Clases y métodos en java

1. Jerarquía de clases de Java (API)

Durante la generación de código en ***Java***, es recomendable y casi necesario tener siempre a la vista la documentación ***on-line*** del ***API*** de ***Java***. En dicha documentación es posible ver tanto la jerarquía de clases, es decir la relación de herencia entre clases, como la información de los distintos ***packages*** que componen las librerías base de ***Java***.

Es importante distinguir entre lo que significa ***herencia*** y ***package***. Un ***package*** es una agrupación arbitraria de clases, una forma de organizar las clases. La ***herencia*** sin embargo consiste en crear nuevas clases en base a otras ya existentes. Las clases incluidas en un *package* ***no derivan*** por lo general de una única clase.

En la documentación ***on-line*** se presentan ambas visiones: "***Package Index***" y "***Class Hierarchy***", en las diferentes versiones de Java, con pequeñas variantes. La primera presenta la estructura del ***API*** de ***Java*** agrupada por ***packages***, mientras que en la segunda aparece la jerarquía de clases. Hay que resaltar el hecho de que todas las clases en ***Java*** son derivadas de la clase ***java.lang.Object***, por lo que heredan todos los métodos y variables de ésta.

Si se selecciona una clase en particular, la documentación muestra una descripción detallada de todos los métodos y variables de la clase. A su vez muestra su herencia completa (partiendo de la clase ***java.lang.Object***).

Algunas de las clases ya definidas y utilizables en Java son:

Éstas vienen en las bibliotecas estándar:

- **java.lang.-** clases esenciales, números, strings, objetos, compilador, runtime, seguridad y threads (es el único paquete que se incluye automáticamente en todo programa Java)
- **java.io.-** clases que manejan entradas y salidas.
- **java.util.-** clases útiles, como estructuras genéricas, manejo de fecha, hora y strings, númeroaleatorios, etc.
- **java.net.-** clases para soportar redes: URL, TCP, UDP, IP, etc.
- **java.awt.-** clases para manejo de interface gráfica, ventanas, etc.
- **java.awt.image.-** clases para manejo de imágenes
- **java.awt.peer.-** clases que conectan la interface gráfica a implementaciones dependientes de la plataforma (motif, windows)
- **java.applet.-** clases para la creación de applets y recursos para reproducción de audio.

Para darnos una idea, los números enteros, por ejemplo, son "instancias" de una clase no redefinible, *Integer*, que desciende de la clase *Number,*

```
java.lang.Object
└ java.lang.Number
   └ java.lang.Integer
```

http://java.sun.com/reference/api/

esta clase implementa los siguientes atributos y métodos:

```
public final class java.lang.Integer extends java.lang.Number {
	// Atributos
	public final static int MAX_VALUE;
	public final static int MIN_VALUE;
	// Métodos Constructores
	public Integer(int value);
	public Integer(String s);
	// Más Métodos
	public double doubleValue();
	public boolean equals(Object obj);
	public float floatValue();
	public static Integer getInteger(String nm);
	public static Integer getInteger(String nm, int val);
	public static Integer getInteger(String nm, Integer val);
	public int hashCode();
	public int intValue();
	public long longValue();
	public static int parseInt(String s);
	public static int parseInt(String s, int radix);
	public static String toBinaryString(int i);
	public static String toHexString(int i);
	public static String toOctalString(int i);
	public String toString();
	public static String toString(int i);
	public static String toString(int i, int radix);
	public static Integer valueOf(String s);
	public static Integer valueOf(String s, int radix);
}
```

Puntos que debemos considerar:

- **La estructura de una clase.**
- **Métodos repetidos.** (como *parseInt* por ejemplo), al llamarse al método el compilador decide cuál de las implementaciones del mismo usar basándose en la cantidad y tipo de parámetros que le pasamos. Por ejemplo, parseInt("134") y parseInt("134",16), al compilarse, generarán llamados a dos métodos distintos.

2. Conceptos básicos

2.1.Concepto de clase

Una clase es una agrupación de ***datos*** (variables o campos) y de ***funciones*** (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

```
[public] class Classname {
// definición de variables y métodos
...
}
```

donde la palabra ***public*** es opcional: si no se pone, la clase tiene la visibilidad por defecto, esto es, sólo es visible para las demás clases del ***package***. Todos los métodos y variables deben ser definidos dentro del ***bloque*** {...} de la clase.

Un ***objeto*** (en inglés, ***instance***) es un ejemplar concreto de una clase. Las ***clases*** son como tipos de variables, mientras que los ***objetos*** son como variables concretas de un tipo determinado.

