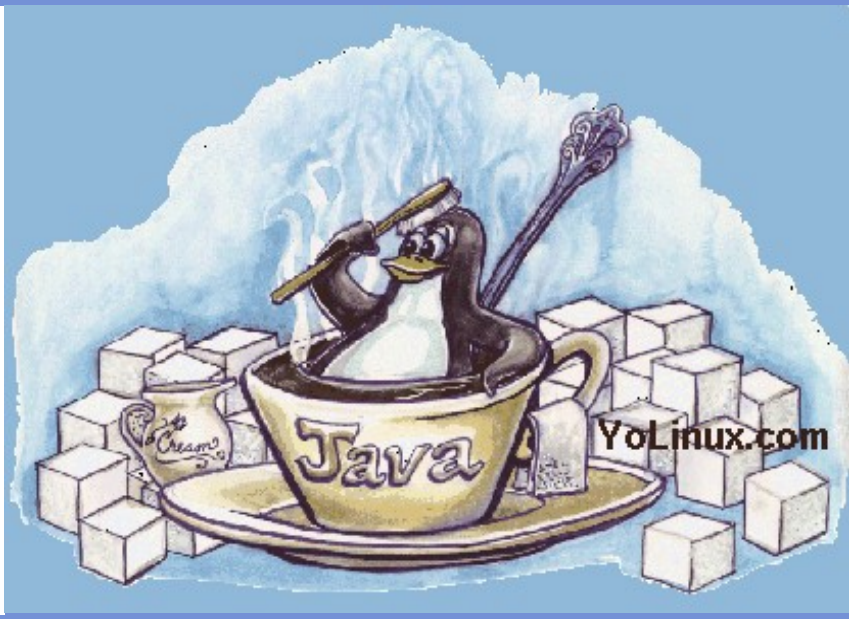
Fundación Código Libre Dominicano

Lic. Henry Terrero.
hterrero@codigolibre.org

Ing. Jose Paredes.
jparedes@codigolibre.org



Desarrollo De aplicaciones Con Java

Fundación Código Libre Dominicano
www.codigolibre.org

TEMARIO CURSO DE JAVA.

Modulo I.

1. **Conceptos Básicos.**
    - Breve historia de Java
    - El compilador de Java
    - La Java Virtual Machine
    - Las variables PATH y CLASSPATH

2. **Características del Lenguaje.**
    - Variables y Tipos de Datos
    - Operadores
    - Control de Flujo
    - Arrays y Cadenas
    - Ejemplos

3. **Objetos, Clases e Interfaces.**
    - Conceptos de programación orientada a objetos
    - Crear y Utilizar Objetos
    - Declarar Clases
    - La Clase como generadora de objetos
    - Herencia
    - Métodos de objeto
    - Paso de argumentos a métodos
    - Métodos de clase (static)
    - Constructores
    - Métodos sobrecargados (overloaded)
    - Qué es un package
    - Ejemplos

Conceptos Básicos.

Breve historia de Java

Java surgió en 1991 cuando un grupo de ingenieros de Sun Microsystems trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una "máquina hipotética o virtual" denominada Java Virtual Machine (JVM). Era la JVM quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "Write Once, Run Everywhere". A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadores, Java se introdujo a finales de 1995. La clave fue la incorporación de un intérprete Java en la versión 2.0 del programa Netscape Navigator, produciendo una verdadera revolución en Internet. Java 1.1 apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje. Java 1.2, más tarde rebautizado como Java 2, nació a finales de 1998.

Al programar en Java no se parte de cero. Cualquier aplicación que se desarrolle "cuelga" (o se apoya, según como se quiera ver) en un gran número de clases preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el API o Application Programming Interface de Java). Java incorpora en el propio lenguaje muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.). Por eso muchos expertos opinan que Java es el lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo estándar, mucho más sencillo y claro que con las citadas extensiones de otros lenguajes. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo.

El principal objetivo del lenguaje Java es llegar a ser el "nexo universal" que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor de Web, en una base de datos o en cualquier otro lugar.

Java es un lenguaje muy completo (de hecho se está convirtiendo en un macro-lenguaje: Java 1.0 tenía 12 packages; Java 1.1 tenía 23 y Java 1.2 tiene 59). En cierta forma casi todo depende de casi todo. Por ello, conviene aprenderlo de modo iterativo: primero una visión muy general, que se va refinando en sucesivas iteraciones. Una forma de hacerlo es empezar con un ejemplo completo en el que ya aparecen algunas de las características más importantes.

La compañía Sun describe el lenguaje Java como "simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico". Además de una serie de halagos por parte de Sun hacia su propia criatura, el hecho es que todo ello describe bastante bien el lenguaje Java, aunque en algunas de esas características el lenguaje sea todavía bastante mejorable. Algunas de las anteriores ideas se irán explicando a lo largo de este manual.

Que es Java.

Java 2 (antes llamado Java 1.2 o JDK 1.2) es la tercera versión importante del lenguaje de programación Java.

No hay cambios conceptuales importantes respecto a Java 1.1 (en Java 1.1 sí los hubo respecto a Java 1.0), sino extensiones y ampliaciones, lo cual hace que a muchos efectos –por ejemplo, para esta introducción- sea casi lo mismo trabajar con Java 1.1 o con Java 1.2.

Los programas desarrollados en Java presentan diversas ventajas frente a los desarrollados en otros lenguajes como C/C++. La ejecución de programas en Java tiene muchas posibilidades: ejecución como aplicación independiente (Stand-alone Application), ejecución como applet, ejecución como servlet, etc. Un applet es una aplicación especial que se ejecuta dentro de un navegador o browser (por ejemplo Netscape Navigator o Internet Explorer) al cargar una página HTML desde un servidor Web. El applet se descarga desde el servidor y no requiere instalación en el ordenador donde se encuentra el browser. Un servlet es una aplicación sin interface gráfica que se ejecuta en un servidor de Internet. La ejecución como aplicación independiente es análoga a los programas desarrollados con otros lenguajes.

Además de incorporar la ejecución como Applet, Java permite fácilmente el desarrollo tanto de arquitecturas cliente-servidor como de aplicaciones distribuidas, consistentes en crear aplicaciones capaces de conectarse a otros ordenadores y ejecutar tareas en varios ordenadores simultáneamente, repartiendo por lo tanto el trabajo. Aunque también otros lenguajes de programación permiten crear aplicaciones de este tipo, Java incorpora en su propio API estas funcionalidades.

El entorno de desarrollo de Java

Existen distintos programas comerciales que permiten desarrollar código Java. La compañía Sun, creadora de Java, distribuye gratuitamente el Java(tm) Development Kit (JDK). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado Debugger). Cualquier programador con un mínimo de experiencia sabe que una parte muy importante (muchas veces la mayor parte) del tiempo destinado a la elaboración de un programa se destina a la detección y corrección de errores. Existe también una versión reducida del JDK, denominada JRE (Java Runtime Environment) destinada únicamente a ejecutar código Java (no permite compilar).

Los IDEs (Integrated Development Environment), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar Debug gráficamente, frente a la versión que incorpora el JDK basada en la utilización de una consola (denominada habitualmente ventana de comandos de Linux)
Los entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas, y ficheros resultantes de mayor tamaño que los basados en clases estándar.

El compilador de Java

Se trata de una de las herramientas de desarrollo incluidas en el JDK. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de Java (con extensión *.java). Si no encuentra errores en el código genera los ficheros compilados (con extensión *.class). En otro caso muestra la línea o líneas erróneas. En el JDK de Sun dicho compilador se llama javac. Tiene numerosas opciones, algunas de las cuales varían de una versión a otra. Se aconseja consultar la documentación de la versión del JDK utilizada para obtener una información detallada de las distintas posibilidades.

La Java Virtual Machine

Tal y como se ha comentado al comienzo del capítulo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de Sun a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se planteó la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código "neutro" el cual estuviera preparado para ser ejecutado sobre una "máquina hipotética o virtual", denominada Java Virtual Machine (JVM). Es esta JVM quien interpreta este código neutro convirtiéndolo a código particular de la CPU utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La JVM es el intérprete de Java. Ejecuta los "bytecodes" (ficheros compilados con extensión *.class) creados por el compilador de Java (javac). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado JIT (Just-In-Time Compiler), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

Las variables PATH y CLASSPATH

El desarrollo y ejecución de aplicaciones en Java exige que las herramientas para compilar (javac) y ejecutar (java) se encuentren accesibles. El ordenador, desde una ventana de comandos de Linux, sólo es capaz de ejecutar los programas que se encuentran en los directorios indicados en la variable PATH del ordenador (o en el directorio activo). Si se desea compilar o ejecutar código en Java, el directorio donde se encuentran estos programas (java y javac) deberá encontrarse en el PATH. Tecleando $PATH en una ventana de comandos de Linux se muestran los nombres de directorios incluidos en dicha variable de entorno.

Java utiliza además una nueva variable de entorno denominada CLASSPATH, la cual determina dónde buscar tanto las clases o librerías de Java (el API de Java) como otras clases de usuario. A partir de la versión 1.1.4 del JDK no es necesario indicar esta variable, salvo que se desee añadir conjuntos de clases de usuario que no vengan con dicho JDK. La variable CLASSPATH puede incluir la ruta de directorios o ficheros *.zip o *.jar en los que se encuentren los ficheros *.class.

Si no la tienes agregadas en tus variables de entorno, la agregas con:

```
export JAVA_HOME=/opt/java/jdk1.6.0_06/
export PATH=/opt/java/jdk1.6.0_06//bin
```

Los parámetros son:

```
JAVA_HOME=/opt/java/jdk1.6.0_06/
PATH=/opt/java/jdk1.6.0_06//bin
```

CLASSPATH=/opt/java/jdk1.6.0_06/jre/lib/jce.jar

lo cual sería válido en el caso de que el JDK estuviera situado en el directorio /opt/java/jdk1.6.0_06/.

Cuando un fichero filename.java se compila y en ese directorio existe ya un fichero filename.class, se comparan las fechas de los dos ficheros. Si el fichero filename.java es más antiguo que el filename.class no se produce un nuevo fichero filename.class. Esto sólo es válido para ficheros *.class que se corresponden con una clase public.

Características del Lenguaje.

En este capítulo se presentan las características generales de Java como lenguaje de programación algorítmico. En este apartado Java es muy similar a C/C++, lenguajes en los que está inspirado. Se va a intentar ser breve, considerando que el lector ya conoce algunos otros lenguajes de programación y está familiarizado con lo que son variables, bifurcaciones, bucles, etc.

Variables

Una variable es un nombre que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

1. Variables de tipos primitivos. Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. Java permite distinta precición y distintos rangos de valores para estos tipos de variables (char, byte, short, int, long, float, double, boolean). Ejemplos de variables de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', True, etc.
2. Variables referencia. Las variables referencia son referencias o nombres de una información más compleja: arrays u objetos de una determinada clase.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

1. Variables miembro de una clase: Se definen en una clase, fuera de cualquier método; pueden ser tipos primitivos o referencias.
2. Variables locales: Se definen dentro de un método o más en general dentro de cualquier bloque entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también tipos primitivos o referencias.

