del otro paquete. En el segundo paquete tendremos una clase pública con nombre ClasePaquete2 ; esta clase constituye la interfaz del segundo paquete. También tendremos una clase que es inaccesible fuera del paquete, ClasePaquete2Privada de , pero que es empleada por la primera clase. Esa clase es "un detalle de implementación" de este paquete. Si en el futuro la modificamos, la eliminamos, creamos más clases para repartir sus responsabilidades... ningún código que emplee paquete2 se dará cuenta de dichos cambios ya que nunca conoció la existencia de dicha clase.

```
//LISTADO7:Estecdigodemuestralasdistintasvisibilidadesentreclasesqueestnendistintospaquetes
```

```
package paquete2;
class ClasePaquete2Privada {
void visibilidadPublica() {
System.out.println("Mensaje del mtodo con visibilidad pblica...");
}
void visibilidadPaquete(){
System.out.println("Mensaje del mtodo con visibilidad de paquete...");
}

protected void visibilidadProtegida(){
System.out.println("Mensaje del mtodo protegido...");
}

private void visibilidadPrivada (){
System.out.println("Mensaje del mtodo privado...");
}
}
//comienza una nueva clase, la pública de este paquete
package paquete2;

public class ClasePaquete2{

public void saludar() {
System.out.println("Hola");
...
//aquí usamos los "detalles de implementación" del paquete
ClasePaquete2Privada objeto2 = new ClasePaquete2Privada();
objeto2.visibilidadPublica();
objeto2.visibilidadPaquete ();
objeto2.visibilidadProtegida ();
}
void visibilidadPaquete(){
System.out.println("Mensaje del mtodo con visibilidad de paquete");
}

protected void visibilidadProtegida(){
System.out.println("Mensaje del mtodo con visibilidad protegida");
}

private void privado (){
System.out.println("Mensaje del mtodo privado");
}
}
}
```

```
//LISTADO 8:
```

```
package paquete1;
```

```

import paquete2.*;

public class ClasePaquete1{
publicstaticvoidmain(String[]args){
    ClasePaquete2 objeto = new ClasePaquete2();
    objeto.saludar();
}
}

package paquete1;
//comienza la segunda clase
import paquete2.*;
public class ClasePaquete1Herencia extends ClasePaquete2{

publicstaticvoidmain(String[]args){
    ClasePaquete1Herencia objeto = new ClasePaquete1Herencia();
    objeto.visibilidadProtegida();
    objeto.saludar();
}
}

```

La salida que produce la ejecución del método main de ClasePaquete1 se muestra en la figura 3, y la que produce el método main de ClasePaquete1Herencia se muestra la figura 4.