NombreDeClase objeto1;
NombreDeClase objeto2;

A continuación se enumeran algunas características importantes de las clases:

1. En Java no existen variables o funciones globales, ya que, todas las variables y funciones deben pertenecer a una clase.
2. Una clase que deriva de otra ***(extends***), hereda todas sus variables y métodos.
3. Una clase sólo puede heredar de una única clase (en ***Java*** no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de **java.lang.*Object***, que es la base de toda la jerarquía de clases de ***Java***.
4. En un archivo es posible definir varias clases, pero en únicamente puede existir una clase ***public***, el archivo debe de llamarse como la clase ***public*** con extensión ****.java***. En caso de existir, una clase en un archivo que no es ***public***, no es necesario que compartan el mismo nombre.
5. Los métodos de una clase pueden referirse de modo global al ***objeto*** de esa clase al que se aplican por medio de la referencia ***this***.
6. Las clases se pueden agrupar en paquetes (***packages)*** que está relacionada con la jerarquía de directorios y archivos en la que se guardan las clases. Ejemplo:
   package packageName

2.2.Concepto de interface

Una ***interface*** es un conjunto de declaraciones de funciones. Si una clase implementa (***implements***) una ***interface***, debe definir ***todas*** las funciones especificadas por la ***interface***. Las interfaces pueden definir también ***variables finales*** (constantes). Una ***clase*** puede implementar más de una ***interface***, representando una alternativa a la herencia múltiple.

En algunos aspectos los nombres de las ***interfaces*** pueden utilizarse en lugar de las ***clases***. Por ejemplo, las ***interfaces*** sirven para definir ***referencias*** a cualquier objeto de cualquiera de las clases que implementan esa ***interface***. Con ese nombre o referencia, sin embargo, sólo se pueden utilizar los métodos de la interface. Éste es un aspecto importante del ***polimorfismo***.

Una ***interface*** puede derivar de otra o incluso de varias ***interfaces***, en cuyo caso incorpora las declaraciones de todos los métodos de las ***interfaces*** de las que deriva (a diferencia de las clases, las interfaces de ***Java*** sí tienen herencia múltiple).

3. Definición de una clase

A continuación se reproduce como ejemplo la clase ***Circulo:***

```
//Circulo.java

public class Circulo extends Geometria {
static int numCirculos = 0;
public static final double PI=3.14159265358979323846;
public double x, y, r;
public Circulo(double x, double y, double r) {
      this.x=x;
      this.y=y;
      this.r=r;
```

```
        numCirculos++;
}
public Circulo(double r) {
        this(0.0, 0.0, r);
}
public Circulo(Circulo c) {
        this(c.x, c.y, c.r);
}
public Circulo() {
        this(0.0, 0.0, 1.0);
}
public double perimetro() {
        return 2.0 * PI * r;
}
public double area() {
        return PI * r * r;
}

// método de objeto para comparar círculos
public Circulo elMayor(Circulo c) {
        if (this.r>=c.r)
                return this;
        else
                return c;
}
// método de clase para comparar círculos
public static Circulo elMayor(Circulo c, Circulo d) {
        if (c.r>=d.r)
                return c;
        else
                return d;
}
} // fin de la clase Circulo
```

En este ejemplo se ve cómo se definen las variables miembro y los métodos dentro de la clase. Dichas variables y métodos pueden ser ***de objeto*** o ***de clase*** (***static***). Se puede ver también cómo el nombre del archivo coincide con el de la clase ***public*** con la extensión ****.java***.

4. Variables miembro

A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está ***centrada en los datos***. Una clase está constituida por unos ***datos*** y unos ***métodos*** que operan sobre esos datos.

4.1.Variables miembro de objeto

Cada objeto, es decir cada ejemplar concreto de la clase, tiene su propia copia de las variables miembro.

Las variables miembro de una clase (también llamadas ***campos***) pueden ser de ***tipos primitivos*** (*boolean*, *int*, *long*, *double*, …) o referencias a ***objetos*** de otra clase (*composición*).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de ***tipos primitivos*** se inicializan siempre de modo automático, incluso antes de llamar al ***constructor*** (***false*** para ***boolean***, el carácter nulo

para ***char*** y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas también en el constructor.

Las variables miembro pueden también inicializarse explícitamente en la ***declaración***, como las variables locales, por medio de constantes o llamadas a métodos (esta inicialización no está permitida en C++). Por ejemplo,

long nDatos = 100;

Las variables miembro se inicializan en el mismo orden en que aparecen en el código de la clase.

Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada ***objeto*** que se crea de una clase tiene ***su propia copia*** de las variables miembro. Por ejemplo, cada objeto de la clase ***Circulo*** tiene sus propias coordenadas del centro ***x*** e ***y***, y su propio valor del radio ***r***.

Los ***métodos de objeto*** se aplican a un objeto concreto poniendo el nombre del objeto y luego el nombre del método, separados por un punto. A este objeto se le llama ***argumento implícito***. Por ejemplo, para calcular el área de un objeto de la clase ***Circulo*** llamado ***c1*** se escribirá: ***c1.area();***. Las variables miembro del argumento implícito se acceden directamente o precedidas por la palabra ***this*** y el operador punto.

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: ***public***, ***private***, ***protected*** y ***package*** (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (***public*** y ***package***), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro.

Existen otros dos modificadores (no de acceso) para las variables miembro:

1. ***transient***: indica que esta variable miembro no forma parte de la ***persistencia*** (capacidad de los objetos de mantener su valor cuando termina la ejecución de un programa) de un objeto y por tanto no debe ser ***serializada*** (convertida en flujo de caracteres para poder ser almacenada en disco o en una base de datos) con el resto del objeto.
2. ***volatile***: indica que esta variable puede ser utilizada por distintas ***threads*** sincronizadas y que el compilador no debe realizar optimizaciones con esta variable.

Al nivel de estos apuntes, los modificadores ***transient*** y ***volatile*** no serán utilizados

4.2.Variables miembro de clase (static)

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama ***variables de clase*** o variables ***static***. Las variables ***static*** se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo ***PI*** en la clase ***Circulo***) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como ***numCirculos*** en la clase ***Circulo***).

Las variables de clase son lo más parecido que ***Java*** tiene a las ***variables globales*** de C/C++.

Las variables de clase se crean anteponiendo la palabra ***static*** a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar

también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, ***Circulo.numCirculos*** es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembro ***static*** se inicializan con los valores por defecto para los tipos primitivos (***false*** para ***boolean***, el carácter nulo para ***char*** y cero para los tipos numéricos), y con ***null*** si es una referencia.

Las variables miembro ***static*** se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método ***static*** o en cuanto se utiliza una variable ***static*** de dicha clase. Lo importante es que las variables miembro ***static*** se inicializan siempre antes que cualquier objeto de la clase.

4.3.Variables Finales

Una variable de un tipo primitivo declarada como ***final*** no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como una ***constante***, y equivale a la palabra ***const*** de C/C++.

Java permite separar la ***definición*** de la ***inicialización*** de una variable ***final***. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos. La variable ***final*** así definida es ***constante*** (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados ***final***.

Declarar como ***final*** un objeto miembro de una clase hace ***constante*** la ***referencia***, pero no el propio objeto, que puede ser modificado a través de otra referencia. En ***Java*** no es posible hacer que un objeto sea constante.

5. Métodos (funciones miembro)

5.1.Métodos de objeto

Los ***métodos*** son funciones definidas dentro de una clase. Salvo los métodos ***static*** o de clase, se aplican siempre a un objeto de la clase por medio del ***operador punto*** (.). Dicho objeto es su ***argumento implícito***. Los métodos pueden además tener otros ***argumentos explícitos*** que van entre paréntesis, a continuación del nombre del método.

La primera línea de la definición de un método se llama ***declaración*** o ***header***; el código comprendido entre las ***llaves*** {…} es el ***cuerpo*** o ***body*** del método. Considérese el siguiente método tomado de la clase ***Circulo***:

```
public Circulo elMayor(Circulo c) { // header y comienzo del método
       if (this.r>=c.r) // body
              return this; // body
       else // body
       return c; // body
} // final del método
```

El ***header*** consta del cualificador de acceso (***public***, en este caso), del tipo del valor de retorno (***Circulo*** en este ejemplo, ***void*** si no tiene), del ***nombre de la función*** y de una lista de ***argumentos explícitos*** entre paréntesis, separados por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Los métodos tienen ***visibilidad directa*** de las variables miembro del objeto que es su ***argumento implícito***, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia ***this***, de modo discrecional (como en el ejemplo anterior con ***this.r***) o si alguna variable local o argumento las oculta.

El ***valor de retorno*** puede ser un valor de un ***tipo primitivo*** o una ***referencia***. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de ***interface***. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una subclase, pero nunca de una super-clase.

Los métodos pueden definir ***variables locales***. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

Si en el ***header*** del método se incluye la palabra ***native*** (Ej: `public native void miMetodo();`) no hay que incluir el código o implementación del método. Este código deberá estar en una librería dinámica (*Dynamic Link Library* o DLL). Estas librerías son archivos de funciones compiladas normalmente en lenguajes distintos de ***Java*** (C, C++, Fortran, etc.). Es la forma de poder utilizar conjuntamente funciones realizadas en otros lenguajes desde código escrito en ***Java***. Este tema queda fuera del carácter fundamentalmente introductorio de este manual.