Nombres de Variables

Los nombres de variables en Java se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por Java como operadores o separadores (,.+-*/ etc.).

Existe una serie de palabras reservadas las cuales tienen un significado especial para Java y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

abstract	boolean	break	byte	case	catch
char	class	const*	continue	default	do
double	else	extends	final	finally	float
for	goto*	if	implements	import	instanceof
int	interface	long	native	new	null
package	private	protected	public	return	short
static	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje Java.

Tipos Primitivos de Variables

Se llaman tipos primitivos de variables de Java a aquellas variables sencillas que contienen los tipos de información más habituales: valores boolean, caracteres y valores numéricos enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores true y false (boolean); un tipo para almacenar caracteres (char), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (byte, short, int y long) y dos para valores reales de punto flotante (float y double). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la Tabla 2.1.

Tipo de variable	Descripción
Boolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38

Double 8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 2.1. Tipos primitivos de variables en Java.

Los tipos primitivos de Java tienen algunas características importantes que se resumen a continuación:

1. El tipo boolean no es un valor numérico: sólo admite los valores true o false. El tipo boolean no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser boolean.
2. El tipo char contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.
3. Los tipos byte, short, int y long son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en Java no hay enteros unsigned.
4. Los tipos float y double son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.
5. Se utiliza la palabra void para indicar la ausencia de un tipo de variable determinado.
6. A diferencia de C/C++, los tipos de variables en Java están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un int ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.
7. Existen extensiones de Java 1.2 para aprovechar la arquitectura de los procesadores Intel, que permiten realizar operaciones de punto flotente con una precisión extendida de 80 bits.

Cómo se definen e inicializan las variables

Una variable se define especificando el tipo y el nombre de dicha variable. Estas variables pueden ser tanto de tipos primitivos como referencias a objetos de alguna clase perteneciente al API de Java o generada por el usuario. Si no se especifica un valor en su declaración, las variable primitivas se inicializan a cero (salvo boolean y char, que se inicializan a false y '\0'). Análogamente las variables de tipo referencia son inicializadas por defecto a un valor especial: null.

Es importante distinguir entre la referencia a un objeto y el objeto mismo. Una referencia es una variable que indica dónde está guardado un objeto en la memoria del ordenador (a diferencia de C/C++, Java no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los punteros). Al declarar una referencia todavía no se encuentra “apuntando” a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor null. Si se desea que esta referencia apunte a un nuevo objeto es necesario crear el

objeto utilizando el operador new. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la referencia declarada a otra referencia a un objeto existente previamente.

Un tipo particular de referencias son los arrays o vectores, sean éstos de variables primitivas (por ejemplo, un vector de enteros) o de objetos. En la declaración de una referencia de tipo array hay que incluir los corchetes []. En los siguientes ejemplos aparece cómo crear un vector de 10 números enteros y cómo crear un vector de elementos MyClass. Java garantiza que los elementos del vector son inicializados a null o a cero (según el tipo de dato) en caso de no indicar otro valor.

Ejemplos de declaración e inicialización de variables:

```
int x;                    // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5;                // Declaración de la variable primitiva y. Se inicializa a 5
MyClass unaRef;           // Declaración de una referencia a un objeto MyClass.
                          // Se inicializa a null
unaRef = new MyClass();   // La referencia "apunta" al nuevo objeto creado
                          // Se ha utilizado el constructor por defecto
MyClass segundaRef = unaRef;  // Declaración de una referencia a un objeto MyClass.
                          // Se inicializa al mismo valor que unaRef
int [] vector;            // Declaración de un array. Se inicializa a null
vector = new int[10];     // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1};   // Declaración e inicialización de un vector de 3
                          // elementos con los valores entre llaves
MyClass [] lista=new MyClass[5]; // Se crea un vector de 5 referencias a objetos
                          // Las 5 referencias son inicializadas a null
lista[0] = unaRef;        // Se asigna a lista[0] el mismo valor que unaRef
lista[1] = new MyClass(); // Se asigna a lista[1] la referencia al nuevo objeto
                          // El resto (lista[2]...lista[4] siguen con valor null
```

En el ejemplo mostrado las referencias unaRef, segundaRef y lista[0] actuarán sobre el mismo objeto. Es equivalente utilizar cualquiera de las referencias ya que el objeto al que se refieren es el mismo.

Visibilidad y vida de las variables

Se entiende por visibilidad, ámbito o scope de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en una expresión. En Java todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves {}, es

decir dentro de un bloque, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque if no serán válidas al finalizar las sentencias correspondientes a dicho if y las variables miembro de una clase (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Las variables miembro de una clase declaradas como public son accesibles a través de una referencia a un objeto de dicha clase utilizando el operador punto (.). Las variables miembro declaradas como private no son accesibles directamente desde otras clases. Las funciones miembro de una clase tienen acceso directo a todas las variables miembro de la clase sin necesidad de anteponer el nombre de un objeto de la clase. Sin embargo las funciones miembro de una clase B derivada de otra A, tienen acceso a todas las variables miembro de A declaradas como public o protected, pero no a las declaradas como private. Una clase derivada sólo puede acceder directamente a las variables y funciones miembro de su clase base declaradas como public o protected. Otra característica del lenguaje es que es posible declarar una variable dentro de un bloque con el mismo nombre que una variable miembro, pero no con el nombre de otra variable local que ya existiera. La variable declarada dentro del bloque oculta a la variable miembro en ese bloque. Para acceder a la variable miembro oculta será preciso utilizar el operador this, en la forma this.varname.

Uno de los aspectos más importantes en la programación orientada a objetos (OOP) es la forma en la cual son creados y eliminados los objetos. En Java la forma de crear nuevos objetos es utilizando el operador new. Cuando se utiliza el operador new, la variable de tipo referencia guarda la posición de memoria donde está almacenado este nuevo objeto. Para cada objeto se lleva cuenta de por cuántas variables de tipo referencia es apuntado. La eliminación de los objetos la realiza el programa denominado garbage collector, quien automáticamente libera o borra la memoria ocupada por un objeto cuando no existe ninguna referencia apuntando a ese objeto. Lo anterior significa que aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no es eliminado si hay otras referencias apuntando a ese mismo objeto.

Operadores de Java

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen brevemente en los apartados siguientes.

Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: suma (+), resta (-), multiplicación (*), división (/) y resto de la división (%).

Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el operador igual (=). La forma general de las sentencias de asignación con este operador es:

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 – op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

variable = expression;

Tabla 2.2. Otros operadores de asignación.

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones “acumulativas” sobre una variable. La Tabla 2.2 muestra estos operadores y su equivalencia con el uso del operador igual (=).

Operadores unarios

Los operadores más (+) y menos (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en Java es el estándar de estos operadores.

Operador instanceof

El operador instanceof permite saber si un objeto pertenece o no a una determinada clase. Es un operador binario cuya forma general es,

objectName instanceof ClassName

y que devuelve true o false según el objeto pertenezca o no a la clase.

Operador condicional ?:

Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

booleanExpression ? res1 : res2

donde se evalúa booleanExpression y se devuelve res1 si el resultado es true y res2 si el resultado es false. Es el único operador ternario (tres argumentos) de Java. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

x=1 ; y=10; z = (x<y)?x+3:y+8;

asignarían a z el valor 4, es decir x+3.

Operadores incrementales

Java dispone del operador incremento (++) y decremento (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

1. Precediendo a la variable (por ejemplo: ++i). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
2. Siguiendo a la variable (por ejemplo: i++). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles for es una de las aplicaciones más frecuentes de estos operadores.

Operadores relacionales

Los operadores relacionales sirven para
realizar comparaciones de igualdad,
desigualdad y relación de menor o mayor.
El resultado de estos operadores es

siempre un valor boolean (true o false)
según se cumpla o no la relación
considerada.

La Tabla 2.3 muestra los operadores relacionales de Java.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Tabla 2.3. Operadores relacionales.
operadores relacionales de Java.

Estos operadores se utilizan con mucha frecuencia en las bifurcaciones y en los bucles, que se verán en próximos apartados de este capítulo.

Operadores lógicos

Los operadores lógicos se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales. La Tabla 2.4 muestra los operadores lógicos de Java. Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser true y el primero es false, ya se sabe que la condición de que ambos sean true no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

Tabla 2.4. Operadores lógicos.

Operador de concatenación de cadenas de caracteres (+)

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método println(). La variable numérica result es convertida automáticamente por Java en cadena de caracteres para poderla concatenar. En otras ocasiones se

deberá llamar explícitamente a un método para que realice esta conversión.

Operadores que actúan a nivel de bits

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o flags, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no. La Tabla 2.5 muestra los operadores de Java que actúan a nivel de bits.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1)
~	~op2	Operador complemento (invierte el valor de cada bit)

Tabla 2.5. Operadores a nivel de bits.

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000), utilizando sólo 8 bits. La suma de estos números permite construir una variable flags con los bits activados que se deseen. Por ejemplo, para construir una variable flags que sea 00010010 bastaría hacer flags=2+16. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

```
if (flags & 2 == 2) {...}
```

La Tabla 2.6 muestra los operadores de asignación a nivel de bits.

Operador	Utilización	Equivalente a
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Tabla 2.6. Operadores de asignación a nivel de bits.

Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de x/y*z depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de mayor a menor precedencia:

postfix operators	[] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type)expr
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

En Java, todos los operadores binarios, excepto los operadores de asignación, se evalúan de izquierda a derecha. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

Estructuras de programación

En este apartado se supone que el lector tiene algunos conocimientos de programación y por lo tanto
no se explican en profundidad los conceptos que aparecen.

Las estructuras de programación o estructuras de control permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados bifurcaciones y bucles. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro. La sintaxis de Java coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

Sentencias o expresiones

Una expresión es un conjunto variables unidos por operadores. Son órdenes que se le dan al computador para que realice una tarea determinada.

Una sentencia es una expresión que acaba en punto y coma (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias
```

Comentarios

Existen dos formas diferentes de introducir comentarios entre el código de Java (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras “//” en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos /*...*/. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
```

```
// las dos barras. Requiere incluir dos barras al comienzo de cada línea
/* Esta segunda forma es mucho más cómoda para comentar un número elevado de líneas
   ya que sólo requiere modificar
   el comienzo y el final. */
```

En Java existe además una forma especial de introducir los comentarios (utilizando /**...*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las clases y packages desarrollados por el programador. Una vez introducidos los comentarios, el programa javadoc.exe (incluido en el JDK) genera de forma automática la información de forma similar a la presentada en la propia documentación del JDK. La sintaxis de estos comentarios y la forma de utilizar el programa javadoc.exe se puede encontrar en la información que viene con el JDK.