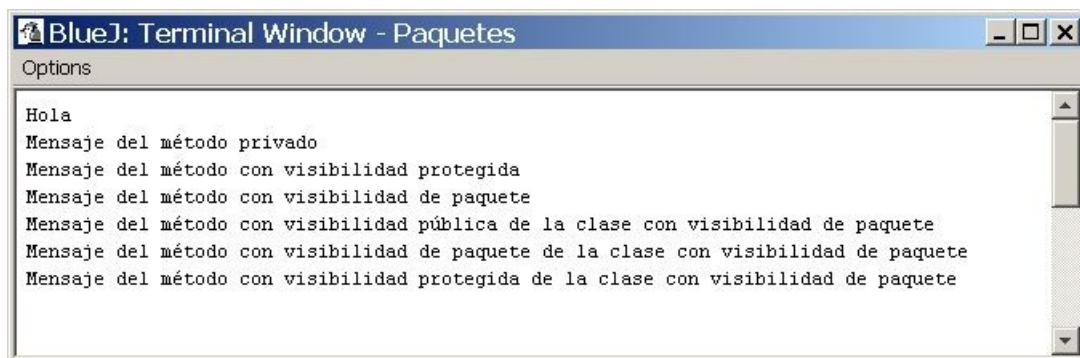


FIGURA 3: Resultado de la ejecución del método main de ClasePaquete1

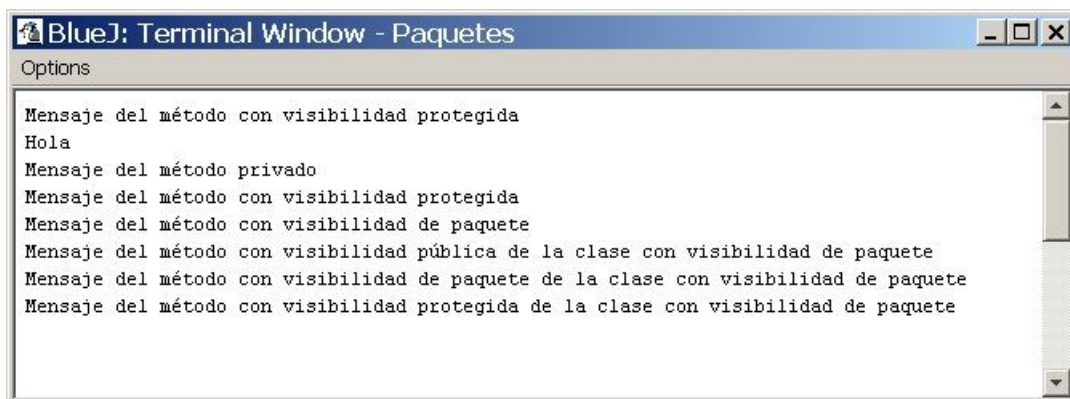


FIGURA 4: Resultado de la ejecución del método main de ClasePaquete1Herencia

3.1 Sobre el nombre de los paquetes

Por último, para terminar con este apartado dedicado a los packages, comentaremos un convenio de nomenclatura muy extendido para los nombres paquetes. Además de proporcionar niveles de visibilidad para las clases, los paquetes evitan colisiones entre espacios de nombres. Los paquetes me permiten definir una clase Cliente, y permiten que otro programador tenga también su clase Cliente y que ambas clases sean distinguibles, mientras se encuentren en paquetes diferentes. Para que haya una colisión a nivel de clase, las clases deberán tener el mismo nombre y estar dentro del mismo paquete.

Para evitar que haya colisiones de paquetes, por convenio, suele emplearse URLs (que el World Wide Consortium garantiza que son siempre únicas, es decir, no hay dos URLs iguales) para nombrar paquetes. Por ejemplo, si la editorial de esta revista quisiese crear su propio paquete siguiendo este convenio en el nombre del paquete debería ser `com.revistas.profesionales.XXX`. No hay nada dentro de la sintaxis de Java que nos obligue a seguir este convenio, pero es recomendable seguirlo y la mayor parte del software profesional lo hace.

3.2 El paquete `java.lang`

Cuando queremos acceder a la funcionalidad de algún paquete de la librería estándar de Java también es necesario importarlo. Esto ya lo hemos hecho en alguna ocasión: por ejemplo, cuando quisimos usar la clase `Random` del paquete `java.util`. Sin embargo, ha habido muchas ocasiones en las que hemos usado clases de la librería de Java sin importarlas: `Math` o `System` son dos ejemplos.

Dentro de las librerías estándar de Java hay un paquete especial: `java.lang`. Todos los programas Java importan todas las clases que hay en ese paquete por defecto. A todos los efectos, al principio de cualquier programa Java hay un `import java.lang.*` implícito. Esto se debe a que en ese paquete se encuentran librerías de uso muy frecuente en cualquier programa y es virtualmente imposible escribir un programa en Java sin necesitar usar una o varias de las clases de este paquete. Por ello este paquete siempre "está importado por defecto". Y, obviamente, es en él donde se encuentran las clases `Math` y `System`.