Un método también puede declararse como ***synchronized*** (Ej: `public synchronized doublé miMetodoSynch(){...}`). Estos métodos tienen la particularidad de que sobre un objeto no pueden ejecutarse simultáneamente dos métodos que estén sincronizados.

5.2.Métodos sobrecargados (overloading)

Al igual que C++, ***Java*** permite métodos ***sobrecargados*** (***overloaded***), es decir, métodos distintos que tienen ***el mismo nombre***, pero que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase ***Circulo*** del Apartado 3.2 presenta dos casos de métodos sobrecargados: los cuatro constructores y los dos métodos llamados ***elMayor()***.

A la hora de llamar a un método sobrecargado, ***Java*** sigue unas reglas para determinar el método concreto que debe llamar:

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.
3. Si sólo existen métodos con argumentos de un tipo más restringido (por ejemplo, *int* en vez de *long*), el programador debe hacer un ***cast*** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear

dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la ***sobrecarga*** de métodos es la ***redefinición***. Una clase puede ***redefinir*** (***override***) un método heredado de una superclase. ***Redefinir*** un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la ***herencia***.

5.3.Paso de parámetros y retorno de variables

En ***Java*** los argumentos de los ***tipos primitivos*** se pasan siempre ***por valor***. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro en una clase y pasar como argumento una referencia a un objeto de dicha clase. Las ***referencias*** se pasan también ***por valor***, pero a través de ellas se pueden modificar los objetos referenciados.

En ***Java*** no se pueden pasar métodos como argumentos a otros métodos (en C/C++ se pueden pasar punteros a función como argumentos). Lo que se puede hacer en ***Java*** es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Dentro de un método se pueden crear ***variables locales*** de los tipos primitivos o referencias. Estas variables locales dejan de existir al terminar la ejecución del método1. Los argumentos formales de un método (las variables que aparecen en el ***header*** del método para recibir el valor de los argumentos actuales) tienen categoría de variables locales del método.

Si un método devuelve ***this*** (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.), por ejemplo,

```
String numeroComoString = ”8.978”;
float p = Float.valueOf(numeroComoString).floatValue();
```

donde el método ***valueOf(String)*** de la clase ***java.lang.Float*** devuelve un objeto de la clase ***Float*** sobre el que se aplica el método ***floatValue()***, que finalmente devuelve una variable primitiva de tipo ***float***. El ejemplo anterior se podía desdoblar en las siguientes sentencias:

```
String numeroComoString = ”8.978”;
Float f = Float.valueOf(numeroComoString);
float p = f.floatValue();
```

Obsérvese que se pueden encadenar varias llamadas a métodos por medio del operador punto (.) que, como todos los operadores de ***Java*** excepto los de asignación, se ejecuta de izquierda a derecha.

5.4.Métodos de clase (static)

Análogamente, puede también haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama ***métodos de clase*** o ***static***. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia ***this***. Un ejemplo típico de métodos ***static*** son los métodos matemáticos de la clase ***java.lang.Math*** (*sin()*, *cos()*, *exp()*, *pow()*, etc.).

De ordinario el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

Los métodos y variables de clase se crean anteponiendo la palabra ***static***. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, ***Math.sin(ang)***, para calcular el seno de un ángulo).

Los métodos y las variables de clase son lo más parecido que ***Java*** tiene a las funciones y variables globales de C/C++ o Visual Basic.

5.5.Construcción de clases

5.5.1. Constructores

Un punto clave de la *Programación Orientada Objetos* es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. ***Java*** no permite que haya variables miembro que no estén inicializadas2. Ya se ha dicho que ***Java*** inicializa siempre con ***valores por defecto*** las variables miembro de clases y objetos. El segundo paso en la inicialización correcta de objetos es el uso de ***constructores***.

Un ***constructor*** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del ***constructor*** es reservar memoria e inicializar las variables miembro de la clase. Los ***constructores*** no tienen valor de retorno (ni siquiera ***void***) y su ***nombre*** es el mismo que el de la clase. Su ***argumento implícito*** es el objeto que se está creando.

De ordinario una clase tiene ***varios constructores***, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos ***sobrecargados***). Se llama ***constructor por defecto*** al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un ***constructor*** de una clase puede llamar a ***otro constructor previamente definido*** en la misma clase por medio de la palabra ***this***. En este contexto, la palabra ***this*** sólo puede aparecer en la ***primera sentencia*** de un ***constructor***.