Bifurcaciones

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el flujo de ejecución de un programa. Existen dos bifurcaciones diferentes: if y switch.

Bifurcación if

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor true). Tiene la forma siguiente:

```
if (booleanExpression) {
    statements;
}
```

Las llaves {} sirven para agrupar en un bloque las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del if.

Bifurcación if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el else se ejecutan en
el caso de no cumplirse la expresión de comparación (false),

```
if (booleanExpression) {
```

```
    statements1;
} else {
    statements2;
}
```

Bifurcación if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al else.

```
if (booleanExpression1) {
    statements1;
} else if (booleanExpression2) {
    statements2;
} else if (booleanExpression3) {
    statements3;
} else {
    statements4;
}
```

Véase a continuación el siguiente ejemplo:

```
int numero = 61;                        // La variable "numero" tiene dos dígitos
if(Math.abs(numero) < 10)                   // Math.abs() calcula el valor absoluto. (false)
    System.out.println("Numero tiene 1 digito ");
else if (Math.abs(numero) < 100) // Si numero es 61, estamos en este caso (true)
    System.out.println("Numero tiene 1 digito ");
else {                                  // Resto de los casos
    System.out.println("Numero tiene mas de 3 digitos ");
    System.out.println("Se ha ejecutado la opcion por defecto ");
}
```

Sentencia switch

Se trata de una alternativa a la bifurcación if elseif else cuando se compara la misma expresión con distintos valores. Su forma general es la siguiente:

```
switch (expression) {
```

```
    case value1: statements1;          break;
    case value2: statements2;          break;
    case value3: statements3;          break;
    case value4: statements4;          break;
    case value5: statements5;          break;
    case value6: statements6;          break;
    [default: statements7;]
}
```

Las características más relevantes de switch son las siguientes:

1. Cada sentencia case se corresponde con un único valor de expression. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos. El ejemplo del Apartado 2.3.3.3 no se podría realizar utilizando switch.
2. Los valores no comprendidos en ninguna sentencia case se pueden gestionar en default, que es opcional.
3. En ausencia de break, cuando se ejecuta una sentencia case se ejecutan también todas las case que van a continuación, hasta que se llega a un break o hasta que se termina el switch.

Ejemplo:

```
char c = (char)(Math.random()*26+'a'); // Generación aleatoria de letras minúsculas
System.out.println("La letra " + c );
switch (c) {
   case 'a': // Se compara con la letra a
   case 'e': // Se compara con la letra e
   case 'i': // Se compara con la letra i
   case 'o': // Se compara con la letra o
   case 'u': // Se compara con la letra u
       System.out.println(" Es una vocal "); break;
   default:
       System.out.println(" Es una consonante ");
}
```

Bucles

Un bucle se utiliza para realizar un proceso repetidas veces. Se denomina también lazo o loop. El código incluido entre las llaves {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a

los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (booleanExpression) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

Bucle while

Las sentencias statements se ejecutan mientras booleanExpression sea true.

```
while (booleanExpression) {
    statements;
}
```

Bucle for

La forma general del bucle for es la siguiente:

```
for (initialization; booleanExpression; increment) {
    statements;
}
```

que es equivalente a utilizar while en la siguiente forma,

```
initialization;
while (booleanExpression) {
    statements;
    increment;
}
```

La sentencia o sentencias initialization se ejecuta al comienzo del for, e increment después de statements. La booleanExpression se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de comparación toma el valor false. Cualquiera de las tres partes puede estar vacía. La initialization y el increment pueden tener varias expresiones separadas por comas.

Por ejemplo, el código situado a la izquierda produce la salida que aparece a la derecha:

Código:

```
for(int i = 1, j = i + 10; i < 5; i++, j = 2*i) {
    System.out.println(" i = " + i + " j = " + j);
}
```

Salida:

```
i = 1  j = 11
i = 2  j = 4
i = 3  j = 6
i = 4  j = 8
```

Bucle do while

Es similar al bucle while pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los statements, se evalúa la condición: si resulta true se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a false finaliza el bucle. Este tipo de bucles se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

```
do {
    statements
} while (booleanExpression);
```

Sentencias break y continue

La sentencia break es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando, sin sin realizar la ejecución del resto de las sentencias.

La sentencia continue se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración “i” que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración (i+1).

Sentencias break y continue con etiquetas

Las etiquetas permiten indicar un lugar donde continuar la ejecución de un programa después de un break o continue. El único lugar donde se pueden incluir etiquetas es justo delante de un bloque de código entre llaves {} (if, switch, do...while, while, for) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (break) o continuar con la siguiente iteración (continue) de un bucle que no es el actual.

Por tanto, la sentencia break labelName finaliza el bloque que se encuentre a continuación de labelName. Por ejemplo, en las sentencias,

```
bucleI:       // etiqueta o label
for ( int i = 0, j = 0; i < 100; i++){
    while ( true ) {
        if( (++j) > 5) { break bucleI; }                    // Finaliza ambos bucles
        else { break; }                                     // Finaliza el bucle interior (while)
```

```
        }
    }
```

la expresión break bucleI; finaliza los dos bucles simultáneamente, mientras que la expresión break; sale del bucle while interior y seguiría con el bucle for en i. Con los valores presentados ambos bucles finalizarán con i = 5 y j = 6 (se invita al lector a comprobarlo).

La sentencia continue (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia,

```
continue bucle1;
```

transfiere el control al bucle for que comienza después de la etiqueta bucle1: para que realice una nueva iteración, como por ejemplo:

```
bucle1:
for (int i=0; i<n; i++) {
    bucle2:
    for (int j=0; j<m; j++) {
        ...
        if (expression) continue bucle1; then continue bucle2;
        ...
    }
}
```

Sentencia return

Otra forma de salir de un bucle (y de un método) es utilizar la sentencia return. A diferencia de continue o break, la sentencia return sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (return value;).

Bloque try {...} catch {...} finally {...}

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje Java, una Exception es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas excepciones son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un

mensaje explicando el tipo de error que se ha producido. Otras excepciones, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (definiendo por ejemplo un nuevo path del fichero no encontrado).

Los errores se representan mediante clases derivadas de la clase Throwable, pero los que tiene
que chequear un programador derivan de Exception (java.lang.Exception que a su vez deriva de Throwable). Existen algunos tipos de excepciones que Java obliga a tener en cuenta. Esto se hace mediante el uso de bloques try, catch y finally.

El código dentro del bloque try está “vigilado”. Si se produce una situación anormal y se lanza
como consecuencia una excepción, el control pasa al bloque catch, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques catch como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque finally, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error.

En el caso en que el código de un método pueda generar una Exception y no se desee incluir en dicho método la gestión del error (es decir los bucles try/catch correspondientes), es necesario que el método pase la Exception al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra throws seguida del nombre de la Exception concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques try/catch o volver a pasar la Exception. De esta forma se puede ir pasando la Exception de un método a otro hasta llegar al último método del programa, el método main().

En el siguiente ejemplo se presentan dos métodos que deben "controlar" una IOException relacionada con la lectura ficheros y una MyException propia. El primero de ellos (metodo1) realiza la gestión de las excepciones y el segundo (metodo2) las pasa al siguiente método.

```
void metodo1() {
    ...
    try {
        ... // Código que puede lanzar las excepciones IOException y MyException
    } catch (IOException e1) {// Se ocupa de IOException simplemente dando aviso
        System.out.println(e1.getMessage());
    } catch (MyException e2) {
        // Se ocupa de MyException dando un aviso y finalizando la función
        System.out.println(e2.getMessage()); return;
    } finally { // Sentencias que se ejecutarán en cualquier caso
        ...
```

```
    }]
    ...
} // Fin del metodo1
void metodo2() throws IOException, MyException {
    ...
    // Código que puede lanzar las excepciones IOException y MyException
    ...
} // Fin del metodo2
```

Arrays y Cadenas.

Arrays Unidimensionales.

Un array es una colección de valores de un mismo tipo engrosados en la misma variable. De forma que se puede acceder a cada valor independientemente. Para Java además un array es un objeto que tiene propiedades que se pueden manipular.

Los arrays solucionan problemas concernientes al manejo de muchas variables que se refieren a datos similares. Por ejemplo si tuviéramos la necesidad de almacenar las notas de una clase con 18 alumnos, necesitaríamos 18 variables, con la tremenda lentitud de manejo que supone eso. Solamente calcular la nota media requeriría una tremenda línea de código. Almacenar las notas supondría al menos 18 líneas de código.

Gracias a los arrays se puede crear un conjunto de variables con el mismo nombre. La diferencia será que un número (índice del array) distinguirá a cada variable.

En el caso de las notas, se puede crear un array llamado notas, que representa a todas las notas de la clase. Para poner la nota del primer alumno se usaría notas[0], el segundo sería notas[1], etc. (los corchetes permiten especificar el índice en concreto del array).

La declaración de un array unidimensional se hace con esta sintaxis.

```
tipo nombre[];
```

Ejemplo:

```
double cuentas[]; //Declara un array que almacenará valores
                  // doubles
```

Declara un array de tipo double. Esta declaración indica para qué servirá el array, pero no reserva espacio en la RAM al no saberse todavía el tamaño del mismo.

Tras la declaración del array, se tiene que iniciar. Eso lo realiza el operador new,

que es el que realmente crea el array indicando un tamaño. Cuando se usa new es cuando se reserva el espacio necesario en memoria. Un array no inicializado es un array null. Ejemplo:

```
int notas[]; //sería válido también int[] notas;
notas = new int[3]; //indica que el array constará de tres
                    //valores de tipo int
//También se puede hacer todo a la vez
//int notas[]=new int[3];
```

En el ejemplo anterior se crea un array de tres enteros (con los tipos básicos se crea en memoria el array y se inicializan los valores, los números se inician a 0).

Los valores del array se asignan utilizando el índice del mismo entre corchetes:

```
notas[2]=8;
```

También se pueden asignar valores al array en la propia declaración:

```
int notas[] = {8, 7, 9};
int notas2[]= new int[] {8,7,9};//Equivalente a la anterior
```

Esto declara e inicializa un array de tres elementos. En el ejemplo lo que significa es que notas[0] vale 8, notas[1] vale 7 y notas[2] vale 9.

En Java (como en otros lenguajes) el primer elemento de un array es el cero. El primer elemento del array notas, es notas[0]. Se pueden declarar arrays a cualquier tipo de datos (enteros, booleanos, doubles, ... e incluso objetos).

La ventaja de usar arrays (volviendo al caso de las notas) es que gracias a un simple bucle for se puede rellenar o leer fácilmente todos los elementos de un array:

```
//Calcular la media de las 18 notas
suma=0;
for (int i=0;i<=17;i++){
  suma+=nota[i];
}
media=suma/18;
```

A un array se le puede inicializar las veces que haga falta:

```
int notas[]=new notas[16];
...
notas=new notas[25];
```

Pero hay que tener en cuenta que el segundo new hace que se pierda el contenido anterior. Realmente un array es una referencia a valores que se almacenan en memoria

mediante el operador new, si el operador new se utiliza en la misma referencia, el anterior contenido se queda sin referencia y, por lo tanto se pierde.