4 Un ejemplo donde se ponga todo esto junto

Vamos a desarrollar un ejemplo que va a incorporar muchos conceptos relacionados con la herencia, con la programación orientada a objetos y con la organización del código en paquetes que hemos estado viendo a lo largo de este artículo y del anterior. Supongamos que tenemos que desarrollar un software que permita evaluar múltiples funciones matemáticas diferentes en un mismo punto del eje x . Este software debe contar con un "contenedor" de funciones, al cual se deberán poder añadir todas las funciones matemáticas sobre las cuales queremos trabajar. Será posible pedirle a esta especie de contenedor que evalúe todas las funciones que contiene en un determinado punto, y como respuesta a esta petición el contenedor deberá mostrar cada una de las funciones y el valor de la función en el punto indicado.

El software deberá ser extensible, en el sentido de que los usuarios podrán crear las funciones matemáticas que deseen y emplearlas con él. No obstante, deberemos proporcionar implementaciones de algunas funciones matemáticas básicas (funciones lineales, cuadráticas y exponenciales, por ejemplo).

Vamos a organizar nuestro código en dos paquetes diferentes. Por un lado tendremos un paquete donde colocaremos el contenedor de las funciones y una interfaz que deberán implementar todas las funciones que vayan a ser gestionadas por nuestro contenedor. En ese paquete también

colocaremos una clase abstracta que implementa la interfaz y proporciona cuerpo para una de sus funciones. El contenido de este paquete, que denominaremos funciones, se muestra en el listado 9.

```
//LISTADO 9: Contenido del paquete funciones de nuestro ejemplo
package funciones;
// el cuerpo es igual que el que se muestra en el listado 3
public interface Funcion {...}

package funciones;
// el cuerpo es igual que el que se muestra en el listado 4
public abstract class FuncionAbstracta implements Funcion {...}

package funciones;

public class ContenedorDeFunciones {
private int numFunciones=0;
    private Funcion[] funciones = new Funcion[10];

    public void anhadirFuncion(Funcion f) {
        if (numFunciones < 10) {
            funciones[numFunciones] = f;
            numFunciones++;
        }...

    public void evaluarFunciones(float x){
        if (numFunciones > 0) {
            System.out.println("Evaluandolasfunciones...\n");
            for(inti=0;i<numFunciones;i++){
                System.out.println("Elvalordelafuncion"+funciones[i].getRepresentacion()+
                    " En el punto " + x + "es " + funciones[i].evalua(x));
            }
        }...

        public void listarFunciones() {
            if (numFunciones > 0) {
                System.out.println("Lasfuncionesalmacenadasson:");
                for(inti=0;i<numFunciones;i++){
                    System.out.println(funciones[i].getRepresentacion());
                }
            }...
        }
    }
}
```

En el listado 9 podemos observar como la interfaz Funcion y la clase abstracta y FuncionAbstracta son idénticas a las que ya hemos presentado anteriormente en este artículo, sólo que esta vez están definidas dentro del paquete funciones. La clase ContenedorDeFunciones contiene un array de objetos Funcion . Es posible añadir funciones al contenedor a través del método anhadirFuncion(Funcion f) . Podemos mostrar en la consola una representación textual de todas las funciones que contiene el contenedor empleando el método listarFunciones() . Observa que este método delega en cada una de las funciones del array para generar cada una de las cadenas de caracteres que representan las funciones. Finalmente, podemos evaluar todas las funciones del contenedor empleando el método evaluarFunciones(float x) . Nuevamente, este método delega en cada una de las funciones tanto para obtener una representación textual de ellas como para obtener el valor de la función en el punto que se especifica.

Lo bonito del código del paquete funciones es que no hace absolutamente ninguna suposición sobre qué tipo de funciones (polinomios, exponenciales, senos, etc.) está tratando. El paquete es capaz de gestionar cualquier función que implemente la interfaz Funcion . Y dicho paquete no depende de ningún recurso externo a él. Es decir, cualquier cambio que se produzca fuera del

paquete funciones no va a afectar en absoluto a ninguna de las tres clases que hemos presentado, ya que ninguna de estas clases depende de nada que quede fuera de su paquete.