El ***constructor*** de una ***sub-clase*** puede llamar al constructor de su ***super-clase*** por medio de la palabra ***super***, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor sólo tiene que inicializar por sí mismo las variables no heredadas.

El ***constructor*** es tan importante que, si el programador no prepara ***ningún constructor*** para una clase, el ***compilador*** crea un ***constructor por defecto***, inicializando las variables de los

tipos primitivos a su valor por defecto, y los ***Strings*** y las demás ***referencias*** a objetos a ***null***. Si hace falta, se llama al ***constructor*** de la ***super-clase*** para que inicialice las variables heredadas.

Al igual que los demás métodos de una clase, los ***constructores*** pueden tener también los modificadores de acceso ***public***, ***private***, ***protected*** y ***package***. Si un ***constructor*** es ***private***, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos ***public*** y ***static*** (*factory methods*) que llamen al ***constructor*** y devuelvan un objeto de esa clase.

Dentro de una clase, los ***constructores*** sólo pueden ser llamados por otros ***constructores*** o por métodos ***static***. No pueden ser llamados por los ***métodos de objeto*** de la clase.

6. Paquetes (package)

¿Qué es un package?

Un ***package*** es una agrupación de clases. En la API de ***Java 1.1*** había 22 ***packages***; en ***Java 1.2*** hay 59 ***packages***, lo que da una idea del "crecimiento" experimentado por el lenguaje.

Además, el usuario puede crear sus propios ***packages***. Para que una clase pase a formar parte de un ***package*** llamado ***pkgName***, hay que introducir en ella la sentencia:

```
package pkgName;
```

que debe ser la primera sentencia del archivo sin contar comentarios y líneas en blanco.

Los nombres de los ***packages*** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un ***package*** puede constar de varios nombres unidos por puntos (los propios ***packages*** de ***Java*** siguen esta norma, como por ejemplo ***java.awt.event***).

Todas las clases que forman parte de un ***package*** deben estar en el mismo directorio. Los nombres compuestos de los ***packages*** están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los ***nombres de las clases*** de ***Java*** sean únicos en ***Internet***. Es el nombre del ***package*** lo que permite obtener esta característica. Una forma de conseguirlo es incluir el ***nombre del dominio*** (quitando quizás el país), como por ejemplo en el ***package*** siguiente:

```
es.ceit.jgjalon.infor2.ordenar
```

Las clases de un ***package*** se almacenan en un directorio con el mismo nombre largo (***path***) que el ***package***. Por ejemplo, la clase,

```
es.ceit.jgjalon.infor2.ordenar.QuickSort.class
```

debería estar en el directorio,

```
CLASSPATH\es\ceit\jgjalon\infor2\ordenar\QuickSort.class
```

donde CLASSPATH es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de ***Java*** (clases del sistema o de usuario), en este caso la posición del directorio ***es*** en los discos locales del ordenador.

Los ***packages*** se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.
2. Para evitar conflictos de nombres (se recuerda que el dominio de nombres de ***Java*** es la ***Internet***). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del ***package***.
3. Para ayudar en el control de la accesibilidad de clases y miembros.

Cómo funcionan los packages

Con la sentencia ***import packname;*** se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre nombres de clases, ***Java*** da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del ***package***.

El importar un ***package*** no hace que se carguen todas las clases del ***package***: sólo se cargarán las clases ***public*** que se vayan a utilizar. Al importar un ***package*** no se importan los ***sub-packages***. Éstos deben ser importados explícitamente, pues en realidad son ***packages*** distintos. Por ejemplo, al importar ***java.awt*** no se importa ***java.awt.event***.

Es posible guardar en jerarquías de directorios diferentes los archivos ****.class*** y ****.java***, con objeto por ejemplo de no mostrar la situación del código fuente. Los ***packages*** hacen referencia a los archivos compilados ****.class***.

En un programa de ***Java***, una clase puede ser referida con su nombre completo (el nombre del ***package*** más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia ***import*** permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del ***package*** importado. Se importan por defecto el ***package java.lang*** y el ***package*** actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar ***import***: para ***una clase*** y para ***todo un package***:

```
import es.ceit.jgjalon.infor2.ordenar.QuickSort.class;
import es.ceit.jgjalon.infor2.ordenar.*;
```

que deberían estar en el directorio:

```
classpath\es\ceit\jgjalon\infor2\ordenar
```

El cómo afectan los ***packages*** a los permisos de acceso de una clase se estudiara con más detalle en las siguientes secciones.