Un array se puede asignar a otro array (si son del mismo tipo):

```
int notas[];
int ejemplo[]=new int[18];
notas=ejemplo;
```

En el último punto, notas equivale a ejemplo. Esta asignación provoca que cualquier cambio en notas también cambie el array ejemplos. Es decir esta asignación anterior, no copia los valores del array, sino que notas y ejemplo son referencias al mismo array.
Ejemplo:

```
int notas[]={3,3,3};
int ejemplo[]=notas;
ejemplo= notas;
ejemplo[0]=8;
System.out.println(notas[0]);//Escribirá el número 8
```

Arrays Multidimensionales.

Los arrays además pueden tener varias dimensiones. Entonces se habla de arrays de arrays (arrays que contienen arrays) Ejemplo:

```
int notas[][];
```

notas es un array que contiene arrays de enteros

```
notas = new int[3][12];//notas está compuesto por 3 arrays
                       //de 12 enteros cada uno
notas[0][0]=9;//el primer valor es 0
```

Puede haber más dimensiones incluso (notas[3][2][7]). Los arrays multidimensionales se pueden inicializar de forma más creativa incluso. Ejemplo:

```
int notas[][]=new int[5][];//Hay 5 arrays de enteros
notas[0]=new int[100]; //El primer array es de 100 enteros
notas[1]=new int[230]; //El segundo de 230
notas[2]=new int[400];
notas[3]=new int[100];
notas[4]=new int[200];
```

Hay que tener en cuenta que en el ejemplo anterior, notas[0] es un array de 100 enteros. Mientras que notas, es un array de 5 arrays de enteros.

Se pueden utilizar más de dos dimensiones si es necesario.

longitud de un array

Los arrays poseen un método que permite determinar cuánto mide un array. Se trata de length. Ejemplo (continuando del anterior):

```
System.out.println(notas.length); //Sale 5
System.out.println(notas[2].length); //Sale 400
```

la clase Arrays.

En el paquete java.utils se encuentra una clase estática llamada Arrays. Una clase estática permite ser utilizada como si fuera un objeto (como ocurre con Math). Esta clase posee métodos muy interesantes para utilizar sobre arrays.

Su uso es

```
Arrays.método(argumentos);
```

fill

Permite rellenar todo un array unidimensional con un determinado valor. Sus argumentos son el array a rellenar y el valor deseado:

```
int valores[]=new int[23];
Arrays.fill(valores,-1);//Todo el array vale -1
```

También permite decidir desde que índice hasta qué índice rellenamos:

```
Arrays.fill(valores,5,8,-1);//Del elemento 5 al 7 valdrán -1
```

equals

Compara dos arrays y devuelve true si son iguales. Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores.

sort

Permite ordenar un array en orden ascendente. Se pueden ordenar sólo una serie de elementos desde un determinado punto hasta un determinado punto.

```
int x[]={4,5,2,3,7,8,2,3,9,5};
Arrays.sort(x);//Estará ordenado
Arrays.sort(x,2,5);//Ordena del 2o al 4o elemento
```

binarySearch

Permite buscar un elemento de forma ultrarrápida en un array ordenado (en un array desordenado sus resultados son impredecibles). Devuelve el índice en el que está colocado el elemento. Ejemplo:

```
int x[]={1,2,3,4,5,6,7,8,9,10,11,12};
Arrays.sort(x);
System.out.println(Arrays.binarySearch(x,8));//Da
    7
```

El método System.arraysCopy

La clase System también posee un método relacionado con los arrays, dicho método permite copiar un array en otro. Recibe cinco argumentos: el array que se copia, el índice desde que se empieza a copia en el origen, el array destino de la copia, el índice desde el que se copia en el destino, y el tamaño de la copia (número de elementos de la copia).

```
int uno[]={1,1,2};
int dos[]={3,3,3,3,3,3,3,3,3};
System.arraycopy(uno, 0, dos, 0, uno.length);
for (int i=0;i<=8;i++){
    System.out.print(dos[i]+" ");
} //Sale 112333333
```

Clase String

Introducción

Para Java las cadenas de texto son objetos especiales. Los textos deben manejarse creando objetos de tipo String. Ejemplo:

```
String texto1 = “¡Prueba de texto!”;
```

Las cadenas pueden ocupar varias líneas utilizando el operador de concatenación “+”.

```
String texto2 =”Este es un texto que ocupa “ +
            “varias líneas, no obstante se puede “+
            “perfectamente encadenar”;
```

También se pueden crear objetos String sin utilizar constantes entrecomilladas, usando otros constructores:

```
char[]    palabra = {‘P’,’a’,’l’,’b’,’r’,’a’};//Array de char
String    cadena = new String(palabra);
byte[]    datos = {97,98,99};
String    codificada = new String (datos, “8859_1”);
```

En el último ejemplo la cadena codificada se crea desde un array de tipo byte que contiene números que serán interpretados como códigos Unicode. Al asignar, el valor 8859_1 indica la tabla de códigos a utilizar.

comparación entre objetos String

Los objetos String no pueden compararse directamente con los operadores de comparación. En su lugar se deben utilizar estas expresiones:

cadena1.equals(cadena2). El resultado es true si la cadena1 es igual a la cadena2. Ambas cadenas son variables de tipo String.

cadena1.equalsIgnoreCase(cadena2). Como la anterior, pero en este caso no se tienen en cuenta mayúsculas y minúsculas.

s1.compareTo(s2). Compara ambas cadenas, considerando el orden alfabético. Si la primera cadena es mayor en orden alfabético que la segunda devuelve 1, si son iguales devuelve 0 y si es la segunda la mayor devuelve -1. Hay que tener en cuenta que el orden no es el del alfabeto español, sino que usa la tabla ASCII, en esa tabla la letra ñ es mucho mayor que la o.

s1.compareToIgnoreCase(s2). Igual que la anterior, sólo que además ignora las mayúsculas (disponible desde Java 1.2)

String.valueOf

Este método pertenece no sólo a la clase String, sino a otras y siempre es un método que convierte valores de una clase a otra. En el caso de los objetos String, permite convertir valores que no son de cadena a forma de cadena. Ejemplos:

```
String numero = String.valueOf(1234);
String fecha = String.valueOf(new Date());
```

En el ejemplo se observa que este método pertenece a la clase String directamente, no hay que utilizar el nombre del objeto creado (como se verá más adelante, es un método estático).

Métodos de las variables de las cadenas

Son métodos que poseen las propias variables de cadena. Para utilizarlos basta con poner el nombre del método y sus parámetros después del nombre de la variable String. Es decir: variableString.método(argumentos)

length

Permite devolver la longitud de una cadena (el número de caracteres de la cadena):

```
String texto1=”Prueba”;
System.out.println(texto1.length());//Escribe 6
```

Concatenar cadenas

Se puede hacer de dos formas, utilizando el método concat o con el operador +. Ejemplo:

```
String s1=”Buenos ”, s2=”días”, s3, s4;
s3 = s1 + s2;
s4 = s1.concat(s2);
```

charAt

Devuelve un carácter de la cadena. El carácter a devolver se indica por su posición (el primer carácter es la posición 0) Si la posición es negativa o sobrepasa el tamaño de la cadena, ocurre un error de ejecución, una excepción tipo IndexOutOfBounds-Exception. Ejemplo:

```
String s1=”Prueba”;
char c1=s1.charAt(2); //c1 valdrá ‘u’
```

substring

Da como resultado una porción del texto de la cadena. La porción se toma desde una posición inicial hasta una posición final (sin incluir esa posición final). Si las posiciones indicadas no son válidas ocurre una excepción de tipo IndexOutOfBounds-Exception. Se empieza a contar desde la posición 0. Ejemplo:

```
String s1=”Buenos días”;

String s2=s1.substring(7,10); //s2 = día
```

indexOf

Devuelve la primera posición en la que aparece un determinado texto en la cadena. En el caso de que la cadena buscada no se encuentre, devuelve -1. El texto a buscar puede ser char o String. Ejemplo:

```
String s1=”Quería decirte que quiero que te vayas”;
System.out.println(s1.indexOf(“que”)); //Da 15
```

Se puede buscar desde una determinada posición. En el ejemplo anterior:

```
System.out.println(s1.indexOf(“que”,16)); //Ahora da 26
```

lastIndexOf

Devuelve la última posición en la que aparece un determinado texto en la cadena. Es casi idéntica a la anterior, sólo que busca desde el final. Ejemplo:

```
String s1=”Quería decirte que quiero que te vayas”;
System.out.println(s1.lastIndexOf(“que”); //Da 26
```

También permite comenzar a buscar desde una determinada posición.

endsWith

Devuelve true si la cadena termina con un determinado texto. Ejemplo:

```
String s1=”Quería decirte que quiero que te vayas”;
System.out.println(s1.endsWith(“vayas”); //Da true
```

startsWith

Devuelve true si la cadena empieza con un determinado texto.

replace

Cambia todas las apariciones de un carácter por otro en el texto que se indique y lo almacena como resultado. El texto original no se cambia, por lo que hay que asignar el resultado de replace a un String para almacenar el texto cambiado:

```
String s1=”Mariposa”;
System.out.println(s1.replace(‘a’,’e’));//Da Meripose
System.out.println(s1);//Sigue valiendo Mariposa
```

replaceAll

Modifica en un texto cada entrada de una cadena por otra y devuelve el resultado. El primer parámetro es el texto que se busca (que puede ser una expresión regular), el segundo parámetro es el texto con el que se reemplaza el buscado. La cadena original no se modifica.

```
String s1=”Cazar armadillos”;
System.out.println(s1.replace(“ar”,”er”));//Da Cazer ermedillos
System.out.println(s1);//Sigue valiendo Cazar armadilos
```

toUpperCase

Devuelve la versión en mayúsculas de la cadena.

toLowerCase

Devuelve la versión en minúsculas de la cadena.

toCharArray

Obtiene un array de caracteres a partir de una cadena.

Lista completa de métodos

método	descripción
char charAt(int index)	Proporciona el carácter que está en la posición dada por el entero index.
int compareTo(string s)	Compara las dos cadenas. Devuelve un valor menor que cero si la cadena s es mayor que la original, devuelve 0 si son iguales y devuelve un valor mayor que cero si s es menor que la original.
int compareToIgnoreCase(string s)	Compara dos cadenas, pero no tiene e cuenta si el texto es mayúsculas o no.
String concat(String s)	Añade la cadena s a la cadena original.