Esto ha sido posible empleando de modo adecuado a la abstracción: sabemos que todas las funciones tienen una representación textual. Podemos no saber cuál es la representación textual concreta de una función determinada, pero es responsabilidad de la propia función proporcionarla bajo la forma de un String . Tampoco sabemos cómo evaluar cualquier función que se le pueda ocurrir a cualquier programador. Pero sabemos que a todas las funciones de una variable se les pasa un valor del eje x y nos devuelven a cambio el valor de la función en dicho punto. No sabemos los "cómo" de estas dos operaciones. Pero sí sabemos el "qué" tienen que hacer. Y este "qué" está capturado en la interfaz Funcion . Y tanto el código de la clase abstracta FuncionAbstracta como el de ContenedorDeFunciones se basan en lo que deben poder hacer las funciones (que está definido, lo repetiré una vez más, en la interfaz contenida en el paquete) pero no en cómo lo harán. El cómo es un detalle de implementación que no importa.

A continuación presentamos el contenido de un segundo paquete, al que denominaremos funciones.implementaciones . En ese paquete será donde nosotros coloquemos las funciones que ya hemos implementado en nuestro software. En dicho paquete, además de las clases FuncionLineal y FuncionCuadratica (que ya hemos presentado en el artículo) añadiremos una tercera clase, Exponencial , cuyo código mostramos en el listado 10. Esta clase representa una función exponencial de la forma $a * e^{(b * x)}$.

```
//LISTADO 10: Clase que representa una función exponencial
package funciones.implementaciones;
```

```
import funciones.FuncionAbstracta;
```

```
public class Exponencial extends FuncionAbstracta {
    float a, b;
```

```
    public Exponencial(float a, float b) {
        this.a = a;
        this.b = b;
    }
```

```
    public float evalua(float x) {
        return (float)(a * Math.exp(b * x));
    }
```

```
    public String getRepresentacion() {
        return a + "e^" + b + "x";
    }
}
```

Observa como en el código de la clase Exponencial hemos tenido que importar la clase FuncionAbstracta del paquete funciones. Observa también la sentencia package al principio del código declarando el paquete en el cual se encuentra esta clase. Por último, resaltar una vez más que el código del paquete funciones desconoce completamente las funciones que hemos implementado en este segundo paquete. Sin embargo, es capaz de manipularlas sin ningún problema porque implementa la interfaz Funciones.

