**PAQUETES EN JAVA**

Los paquetes se utilizan en Java de forma similar a como se utilizan las librerías en C++, para agrupar funciones y clases, solo que en Java agrupan diferentes clases y/o interfaces. En ellos las clases son únicas, comparadas con las de otros paquetes, y además proporcionan un método de control de acceso. Los paquetes también proporcionan una forma de *ocultar* clases, evitando que otros programas o paquetes accedan a clases que son de uso exclusivo de una aplicación determinada.

**Declaración de Paquetes**

Los paquetes se declaran utilizando la palabra **package** seguida del nombre del paquete. Esto debe estar al comienzo del fichero fuente, en concreto, debe ser la primera sentencia ejecutable del código Java, excluyendo, los comentarios y espacios en blanco. Por ejemplo:

package mamiferos;

class Ballena {

. . .

}

En este ejemplo, el nombre del paquete es **mamiferos**. La clase **Ballena** se considera como parte del paquete. La inclusión de nuevas clases en el paquete es muy sencilla, ya que basta con colocar la misma sentencia al comienzo de los ficheros que contengan la declaración de las clases. Como cada clase se debe colocar en un fichero separado, cada uno de los ficheros que contengan clases pertenecientes a un mismo paquete, deben incluir la misma sentencia package, y solamente puede haber una sentencia package por fichero.

Se recuerda que el compilador Java solamente requiere que se coloquen en ficheros separados las clases que se declaren públicas. Las clases no públicas se pueden colocar en el mismo fichero fuente, al igual que las clases anidadas. Aunque es una buena norma de programación que todas las clases se encuentren en un único fichero, la sentencia package colocada el comienzo de un fichero fuente afectará a todas las clases que se declaren en ese fichero.

Java también soporta el concepto de jerarquía de paquetes. Esto es parecido a la jerarquía de directorios de la mayoría de los sistemas operativos. Se consigue especificando múltiples nombres en la sentencia *package*, separados por puntos. Por ejemplo, en las sentencias siguientes, la clase Ballena pertenece al paquete mamiferos que cae dentro de la jerarquía del paquete animales.

package animales.mamiferos;

class Ballena {

. . .

}

Esto permite agrupar clases relacionadas en un solo paquete, y agrupar paquetes relacionados en un paquete más grande. Para referenciar a un miembro de otro paquete, se debe colocar el nombre del paquete antes del nombre de la clase. La siguiente sentencia es un ejemplo de llamada al método obtenerNombre() de la clase **Ballena** que pertenece al subpaquete mamiferos del paquete **animales**:

animales.mamiferos.Ballena.obtenerNombre();

La analogía con la jerarquía de directorios se ve reforzada por el intérprete Java, ya que este requiere que los ficheros **.class** se encuentren físicamente localizados en subdirectorios que coincidan con el nombre del **subpaquete**. En el ejemplo anterior, si se encontrase en una máquina Unix, la clase Ballena debería estar situada en el camino siguiente:

animales/mamiferos/Ballena.class

En Windows en:

C:\. . .\animales\mamiferos\Ballena.class

Por supuesto, las convenciones en el nombre de los directorios serán diferentes para los distintos sistemas operativos. El compilador Java colocará los ficheros .class en el mismo directorio que se encuentren los ficheros fuentes, por lo que puede ser necesario mover los ficheros .class resultantes de la compilación al directorio adecuado, en el caso de que no se encuentren los fuentes en el lugar correcto del árbol jerárquico. Aunque los ficheros .**class** también se pueden colocar directamente en el directorio que se desee especificando la opción -d (directorio) a la hora de invocar al compilador. La siguiente línea de comando colocará el fichero resultante de la compilación en el subdirectorio animales/mamiferos/Ballenas, independientemente de cual sea el directorio desde el cual se esté invocando al compilador.

> javac -d animales/mamiferos/Ballena Ballena.java

Todas las clases quedan englobadas dentro de un mismo paquete, si no se especifica explícitamente lo contrario, es decir, aunque no se indique nada, las clases pertenecen a un paquete; ya que, como es normal en Java, lo que no se declara explícitamente, toma valores por defecto. En este caso, hay un paquete sin nombre que agrupa a todos los demás paquetes. Si un paquete no tiene nombre, no es posible para los demás paquetes referenciar a ese paquete, por eso es conveniente colocar todas las clases no triviales en paquetes, para que puedan ser referenciadas posteriormente desde cualquier otro programa.

**Acceso a Otros Paquetes**

Como ya se ha visto se pueden referenciar paquetes precediendo con su nombre la clase que se quiere usar. También se puede emplear la palabra clave import, si se van a colocar múltiples referencias a un mismo paquete, o si el nombre del paquete es muy largo o complicado.