String copyValueOf(char[] data)	Produce un objeto String que es igual al array de caracteres data.
boolean endsWith(String s)	Devuelve true si la cadena termina con el texto s
boolean equals(String s)	Compara ambas cadenas, devuelve true si son iguales
boolean equalsIgnoreCase(String s)	Compara ambas cadenas sin tener en cuenta las mayúsculas y las minúsculas.
byte[] getBytes()	Devuelve un array de caracteres que toma a partir de la cadena de texto
void getBytes(int srcBegin, int srcEnd, char[] dest, int dstBegin);	Almacena el contenido de la cadena en el array de caracteres dest. Toma los caracteres desde la posición srcBegin hasta la posición srcEnd y les copia en el array desde la posición dstBegin
int indexOf(String s)	Devuelve la posición en la cadena del texto s
int indexOf(String s, int primeraPos)	Devuelve la posición en la cadena del texto s, empezando a buscar desde la posición PrimeraPos
int lastIndexOf(String s)	Devuelve la última posición en la cadena del texto s

método	descripción
int lastIndexOf(String s, int primeraPos)	Devuelve la última posición en la cadena del texto s, empezando a buscar desde la posición PrimeraPos
int length()	Devuelve la longitud de la cadena

String replace(char carAnterior, char ncarNuevo)	Devuelve una cadena idéntica al original pero que ha cambiando los caracteres iguales a carAnterior por carNuevo
String replaceFirst(String str1, String str2)	Cambia la primera aparición de la cadena str1 por la cadena str2
String replaceFirst(String str1, String str2)	Cambia la primera aparición de la cadena uno por la cadena dos
String replaceAll(String str1, String str2)	Cambia la todas las apariciones de la cadena uno por la cadena dos
String startsWith(String s)	Devuelve true si la cadena comienza con el texto s.
String substring(int primeraPos, int segundaPos)	Devuelve el texto que va desde primeraPos a segunaPos.
char[] toCharArray()	Devuelve un array de caracteres a partir de la cadena dada
String toLowerCase()	Convierte la cadena a minúsculas
String toLowerCase(Locale local)	Lo mismo pero siguiendo las instrucciones del argumento local
String toUpperCase()	Convierte la cadena a mayúsculas
String toUpperCase(Locale local)	Lo mismo pero siguiendo las instrucciones del argumento local
String trim()	Elimina los blancos que tenga la cadena tanto por delante como por detrás

Static String valueOf(tipo elemento)	Devuelve la cadena que representa el valor elemento. Si elemento es booleano, por ejemplo devolvería una cadena con el valor true o false

Objetos y Clases

Programación Orientada a Objetos

Se ha comentado anteriormente en este manual que Java es un lenguaje totalmente orientado a objetos. De hecho siempre hemos definido una clase pública con un método main que permite que se pueda visualizar en la pantalla el programa Java.

La gracia de la POO es que se hace que los problemas sean más sencillos, al permitir dividir el problema. Está división se hace en objetos, de forma que cada objeto funcione de forma totalmente independiente. Un objeto es un elemento del programa que posee sus propios datos y su propio funcionamiento.

Es decir un objeto está formado por datos (propiedades) y funciones que es capaz de realizar el objeto (métodos).

Antes de poder utilizar un objeto, se debe definir su clase. La clase es la definición de un tipo de objeto. Al definir una clase lo que se hace es indicar como funciona un determinado tipo de objetos. Luego, a partir de la clase, podremos crear objetos de esa

Propiedades de la POO

Encapsulamiento

Una clase se compone tanto de variables (propiedades) como de funciones y procedimientos (métodos). De hecho no se pueden definir variables (ni funciones) fuera de una clase (es decir no hay variables globales).
Ocultación. Hay una zona oculta al definir la clases (zona privada) que sólo es utilizada por esa clases y por alguna clase relacionada. Hay una zona pública (llamada también interfaz de la clase) que puede ser utilizada por cualquier parte del código.

Polimorfismo

Cada método de una clase puede tener varias definiciones distintas. En el caso del parchís: partida.empezar(4) empieza una partida para cuatro jugadores, partida.empezar(rojo, azul) empieza una partida de dos jugadores para los colores rojo y azul; estas son dos formas distintas de emplear el método empezar, que es polimórfico.

Herencia

Una clase puede heredar propiedades de otra.

Introducción al concepto de objeto

Un objeto es cualquier entidad representable en un programa informático, bien sea real (ordenador) o bien sea un concepto (transferencia). Un objeto en un sistema posee: una identidad, un estado y un comportamiento.

El estado marca las condiciones de existencia del objeto dentro del programa. Lógicamente este estado puede cambiar. Un coche puede estar parado, en marcha, estropeado, funcionando, sin gasolina, etc.

El comportamiento determina como responde el objeto ante peticiones de otros objetos. Por ejemplo un objeto conductor puede lanzar el mensaje arrancar a un coche. El comportamiento determina qué es lo que hará el objeto.

La identidad determina que cada objeto es único aunque tengan el mismo valor. No existen dos objetos iguales. Lo que sí existe es dos referencias al mismo objeto.

Los objetos se manejan por referencias, existirá una referencia a un objeto. De modo que esa referencia permitirá cambiar los atributos del objeto. Incluso puede haber varias referencias al mismo objeto, de modo que si una referencia cambia el estado del objeto, el resto (lógicamente) mostrarán esos cambios.

Los objetos por valor son los que no usan referencias y usan copias de valores concretos. En Java estos objetos son los tipos simples: int, char, byte, short, long, float, double y boolean. El resto son todos objetos (incluidos los arrays y Strings).

Clases

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc., etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearía tantos objetos casilla como casillas tenga el juego.

Lo mismo ocurriría con las fichas, la clase ficha definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc., etc.,), luego se crearían tantos objetos ficha, como fichas tenga el juego.

Las clases son las plantillas para hacer objetos. Una clase sirve para definir una serie de objetos con propiedades (atributos), comportamientos (operaciones o métodos), y semántica comunes. Hay que pensar en una clase como un molde. A través de las clases se obtienen los objetos en sí.

Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

Sus atributos. Es decir, los datos miembros de esa clase. Los datos pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase. También se las llama campos.

Sus métodos. Las funciones miembro de la clase. Son las acciones (u operaciones) que puede realizar la clase.

Código de inicialización. Para crear una clase normalmente hace falta realizar operaciones previas (es lo que se conoce como el constructor de la clase).

Otras clases. Dentro de una clase se pueden definir otras clases (clases internas, son consideradas como asociaciones dentro de UML).

Nombre de clase

Atributos

Métodos

Ilustración 5, Clase en notación UML

El formato general para crear una clase en Java es:

```
[acceso] class nombreDeClase {
  [acceso] [static] tipo atributo1;
  [acceso] [static] tipo atributo2;
```

```
    [acceso] [static] tipo atributo3;
    ...
    [access] [static] tipo método1(listaDeArgumentos) {
        ...código del método...
    }
    ...
}
```

La palabra opcional static sirve para hacer que el método o la propiedad a la que precede se pueda utilizar de manera genérica (más adelante se hablará de clases genéricas), los métodos o propiedades así definidos se llaman atributos de clase y métodos de clase respectivamente. Su uso se verá más adelante. Ejemplo;

```
class Noria {
    double radio;
    void girar(int velocidad){
        ...//definición del método
    }
    void parar(){...
}
```

Noria
radio:double
parar()
girar(int)

Objetos

Se les llama instancias de clase. Son un elemento en sí de la clase (en el ejemplo del parchís, una ficha en concreto). Un objeto se crea utilizando el llamado constructor de la clase. El constructor es el método que permite iniciar el objeto.

datos miembro (propiedades o atributos)

Para poder acceder a los atributos de un objeto, se utiliza esta sintaxis:

objeto.atributo

Por ejemplo:

Noria.radio;

Métodos

Los métodos se utilizan de la misma forma que los atributos, excepto porque los métodos poseen siempre paréntesis, dentro de los cuales pueden ir valore snecesarios para la ejecución del método (parámetros):

objeto.método(argumentosDelMétodo)

Los métodos siempre tienen paréntesis (es la diferencia con las propiedades) y dentro de los paréntesis se colocan los argumentos del método. Que son los datos que necesita el método para funcionar. Por ejemplo:

MiNoria.gira(5);

Lo cual podría hacer que la Noria avance a 5 Km/h.

Herencia

En la POO tiene mucha importancia este concepto, la herencia es el mecanismo que permite crear clases basadas en otras existentes. Se dice que esas clases descienden de las primeras. Así por ejemplo, se podría crear una clase llamada vehículo cuyos métodos serían mover, parar, acelerar y frenar. Y después se podría crear una clase coche basada en la anterior que tendría esos mismos métodos (les heredaría) y además añadiría algunos propios, por ejemplo abrirCapó o cambiarRueda.

Creación de objetos de la clase

Una vez definida la clase, se pueden utilizar objetos de la clase. Normalmente consta de dos pasos. Su declaración, y su creación. La declaración consiste en indicar que se va a utilizar un objeto de una clase determinada. Y se hace igual que cuando se declara una variable simple. Por ejemplo:

Noria noriaDePalencia;

Eso declara el objeto noriaDePalencia como objeto de tipo Noria; se supone que previamente se ha definido la clase Noria.

Para poder utilizar un objeto, hay que crearle de verdad. Eso consiste en utilizar el operador new. Por ejemplo:

noriaDePalencia = new Noria();

Al hacer esta operación el objeto reserva la memoria que necesita y se inicializa el objeto

mediante su constructor. Más adelante veremos como definir el constructor.

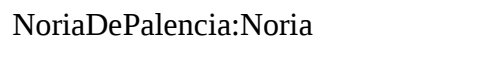


Ilustración 7, Objeto NoriaDePalencia de la clase Noria en notación UML

Especificadores de acceso

Se trata de una palabra que antecede a la declaración de una clase, método o propiedad de clase. Hay tres posibilidades: public, protected y private. Una cuarta posibilidad es no utilizar ninguna de estas tres palabras; entonces se dice que se ha utilizado el modificador por defecto (friendly).

Los especificadores determinan el alcance de la visibilidad del elemento al que se refieren. Referidos por ejemplo a un método, pueden hacer que el método sea visible sólo para la clase que lo utiliza (private), para éstas y las heredadas (protected), para todas las clases del mismo paquete (friendly) o para cualquier clase del tipo que sea (public).

En la siguiente tabla se puede observar la visibilidad de cada especificador:

zona	private (privado)	sin modificador (friendly)	protected (protegido)	public (público)
Misma clase	X	X	X	X
Subclase en el mismo paquete		X	X	X
Clase (no subclase) en el mismo paquete		X		X
Subclase en otro paquete			X	X
No subclase en otro paquete				X

Definir atributos de la clase (variables, propiedades o datos de la clases)

Cuando se definen los datos de una determinada clase, se debe indicar el tipo de propiedad que es (String, int, double, int[][],...) y el especificador de acceso (public, private,...). El especificador indica en qué partes del código ese dato será visible. Ejemplo:

```
class Persona {
  public String nombre;//Se puede acceder desde cualquier clase
  private int contraseña;//Sólo se puede acceder desde la
                                  //clase Persona
  protected String dirección; //Acceden a esta propiedad
                                  //esta clase y sus descendientes
```

Por lo general las propiedades de una clase suelen ser privadas o protegidas, a no ser que se trate de un valor constante, en cuyo caso se declararán como públicos.