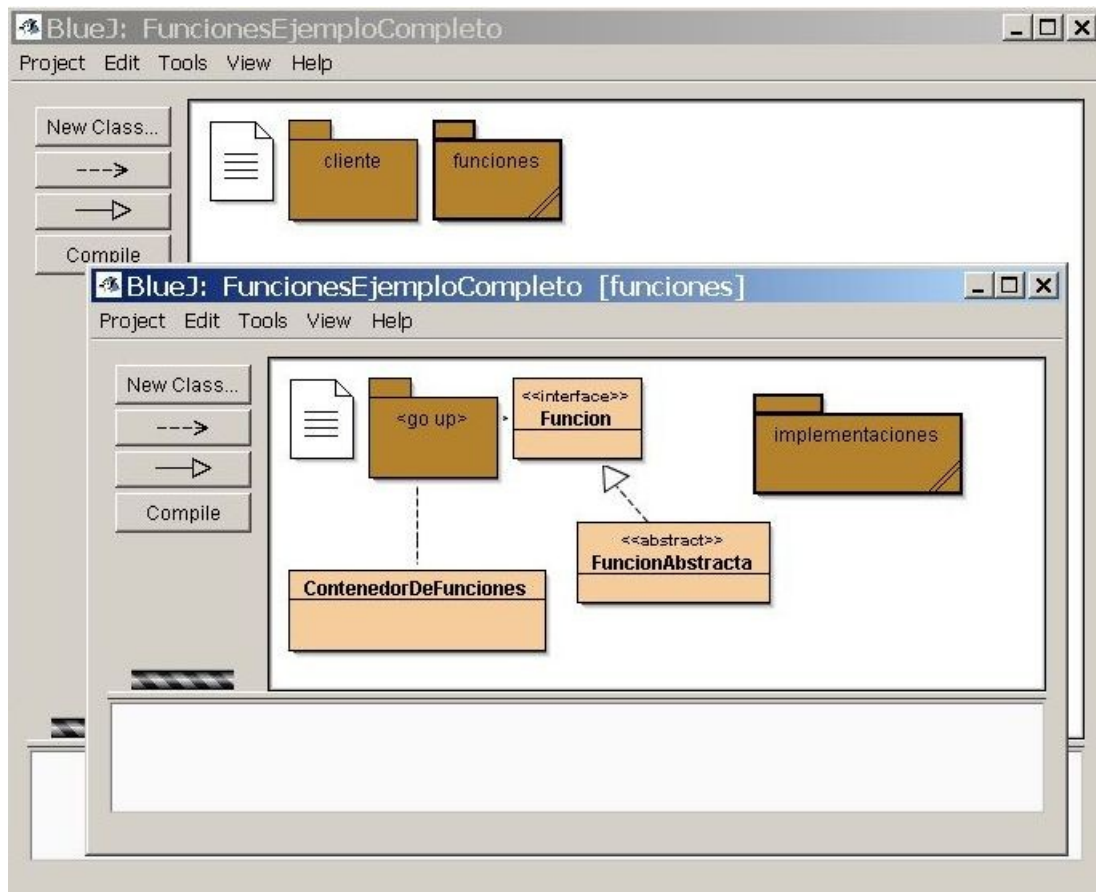


FIGURA 5: Representación que BlueJ realiza del proyecto, y del paquete funciones

Finalmente, vamos a crear un código cliente que use el código de ambos paquetes; el del paquete funciones para almacenar y evaluar un grupo de funciones, y el del paquete funciones.implementaciones para usar las funciones que ya están implementadas en él y para pasárselas al contenedor de funciones del anterior paquete. Este código, que se situará en el paquete cliente, puede verse en el listado 11.

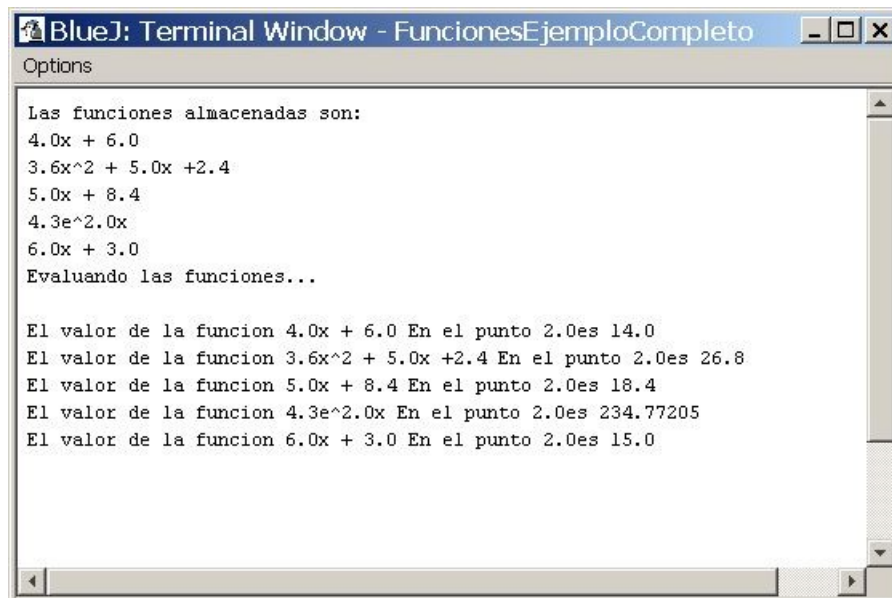
//LISTADO 11: Código cliente que hace uso del contenedor de funciones
package cliente;

```
import funciones.ContenedorDeFunciones;
import funciones.implementaciones.*;

public class Principal {
public static void main(String[] args){
    ContenedorDeFunciones e = new ContenedorDeFunciones();
    e.anadirFuncion(new FuncionLineal(4, 6));
    e.anadirFuncion(new FuncionCuadratica(3.6F, 5, 2.4F));
    e.anadirFuncion(new FuncionLineal(5, 8.4F));
    e.anadirFuncion(new Exponencial(4.3F, 2));
    e.anadirFuncion(new FuncionLineal(6, 3));
    e.listarFunciones();
    e.evaluarFunciones(2);
}
}
```