La sentencia **import** se utiliza para incluir una lista de paquetes en los que buscar una clase determinada, y su sintaxis es:

import nombre\_paquete;

Esta sentencia, o grupo de ellas, deben aparecer antes de cualquier declaración de clase en el código fuente. Por ejemplo:

import animales.mamiferos.Ballena;

En este ejemplo, todos los miembros (variables, métodos) de la clase Ballena están accesibles especificando simplemente su nombre, sin tener que precederlo del nombre completo del paquete.

Esta forma de abreviar tienes sus ventajas y sus desventajas. La ventaja principal es que el código no se vuelve demasiado difícil de leer y además es más rápido de teclear. La desventaja fundamental es que resulta más complicado el saber exactamente a qué paquete pertenece un determinado miembro; y esto es especialmente complicado cuando hay muchos paquetes importados.

En la sentencia import también se admite la presencia del carácter \*, asterisco. Cuando se emplea, se indica que toda la jerarquía de clases localizada a partir del punto en que se encuentre, debe ser importada, en lugar de indicar solamente una determinada clase. Por ejemplo, la siguiente sentencia indicaría que todas las clases del subpaquete *animales.mamiferos*, deben ser importadas:

import animales.mamiferos.\*;

Esta es una forma simple y sencilla de tener acceso a todas las clases de un determinado paquete. Aunque el uso del asterisco debe hacerse con cautela, porque al ya de por sí lento compilador, si se pone un asterisco, se cargarán todos los paquetes, lo que hará todavía más lenta la compilación. No obstante, el asterisco no tiene impacto alguno a la hora de la ejecución, solamente en tiempo de compilación.

La sentencia **import** se utiliza en casi todos los ejemplos del Tutorial, fundamentalmente para acceder a las distintas partes del API de Java. Por defecto, el conjunto de clases bajo java.lang.\* se importan siempre; las otras librerías deben ser importadas explícitamente. Por ejemplo, las siguientes líneas de código permiten el acceso a las clases correspondientes a las librerías de manipulación de imágenes y gráficos:

import java.awt.Image;

import java.awt.Graphics;

Otro ejemplo:

Y este ejemplo no es el peor vemos otro:

**java.util.[List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<**[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html)**> meses= new java.util.**[**ArrayList**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/ArrayList.html)**<**[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html)**>();**

En lugar de lo anterior lo que haremos es importar previamente la clase desde el paquete que la contiene y posteriormente utilizarla simplemente con el nombre por lo que si retomamos el ejemplo anterior quedaría de la siguiente manera:

**import** java.util.List;

**import** java.util.ArrayList;

[**List**](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html)> meses= **new** [**ArrayList**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/ArrayList.html)<[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html)>();

Como vemos esto nos facilita más las cosas, podemos hacerlo todavía más fácil, en el ejemplo anterior notamos que tanto List como ArrayList se encuentran en el mismo paquete por lo cual podemos importar todas las clases del paquete y nos quedaría de la siguiente forma:

**import** java.util.\*;

Con lo cual le indicamos a java que vamos a utilizar una o más clases del paquete java.util.

**Nomenclatura de Paquetes**

Los paquetes pueden nombrarse de cualquier forma que siga el esquema de nomenclatura de Java. Por convenio, no obstante, los nombres de paquetes comienzan por una letra minúscula para hacer más sencillo el reconocimiento de paquetes y clases, cuando se tiene una referencia explícita a una clase. Esto es porque los nombres de las clases, también por convenio, empiezan con una letra mayúscula. Por ejemplo, cuando se usa el convenio citado, es obvio que tanto **animales** como **mamiferos** son paquetes y que **Ballena** es una clase. Cualquier cosa que siga al nombre de la clase es un miembro de esa clase:

animales.mamiferos.Ballena.obtenerNombre();

Java sigue este convenio en todo el API. Por ejemplo, el método System.out.println() que tanto se ha utilizado sigue esta nomenclatura. El nombre del paquete no se declara explícitamente porque java.lang.\* siempre es importado implícitamente. **System** es el nombre de la clase perteneciente al paquete java.lang y está capitalizado. El nombre completo del método es:

java.lang.System.out.println();

Cada nombre de paquete ha de ser único, para que el uso de paquetes sea realmente efectivo. Los conflictos de nombres pueden causar problemas a la hora de la ejecución en caso de duplicidad, ya que los ficheros de clases podrían saltar de uno a otro directorio. En caso de proyectos pequeños no es difícil mantener una unicidad de nombres, pero en caso de grandes proyectos; o se sigue una norma desde el comienzo del proyecto, o este se convertirá en un auténtico caos.