Las variables locales de una clase pueden ser inicializadas.

```
class auto{
  public nRuedas=4;
```

Persona
+nombre:String
-contraseña:String
#direccion:String

Ilustración 8, La clase persona en UML. El signo + significa public, el signo # protected y el signo - private

Definir métodos de clase (operaciones o funciones de clase)

Un método es una llamada a una operación de un determinado objeto. Al realizar esta llamada (también se le llama enviar un mensaje), el control del programa pasa a ese método y lo mantendrá hasta que el método finalice o se haga uso de return.

Para que un método pueda trabajar, normalmente hay que pasarle unos datos en forma de argumentos o parámetros, cada uno de los cuales se separa por comas. Ejemplos de llamadas:

```
balón.botar(); //sin argumentos
miCoche.acelerar(10);
```

```
ficha.comer(posición15);posición 15 es una variable que se
                    //pasa como argumento
partida.empezarPartida("18:15",colores);
```

Los métodos de la clase se definen dentro de ésta. Hay que indicar un modificador de acceso (public, private, protected o ninguno, al igual que ocurre con las variables y con la propia clase) y un tipo de datos, que indica qué tipo de valores devuelve el método.

Esto último se debe a que los métodos son funciones que pueden devolver un determinado valor (un entero, un texto, un valor lógico,...) mediante el comando return. Si el método no devuelve ningún valor, entonces se utiliza el tipo void que significa que no devuelve valores (en ese caso el método no tendrá instrucción return).

El último detalle a tener en cuenta es que los métodos casi siempre necesitan datos para realizar la operación, estos datos van entre paréntesis y se les llama argumentos. Al definir el método hay que indicar que argumentos se necesitan y de qué tipo son.

Ejemplo:

```
public class vehiculo {
      /** Función principal */
      int ruedas;
      private double velocidad=0;
      String nombre;
      /** Aumenta la velocidad*/
        public void acelerar(double cantidad) {
            velocidad += cantidad;
      }
      /** Disminuye la velocidad*/
        public void frenar(double cantidad) {
            velocidad -= cantidad;
      }
      /** Devuelve la velocidad*/
      public double obtenerVelocidad(){
            return velocidad;
        }
public static void main(String args[]){
          vehiculo miCoche = new vehiculo();
```

```
        miCoche.acelerar(12);
        miCoche.frenar(5);
        System.out.println(miCoche.obtenerVelocidad());
    } // Da 7.0
```

En la clase anterior, los métodos acelerar y frenar son de tipo void por eso no tienen sentencia return. Sin embargo el método obtenerVelocidad es de tipo double por lo que su resultado es devuelto por la sentencia return y puede ser escrito en pantalla.

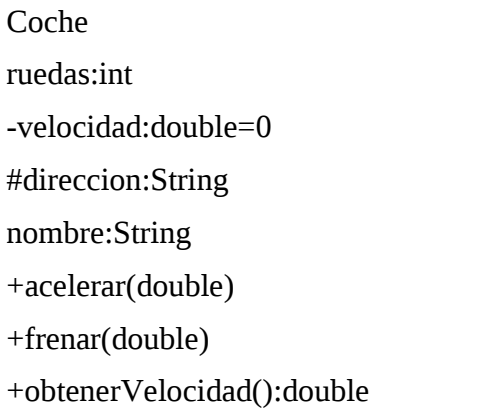


Ilustración 9, Versión UML de la clase Coche

Argumentos por valor y por referencia

En todos los lenguajes éste es un tema muy importante. Los argumentos son los datos que recibe un método y que necesita para funcionar. Ejemplo:

```
public class Matemáticas {
  public double factorial(int n){
        double resultado;
        for (resultado=n;n>1;n--) resultado*=n;
        return resultado;
  }
  ...
public static void main(String args[]){
  Matemáticas m1=new Matemáticas();
  double x=m1.factorial(25);//Llamada al método
  }
```

En el ejemplo anterior, el valor 25 es un argumento requerido por el método factorial

para que éste devuelva el resultado (que será el factorial de 25). En el código del método factorial, este valor 25 es copiado a la variable n, que es la encargada de almacenar y utilizar este valor.

Se dice que los argumentos son por valor, si la función recibe una copia de esos datos, es decir la variable que se pasa como argumento no estará afectada por el código.

Ejemplo:

```
class prueba {
   public void metodo1(int entero){
         entero=18;
...
   }
...
public static void main(String args[]){
         int x=24;
        prueba miPrueba = new prueba();
         miPrueba.metodo1(x);
         System.out.println(x); //Escribe 24, no 18
}
```

Este es un ejemplo de paso de parámetros por valor. La variable x se pasa como argumento o parámetro para el método metodo1, allí la variable entero recibe una copia del valor de x en la variable entero, y a esa copia se le asigna el valor 18. Sin embargo la variable x no está afectada por esta asignación.

Sin embargo en este otro caso:

```
class prueba {
   public void metodo1(int[] entero){
         entero[0]=18;
...
   }
...
public static void main(String args[]){
         int x[]={24,24};
        prueba miPrueba = new prueba();
```

```
        miPrueba.metodo1(x);
        System.out.println(x[0]); //Escribe 18, no 24
```

Aquí sí que la variable x está afectada por la asignación entero[0]=18. La razón es porque en este caso el método no recibe el valor de esta variable, sino la referencia, es decir la dirección física de esta variable. entero no es una replica de x, es la propia x llamada de otra forma.

Los tipos básicos (int, double, char, boolean, float, short y byte) se pasan por valor. También se pasan por valor las variables String. Los objetos y arrays se pasan por referencia.

Devolución de valores

Los métodos pueden devolver valores básicos (int, short, double, etc.), Strings, arrays e incluso objetos.

En todos los casos es el comando return el que realiza esta labor. En el caso de arrays y objetos, devuelve una referencia a ese array u objeto. Ejemplo:

```
class FabricaArrays {
    public int[] obtenArray(){
        int array[]= {1,2,3,4,5};
        return array;
    }
}
public class returnArray {
    public static void main(String[] args) {
        FabricaArrays fab=new FabricaArrays();
        int nuevoArray[]=fab.obtenArray();
    }
}
```

Sobrecarga de métodos

Una propiedad de la POO es el polimorfismo. Java posee esa propiedad ya que admite sobrecargar los métodos. Esto significa crear distintas variantes del mismo método.

Ejemplo:

```
class Matemáticas{
  public double suma(double x, double y) {
      return x+y;
  }
  public double suma(double x, double y, double z){
      return x+y+z;
  }
  public double suma(double[] array){
      double total =0;
      for(int i=0; i<array.length;i++){
          total+=array[i];
      }
      return total;
  }
```

La clase matemáticas posee tres versiones del método suma. una versión que suma dos números double, otra que suma tres y la última que suma todos los miembros de un array de doubles. Desde el código se puede utilizar cualquiera de las tres versiones según convenga.

La referencia this

La palabra this es una referencia al propio objeto en el que estamos. Ejemplo:

```
class punto {
  int posX, posY;//posición del punto
  punto(posX, posY){
      this.posX=posX;
      this.posY=posY;
  }
```

En el ejemplo hace falta la referencia this para clarificar cuando se usan las propiedades posX y posY, y cuando los argumentos con el mismo nombre. Otro ejemplo:

```
class punto {
  int posX, posY;
  ...
```

```
/**Suma las      coordenadas de otro punto*/
public void      suma(punto punto2){
      posX =    punto2.posX;
      posY =    punto2.posY;
}
/** Dobla el valor de las coordenadas del punto*/
      public void dobla(){
            suma(this);
      }
```

En el ejemplo anterior, la función dobla, dobla el valor de las coordenadas pasando el propio punto como referencia para la función suma (un punto sumado a sí mismo, daría el doble).

Los posibles usos de this son:

this. Referencia al objeto actual. Se usa por ejemplo pasarle como parámetro a un método cuando es llamado desde la propia clase.

this.atributo. Para acceder a una propiedad del objeto actual.

this.método(parámetros). Permite llamar a un método del objeto actual con los parámetros indicados.

this(parámetros). Permite llamar a un constructor del objeto actual. Esta llamada sólo puede ser empleada en la primera línea de un constructor.

Creación de constructores

Un constructor es un método que es llamado automáticamente al crear un objeto de una clase, es decir al usar la instrucción new. Sin embargo en ninguno de los ejemplos anteriores se ha definido constructor alguno, por eso no se ha utilizado ningún constructor al crear el objeto.

Un constructor no es más que un método que tiene el mismo nombre que la clase. Con lo cual para crear un constructor basta definir un método en el código de la clase que tenga el mismo nombre que la clase. Ejemplo:

```
class Ficha {
  private int casilla;
  Ficha() { //constructor
        casilla = 1;
```

```
  }
  public void avanzar(int n) {
        casilla += n;
  }
  public int casillaActual(){
        return casilla;
  }
}
public class app {
  public static void main(String[] args) {
        Ficha ficha1 = new Ficha();
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 4
```

En la línea Ficha ficha1 = new Ficha(); es cuando se llama al constructor, que es el que coloca inicialmente la casilla a 1. Pero el constructor puede tener parámetros:

```
class Ficha {
  private int casilla; //Valor inicial de la propiedad
  Ficha(int n) { //constructor
        casilla = n;
  }
  public void avanzar(int n) {
        casilla += n;
  }
  public int casillaActual(){
        return casilla;
  }
}

public class app {
  public static void main(String[] args) {
        Ficha ficha1 = new Ficha(6);
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 9
```

En este otro ejemplo, al crear el objeto ficha1, se le da un valor a la casilla, por lo que la

casilla vale al principio 6.

Hay que tener en cuenta que puede haber más de un constructor para la misma clase. Al igual que ocurría con los métodos, los constructores se pueden sobrecargar.