Observa como el código cliente tiene que importar la clase ContenedorDeFunciones del paquete funciones para poder acceder a su funcionalidad. También tiene que importar todo el contenido del paquete funciones.implementaciones para poder acceder a las funciones lineales, cuadráticas y exponenciales. Observa, sin embargo, que ni el paquete funciones ni el paquete funciones.implementaciones saben de la existencia de este tercer paquete. En la figura 6 puedes ver el resultado de ejecutar el método main de la clase Principal.

La flexibilidad del código de este ejemplo todavía va más allá. El paquete cliente puede definir funciones propias que implementen la interfaz Funcion , o extiendan la clase abstracta FuncionAbstracta . Estas funciones, siendo completamente ajenas al código que yo he implementado, serán manipuladas de modo correcto por él. Te dejo como deberes que tú crees alguna otra función matemática (por ejemplo, una que sea una combinación lineal de un seno y un coseno) y que modifiques el código de la clase Principal del paquete cliente para añadir al contenedor de funciones instancias de tu clase. No me hubiera costado más de dos minutos poner este código en el CD, pero la forma de comprobar que uno realmente entiende estas cosas es haciéndolas uno mismo. No obstante, si alguno de vosotros lo intenta y no lo consigue, que me escriba un correo y estaré encantado de echarle una mano.



```
Options
Las funciones almacenadas son:
4.0x + 6.0
3.6x^2 + 5.0x +2.4
5.0x + 8.4
4.3e^2.0x
6.0x + 3.0
Evaluando las funciones...

El valor de la funcion 4.0x + 6.0 En el punto 2.0es 14.0
El valor de la funcion 3.6x^2 + 5.0x +2.4 En el punto 2.0es 26.8
El valor de la funcion 5.0x + 8.4 En el punto 2.0es 18.4
El valor de la funcion 4.3e^2.0x En el punto 2.0es 234.77205
El valor de la funcion 6.0x + 3.0 En el punto 2.0es 15.0
```

FIGURA 6: Resultado de ejecutar el método main de la clase Principal

5 Conclusiones

Con este artículo terminamos de ver el grueso de la sintaxis de Java. Nos quedan un par de detalles más, que iremos presentando según nos vayan haciendo falta a lo largo del curso. Sin embargo, un lenguaje de programación es más que una sintaxis: también son un conjunto de librerías estándar en las cuales los programadores se apoyan para construir sus programas. En el próximo número comenzaremos a ver las partes más importantes de la librería estándar de Java. Os espero todos el mes que viene.

Descargas

- [Códigos del artículo](#)
- [Video del artículo \(.exe para Windows\)](#)
- [Video del artículo \(.vvl para Linux\)](#)
- [Video del artículo \(.hqx para Mac Os\)](#)
- [Video del artículo \(.swf, flash multiplataforma\)](#)

Cápítulos anteriores del curso:

- [Curso de programación Java I](#)
- [Curso de programación Java II](#)
- [Curso de programación Java III](#)