No hay ninguna organización en Internet que controle esta nomenclatura, y muchas de las aplicaciones Java corren sobre Web. Hay que tener presente que muchos servidores Web incluyen applets de múltiples orígenes, con lo cual parece poco menos que imposible el evitar que alguien duplique nombres.

Una convención admitida para asegurar que los nombres de los paquetes sean únicos, es basarse en los nombres de los dominios, colocándolos al revés. Es decir, un dominio del tipo miempresa.com, debería colocar delante de todos sus paquetes el prefijo com.miempresa. Esto resolvería el problema de la nomenclatura, ya que los desarrolladores podrían controlar sus propios paquetes y, además, se generaría una estructura jerárquica de paquetes muy limpia. De hecho, el paquete Swing en la versión beta 3 del JDK 1.2 se situó bajo el árbol java.awt, lo cual sugería que las clases Swing dependían del AWT, cuando es un paquete autosuficiente y que no tiene mucho que ver con el AWT, así que Javasoft (Sun) dio marcha atrás en su nomenclatura y colocó el paquete en su situación actual com.java.Swing.

Como norma y resumen de todo lo dicho, a la hora de crear un paquete hay que tener presente una serie de ideas:

* La palabra clave package debe ser la primera sentencia que aparezca en el fichero, exceptuando, claro está, los espacios en blanco y comentarios
* Es aconsejable que todas las clases que vayan a ser incluidas en el paquete se encuentren en el mismo directorio. Como se ha visto, esta recomendación se la puede uno, pero se corre el riesgo de que aparezcan determinados problemas difíciles de resolver a la hora de compilar, en el supuesto caso de que no se hile muy fino
* Ante todo, recordar que en un fichero únicamente puede existir, como máximo, una clase con el especificador de acceso public, debiendo coincidir el nombre del fichero con el nombre de la clase.

**Variable de Entorno CLASSPATH**

El intérprete Java debe encontrar todas las clases referenciadas cuando se ejecuta una aplicación Java. Por defecto, Java busca en el árbol de instalación de Java esas librerías. En el *Tutorial de Java*, se indica que "*los ficheros .class del paquete java.util están en un directorio llamado util de un directorio java, situado en algún lugar apuntado por CLASSPATH*".

CLASSPATH es una variable de entorno que indica al sistema dónde debe buscar los ficheros **.class** que necesite. Sin embargo, lo que dice el *Tutorial de Java*, normalmente no es así, lo cual puede ocasionar confusión. Cuando se utiliza el JDK, no existe el directorio que se indica.

La no existencia se debe a que Java tiene la capacidad de buscar ficheros comprimidos que utilicen la tecnología **zip**. Esto redunda en un gran ahorro de espacio en disco y además, mantiene la estructura de directorios en el fichero comprimido. Por tanto, se podría parafrasear lo indicado por Sun escribiendo que "*en algún lugar del disco, se encontrará un fichero comprimido (zip) que contiene una gran cantidad de ficheros .class. Antes de haber sido comprimidos, los ficheros .class del paquete java.util estaban situados en un directorio llamado util de un directorio java. Estos ficheros, junto con sus estructura se almacenar en el fichero comprimido que debe encontrarse en algún lugar apuntado por CLASSPATH*".

CLASSPATH contiene la lista de directorios en los que se debe buscar los árboles jerárquicos de librerías de clases. La sintaxis de esta variable de entorno varía dependiendo del sistema operativo que se esté utilizando; en sistemas Unix, contiene una lista de directorios separados por: (dos puntos), mientras que en sistemas Windows, la lista de directorios está separada por ; (punto y coma). La sentencia siguiente muestra un ejemplo de esta variable en un sistema Unix:

CLASSPATH=/home/users/afq/java/classes:/opt/apps/Java

Indicando al intérprete Java que busque en los directorios:

/home/users/afq/java/classes y /opt/apps/Java las librerías de clases.

**Paquetes de Java**

El lenguaje Java proporciona una serie de paquetes que incluyen ventanas, utilidades, un sistema de entrada/salida general, herramientas y comunicaciones. En la versión actual del JDK, algunos de los paquetes Java que se incluyen son los que se muestran a continuación, que no es una lista exhaustiva, sino para que el lector pueda tener una idea aproximada de lo que contienen los paquetes más importantes que proporciona el JDK. Posteriormente, en el desarrollo de otros apartados del Tutorial, se introducirán otros paquetes que también forman parte del JDK y que, incorporan características a Java que hacen de él un lenguaje mucho más potente y versátil, como son los paquetes Java2D o Swing.