De este modo en el código anterior de la clase Ficha se podrían haber colocado los dos constructores que hemos visto, y sería entonces posible este código:

```
Ficha ficha1= new Ficha(); //La propiedad casilla de la
                           //ficha valdrá 1
Ficha ficha1= new Ficha(6); //La propiedad casilla de la
                           //ficha valdrá 6
```

Cuando se sobrecargan los constructores (se utilizan varias posibilidades de constructor), se pueden hacer llamadas a constructores mediante el objeto this

métodos y propiedades genéricos (static)

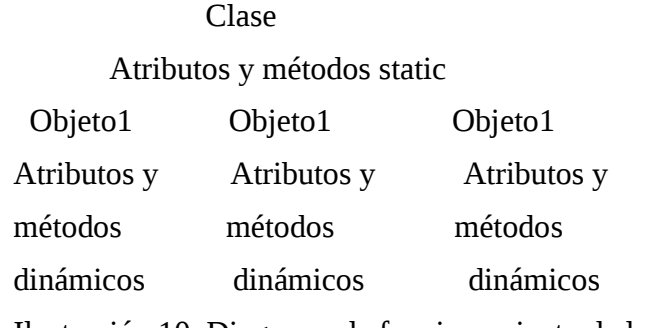


Ilustración 10, Diagrama de funcionamiento de los métodos y atributos static

Hemos visto que hay que crear objetos para poder utilizar los métodos y propiedades de una determinada clase. Sin embargo esto no es necesario si la propiedad o el método se definen precedidos de la palabra clave static. De esta forma se podrá utilizar el método sin definir objeto alguno, utilizando el nombre de la clase como si fuera un objeto. Así funciona la clase Math (véase la clase Math, página 23). Ejemplo:

```
class Calculadora {
  static public int factorial(int n) {
      int fact=1;
      while (n>0) {
          fact *=n--;
      }
      return fact;
```

```
  }
}
public class app {
  public static void main(String[] args) {
        System.out.println(Calculadora.factorial(5));
  }
}
```

En este ejemplo no ha hecho falta crear objeto alguno para poder calcular el factorial.

Una clase puede tener métodos y propiedades genéricos (static) y métodos y propiedades dinámicas (normales).

Cada vez que se crea un objeto con new, se almacena éste en memoria. Los métodos y propiedades normales, gastan memoria por cada objeto que se cree, sin embargo los métodos estáticos no gastan memoria por cada objeto creado, gastan memoria al definir la clase sólo. Es decir los métodos y atributos static son los mismos para todos los objetos creados, gastan por definir la clase, pero no por crear cada objeto.

Hay que crear métodos y propiedades genéricos cuando ese método o propiedad vale o da el mismo resultado en todos los objetos. Pero hay que utilizar métodos normales (dinámicos) cuando el método da resultados distintos según el objeto. Por ejemplo en un clase que represente aviones, la altura sería un atributo dinámico (distinto en cada objeto), mientras que el número total de aviones, sería un método static (es el mismo para todos los aviones).

El método main

Hasta ahora hemos utilizado el método main de forma incoherente como único posible mecanismo para ejecutar programas. De hecho este método dentro de una clase, indica que la clase es ejecutable desde la consola. Su prototipo es:

```
public static void main(String[] args){
  ...instruccionesejecutables....
}
```

Hay que tener en cuenta que el método main es estático, por lo que no podrá utilizar atributos o métodos dinámicos de la clase.

Los argumentos del método main son un array de caracteres donde cada elemento del array es un parámetro enviado por el usuario desde la línea de comandos. A este argumento se le llama comúnmente args. Es decir, si se ejecuta el programa con:

java claseConMain uno dos

Entonces el método main de esta clase recibe un array con dos elementos, el primero es la cadena “uno” y el segundo la cadena “dos” (es decir args[0]=”uno”; args[1]=”dos”).

destrucción de objetos

En C y C++ todos los programadores saben que los objetos se crean con new y para eliminarles de la memoria y así ahorrarla, se deben eliminar con la instrucción delete. Es decir, es responsabilidad del programador eliminar la memoria que gastaban los objetos que se van a dejar de usar. La instrucción delete del C++ llama al destructor de la clase, que es una función que se encarga de eliminar adecuadamente el objeto.

La sorpresa de los programadores C++ que empiezan a trabajar en Java es que no hay instrucción delete en Java. La duda está entonces, en cuándo se elimina la memoria que ocupa un objeto.

En Java hay un recolector de basura (garbage collector) que se encarga de gestionar los objetos que se dejan de usar y de eliminarles de memoria. Este proceso es automático e impredecible y trabajo en un hilo (thread) de baja prioridad.

Por lo general ese proceso de recolección de basura, trabaja cuando detecta que un objeto hace demasiado tiempo que no se utiliza en un programa. Esta eliminación depende de la máquina virtual, en casi todas la recolección se realiza periódicamente en un determinado lapso de tiempo. La implantación de máquina virtual conocida como HotSpot1 suele hacer la recolección mucho más a menudo

Se puede forzar la eliminación de un objeto asignándole el valor null, pero teniendo en cuenta que eso no equivale al famoso delete del lenguaje C++. Con null no se libera inmediatamente la memoria, sino que pasará un cierto tiempo (impredecible, por otro lado) hasta su total destrucción.

Se puede invocar al recolector de basura desde el código invocando al método estático System.gc(). Esto hace que el recolector de basura trabaje en cuanto se lea esa invocación.

Sin embargo puede haber problemas al crear referencias circulares. Como:

```
class uno {
  dos d;
  uno() { //constructor
        d = new dos();
  }
}
```

```
class dos {
  uno u;
  dos() {
        u = new uno();
  }
public class app {
```

Para saber más sobre HotSpot acudir a java.sun.com/products/hotspot/index.html.

```
   public static void main(Stgring[] args) {
        uno prueba = new uno();//referencia circular
        prueba = null; //no se liberará bien la memoria
   }
}
```

Al crear un objeto de clase uno, automáticamente se crea uno de la clase dos, que al crearse creará otro de la clase uno. Eso es un error que provocará que no se libere bien la memoria salvo que se eliminen previamente los objetos referenciados.

El método finalize

Es equivalente a los destructores del C++. Es un método que es llamado antes de eliminar definitivamente al objeto para hacer limpieza final. Un uso puede ser eliminar los objetos creados en la clase para eliminar referencias circulares. Ejemplo:

```
class uno {
  dos d;
  uno() {
        d = new dos();
  }
  protected void finalize(){
        d = null;//Se elimina d por lo que pudiera pasar
  }
}
```

finalize es un método de tipo protected heredado por todas las clases ya que está definido en la clase raíz Object.

La diferencia de finalize respecto a los métodos destructores de C++ estriba en que en Java no se llaman al instante (de hecho es imposible saber cuando son llamados). la llamada System.gc() llama a todos los finalize pendientes inmediatamente (es una forma de probar si el método finalize funciona o no).

Herencia

Introducción

Es una de las armas fundamentales de la programación orientada a objetos. Permite crear nuevas clases que heredan características presentas en clases anteriores. Esto facilita enormemente el trabajo porque ha permitido crear clases estándar para todos los programadores y a partir de ellas crear nuestras propias clases personales. Esto es más cómodo que tener que crear nuestras clases desde cero.

Para que una clase herede las características de otra hay que utilizar la palabra clave extends tras el nombre de la clase. A esta palabra le sigue el nombre de la clase cuyas características se heredarán. Sólo se puede tener herencia de una clase (a la clase de la que se hereda se la llama superclase y a la clase heredada se la llama subclase).
Ejemplo:

```
class coche extends vehiculo {
...
} //La clase coche parte de la definición de vehículo
                    superclase
                   vehículo
                   +ruedas:int;
                   +velocidad:double
                   +acelerar(int)
heredado
                   +frenar(int)
                            subclase
            coche
            +ruedas:int=4
redefinido
            +gasolina:int
   propio
            +repostar(int)
```

Ilustración 11, herencia

Métodos y propiedades no heredados

Por defecto se heredan todos los métodos y propiedades protected y public (no se heredan los private). Además si se define un método o propiedad en la subclase con el mismo nombre que en la superclase, entonces se dice que se está redefiniendo el método, con lo cual no se hereda éste, sino que se reemplaza por el nuevo.

Ejemplo:

```
class vehiculo {
  public int velocidad;
  public int ruedas;
  public void parar() {
        velocidad = 0;
  }
  public void acelerar(int kmh) {
        velocidad += kmh;
  }
class coche extends vehiculo{
  public int ruedas=4;
  public int gasolina;
  public void repostar(int litros) {
        gasolina+=litros;
  }
}
.....
public class app {
  public static void main(String[] args) {
        coche coche1=new coche();
        coche.acelerar(80);//Método heredado
        coche.repostar(12);
```

Anulación de métodos

Como se ha visto, las subclases heredan los métodos de las superclases. Pero es más, también los pueden sobrecargar para proporcionar una versión de un determinado método.

Por último, si una subclase define un método con el mismo nombre, tipo y argumentos que un método de la superclase, se dice entonces que se sobrescribe o anula el método de la superclase. Ejemplo:

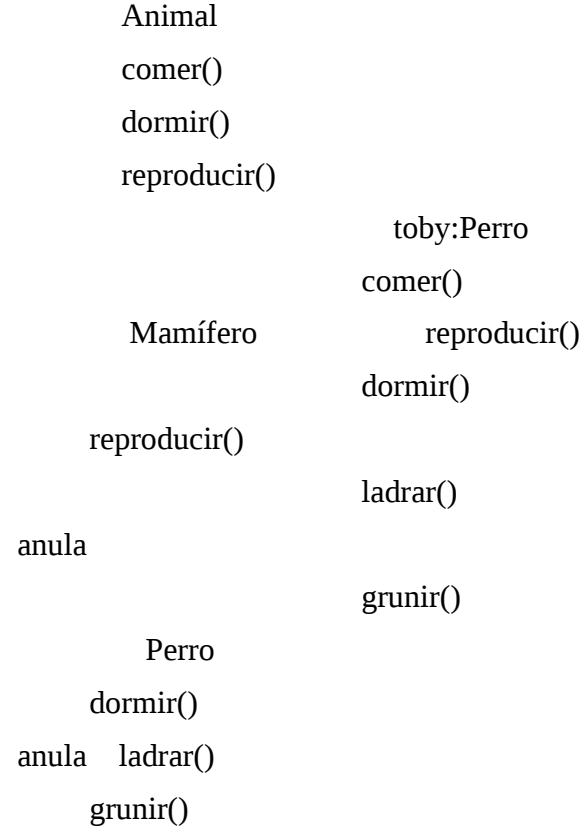


Ilustración 12, anulación de métodos

Super

A veces se requiere llamar a un método de la superclase. Eso se realiza con la palabra reservada super. Si this hace referencia a la clase actual, super hace referencia a la superclase respecto a la clase actual, con lo que es un método imprescindible para poder acceder a métodos anulados por herencia. Ejemplo

```
public class vehiculo{
  double velocidad;
```

```
    ...
    public void acelerar(double cantidad){
        velocidad+=cantidad;
    }
}
public class coche extends vehiculo{
    double gasolina;
    public void acelerar(double cantidad){
        super.acelerar(cantidad);
        gasolina*=0.9;
    }
```

En el ejemplo anterior, la llamada super.acelerar(cantidad) llama al método acelerar de la clase vehículo (el cual acelerará la marcha). Es necesario redefinir el método acelerar

en la clase coche ya que aunque la velocidad varía igual que en la superclase, hay que tener en cuenta el consumo de gasolina

Se puede incluso llamar a un constructor de una superclase, usando la sentencia super(). Ejemplo:

```
public class vehiculo{
    double velocidad;
    public vehiculo(double v){
        velocidad=v;
    }
}
public class coche extends vehiculo{
    double gasolina;
    public coche(double v, double g){
        super(v);//Llama al constructor de la clase vehiculo
        gasolina=g
    }
}
```

Por defecto Java realiza estas acciones:

Si la primera instrucción de un constructor de una subclase es una sentencia que no es ni super ni this, Java añade de forma invisible e implícita una llamada super() al constructor por defecto de la superclase, luego inicia las variables de la

subclase y luego sigue con la ejecución normal.
Si se usa super(..) en la primera instrucción, entonces se llama al constructor seleccionado de la superclase, luego inicia las propiedades de la subclase y luego sigue con el resto de sentencias del constructor.
Finalmente, si esa primera instrucción es this(..), entonces se llama al constructor seleccionado por medio de this, y después continúa con las sentencias del constructor. La inicialización de variables la habrá realizado el constructor al que se llamó mediante this.

Casting de clases

Como ocurre con los tipos básicos (ver conversión entre tipos (casting), página 18, es posible realizar un casting de objetos para convertir entre clases distintas. Lo que ocurre es que sólo se puede realizar este casting entre subclases. Es decir se realiza un casting para especificar más una referencia de clase (se realiza sobre una superclase para convertirla a una referencia de una subclase suya).

En cualquier otro caso no se puede asignar un objeto de un determinado tipo a otro.

Ejemplo:

```
Vehiculo vehiculo5=new Vehiculo();
Coche cocheDePepe = new Coche("BMW");
vehiculo5=cocheDePepe //Esto sí se permite
cocheDePepe=vehiculo5;//Tipos incompatibles
cocheDepepe=(coche)vehiculo5;//Ahora sí se permite
```

Hay que tener en cuenta que los objetos nunca cambian de tipo, se les prepara para su asignación pero no pueden acceder a propiedades o métodos que no les sean propios. Por ejemplo, si repostar() es un método de la clase coche y no de vehículo:

```
Vehiculo v1=new Vehiculo();
Coche c=new Coche();
v1=c;//No hace falta casting
v1.repostar(5);//¡¡¡Error!!!
```

Cuando se fuerza a realizar un casting entre objetos, en caso de que no se pueda realizar ocurrirá una excepción del tipo ClassCastingException. Realmente sólo se puede hacer un casting si el objeto originalmente era de ese tipo. Es decir la instrucción:

cocheDepepe=(Coche) vehiculo4;

Sólo es posible si vehiculo4 hace referencia a un objeto coche.

Instanceof

Permite comprobar si un determinado objeto pertenece a una clase concreta. Se utiliza de esta forma:

objeto instanceof clase

Comprueba si el objeto pertenece a una determinada clase y devuelve un valor true si es así. Ejemplo:

```
Coche miMercedes=new Coche();
if (miMercedes instanceof Coche)
  System.out.println("ES un coche");
if (miMercedes instanceof Vehículo)
  System.out.println("ES un coche");
if (miMercedes instanceof Camión)
  System.out.println("ES un camión");
```

En el ejemplo anterior aparecerá en pantalla:

ES un coche

ES un vehiculo

Clases abstractas

A veces resulta que en las superclases se desean incluir métodos teóricos, métodos que no se desea implementar del todo, sino que sencillamente se indican en la clase para que el desarrollador que desee crear una subclase heredada de la clase abstracta, esté obligado a sobrescribir el método.

A las clases que poseen métodos de este tipo (métodos abstractos) se las llama clases abstractas. Son clases creadas para ser heredadas por nuevas clases creadas por el programador. Son clases base para herencia. Las clases abstractas no deben de ser instanciadas (no se pueden crear objetos de las clases abstractas).

Una clase abstracta debe ser marcada con la palabra clave abstract. Cada método

abstracto de la clase, también llevará el abstract. Ejemplo:

```
abstract class vehiculo {
  public int velocidad=0;
  abstract public void acelera();
  public void para() {velocidad=0;}
}
class coche extends vehiculo {
  public void acelera() {
        velocidad+=5;
  }
}
public class prueba {
  public static void main(String[] args) {
        coche c1=new coche();
        c1.acelera();
        System.out.println(c1.velocidad);
        c1.para();
        System.out.println(c1.velocidad);
  }
}
```

Final

Se trata de una palabra que se coloca antecediendo a un método, variable o clase. Delante de un método en la definición de clase sirve para indicar que ese método no puede ser sobrescrito por las subclases. Si una subclase intentar sobrescribir el método, el compilador de Java avisará del error.

Si esa misma palabra se coloca delante de una clase, significará que esa clase no puede tener descendencia.

Por último si se usa la palabra final delante de la definición de una propiedad de clase, entonces esa propiedad pasará a ser una constante, es decir no se le podrá cambiar el valor en ninguna parte del código.

Clases internas

Se llaman clases internas a las clases que se definen dentro de otra clase. Esto permite simplificar aun más el problema de crear programas. Ya que un objeto complejo se puede descomponer en clases más sencillas. Pero requiere esta técnica una mayor pericia por parte del programador.

Al definir una clase dentro de otra, estamos haciéndola totalmente dependiente. Normalmente se realiza esta práctica para crear objetos internos a una clase (el motor de un coche por ejemplo), de modo que esos objetos pasan a ser atributos de la clase.

Por ejemplo:

```
public class Coche {
    public int velocidad;
    public Motor motor;
    public Coche(int cil) {
        motor=new Motor(cil);
        velocidad=0;
    }
    public class Motor{ //Clase interna
        public int cilindrada;
        public Motor(int cil){
            cilindrada=cil;
        }
    }
}
```

El objeto motor es un objeto de la clase Motor que es interna a Coche. Si quisiéramos acceder al objeto motor de un coche sería:

```
Coche c=new Coche(1200);
System.out.println(c.motor.cilindrada);//Saldrá 1200
```

Las clases internas pueden ser privadas, protegidas o públicas. Fuera de la clase contenedora no pueden crear objetos (sólo se pueden crear motores dentro de un coche), salvo que la clase interna sea static en ese caso sí podrían. Por ejemplo (si la clase motor fuera estática):

```
//suponiendo que la declaración del Motor dentro de Coche es
// public class static Motor{....
```

Coche.Motor m=new Coche.Motor(1200);

Pero eso sólo tiene sentido si todos los Coches tuvieran el mismo motor.

Dejando de lado el tema de las clases static, otro problema está en el operador this. El problema es que al usar this dentro de una clase interna, this se refiere al objeto de la clase interna (es decir this dentro de Motor se refiere al objeto Motor). Para poder referirse al objeto contenedor (al coche) se usa Clase.this (Coche.this). Ejemplo:

```
public class Coche {
    public int velocidad;
    public int cilindrada;
    public Motor motor;
    public Coche(int cil) {
        motor=new Motor(cil);
        velocidad=0;
    }
    public class Motor{
        public int cilindrada;
        public Motor(int cil){
            Coche.this.cilindrada=cil;//Coche
             this.cilindrada=cil;//Motor
        }
    }
}
```

Por último las clases internas pueden ser anónimas (se verán más adelante al estar más relacionadas con interfaces y adaptadores).

Creación de paquetes

Un paquete es una colección de clases e interfaces relacionadas. El compilador de Java usa los paquetes para organizar la compilación y ejecución. Es decir, un paquete es una biblioteca. De hecho el nombre completo de una clase es el nombre del paquete en el que está la clase, punto y luego el nombre de la clase. Es decir si la clase Coche está dentro del paquete locomoción, el nombre completo de Coche es locomoción.Coche.

A veces resulta que un paquete está dentro de otro paquete, entonces habrá que indicar la ruta completa a la clase. Por ejemplo locomoción.motor.Coche

Mediante el comando import (visto anteriormente), se evita tener que colocar el

nombre completo. El comando import se coloca antes de definir la clase. Ejemplo:

```
import locomoción.motor.Coche;
```

Gracias a esta instrucción para utilizar la clase Coche no hace falta indicar el paquete en el que se encuentra, basta indicar sólo Coche. Se puede utilizar el símbolo asterisco como comodín.

Ejemplo:

```
import locomoción.*;
//Importa todas las clase del paquete locomoción
```

Esta instrucción no importa el contenido de los paquetes interiores a locomoción (es decir que si la clase Coche está dentro del paquete motor, no sería importada con esa instrucción, ya que el paquete motor no ha sido importado, sí lo sería la clase locomoción.BarcoDeVela). Por ello en el ejemplo lo completo sería:

```
import locomoción.*;
import locomoción.motor.*;
```

Cuando desde un programa se hace referencia a una determinada clase se busca ésta en el paquete en el que está colocada la clase y, sino se encuentra, en los paquetes que se han importado al programa. Si ese nombre de clase se ha definido en un solo paquete, se usa. Si no es así podría haber ambigüedad por ello se debe usar un prefijo delante de la clase con el nombre del paquete.

Es decir:

```
paquete.clase
```

O incluso:

```
paquete1.paquete2.....clase
```

En el caso de que el paquete sea subpaquete de otro más grande.

Las clases son visibles en el mismo paquete a no ser que se las haya declarado con el modificador private.

Organización de los paquetes

Los paquetes en realidad son subdirectorios cuyo raíz debe ser absolutamente accesible por el sistema operativo. Para ello es necesario usar la variable de entorno CLASSPATH de la línea de comandos. Esta variable se suele definir en el archivo autoexec.bat o en MI PC en el caso de las últimas versiones de Windows (Véase proceso de compilación, página 9). Hay que añadirla las rutas a las carpetas que

contienen los paquetes (normalmente todos los paquetes se suelen crear en la misma carpeta), a estas carpetas se las llama filesystems.

Así para el paquete prueba.reloj tiene que haber una carpeta prueba, dentro de la cual habrá una carpeta reloj y esa carpeta prueba tiene que formar parte del classpath.

Una clase se declara perteneciente aun determinado paquete usando la instrucción package al principio del código (sin usar esta instrucción, la clase no se puede compilar). Si se usa package tiene que ser la primera instrucción del programa Java:

```
//Clase perteneciente al paquete tema5 que está en ejemplos
package ejemplos.tema5;
```

En los entornos de desarrollo o IDEs (NetBeans, JBuilder,...) se puede uno despreocupar de la variable classpath ya que poseen mecanismos de ayuda para gestionar los paquetes. Pero hay que tener en cuenta que si se compila a mano mediante el comando java (véase proceso de compilación, página i) se debe añadir el modificador -cp para que sean accesibles las clases contenidas en paquetes del classpath (por ejemplo java -cp prueba.java).

El uso de los paquetes permite que al compilar sólo se compile el código de la clase y de las clases importadas, en lugar de compilar todas las librerías. Sólo se compila lo que se utiliza.