***java.applet***

Este paquete contiene clases diseñadas para usar con applets. Hay la clase Applet y tres interfaces: AppletContext, AppletStub y AudioClip.

***java.awt***

El paquete Abstract Windowing Toolkit (awt) contiene clases para generar widgets y componentes GUI (Interfaz Gráfico de Usuario), de manipulación de imágenes, impresión, fuentes de caracteres, cursores, etc.. Incluye las clases Button, Checkbox, Choice, Component, Graphics, Menu, Panel, TextArea, TextField...

***java.io***

El paquete de entrada/salida contiene las clases de acceso a ficheros, de filtrado de información, serialización de objetos, etc.: FileInputStream, FileOutputStream, FileReader, FileWriter. También contiene los interfaces que facilitan la utilización de las clases: DataInput, DataOutput, Externalizable, FileFilter, FilenameFilter, ObjectInput, ObjectOutput, Serializable...

***java.lang***

Este paquete incluye las clases del lenguaje Java propiamente dicho: Object, Thread, Exception, System, Integer, Float, Math, String, Package, Process, Runtime, etc.

***java.net***

Este paquete da soporte a las conexiones del protocolo TCP/IP y, además, incluye las clases Socket, URL y URLConnection.

***java.sql***

Este paquete incluye todos los interfaces que dan acceso a Bases de Datos a través de JDBC, Java DataBase Connectivity, como son: Array, Blob, Connection, Driver, Ref, ResultSet, SQLData, SQLInput, SQLOutput, Statement, Struct; y algunas clases específicas: Date, DriveManager, Time, Types...

***java.util***

Este paquete es una miscelánea de clases útiles para muchas cosas en programación: estructuras de datos, fechas, horas, internacionalización,etc. Se incluyen, entre otras, Date (fecha), Dictionary (diccionario), List (lista), Map (mapa), Random (números aleatorios) y Stack (pila FIFO). Dentro de este paquete, hay tres paquetes muy interesantes: java.util.jar, que proporciona clases para leer y crear ficheros JAR; java.util.mime, que proporciona clases para manipular tipos MIME, Multipurpose Internet Mail Extension (RFC 2045, RFC 2046) y java.util.zip, que proporciona clases para comprimir, descomprimir, calcular checksums de datos, etc. con los formatos estándar ZIP y GZIP.

**Como importar clases entre distintos proyectos en Eclipse**

Importar clases dentro de un mismo proyecto es trivial, sólo tenemos que hacer un **import elPaquete.laClase**, y ya podremos crear objetos de las clases que se encuentran en el paquete que acabamos de importar.

Sin embargo, cuando intentamos importar entre distintos proyectos, veremos que no podemos hacerlo con este método, de hecho, no es trivial el cómo hacerlo.

Se puede hacer de dos formas, la primera:

#### Añadir a las fuentes de un proyecto, las fuentes de otro

1. Pinchamos sobre el proyecto al que queremos importar paquetes de otros proyectos, y con click-derecho accedemos a las propiedades (Properties) y ahí accedemos a “Java Build Path”.
2. En esta ventana accedemos a la pestaña “Source” y pinchamos en “Link Source”.
3. Aquí podemos añadiremos el directorio de fuentes de otro proyecto que queremos importar. En “Linked Folder Location” ponemos la ruta de la carpeta src del otro paquete. En “Folder Name” nos pondrá src por defecto, pero este nombre ya existirá asi que lo cambiamos por el nombre que queramos.
4. Ya está hecho, ahora nos saldrá dentro del proyecto otra carpeta que apunta a la carpeta que indicamos y podremos importar dicho paquete para crear clases de este.

Otra solución que se suele utilizar más en grandes proyectos.

#### Añadir a las librerías de un proyecto, bytecode de otro

La solución es generar un fichero .jar con el bytecode de nuestro proyecto, con la evidente pega de tener que recompilar y crear el jar si cambiamos algo en el proyecto que estamos intentando utilizar en otro.

Lo podemos hacer con los siguientes pasos:

1. Para empezar tendremos que compilar el proyecto que quieres usar en otro u otros.
2. Ahora nos dirigimos al directorio bin y añadimos todo el contenido en un fichero .zip
3. Renombramos el .zip a .jar
4. Solo queda incluir la librería .jar como uno de los jars de tu proyecto.

También se podría simplemente añadir la carpeta donde tenemos las clases compiladas, sin comprimir, aunque en proyectos grandes es preferible comprimir.