PRÓLOGO

El intento de trasladar al lector los conceptos y mecanismos fundamentales de la Programación Orientada a Objetos, ha sido una constante en toda la redacción del libro, es más, el principal objetivo planteado. Siempre enfocado a facilitar al máximo su asimilación por parte del lector, a pesar de la complejidad inherente a los contenidos tratados. Se han utilizado, para ello, ejercicios ejemplo, especialmente diseñados para facilitar la comprensión de dichos aspectos fundamentales.

Java ha sido el lenguaje escogido como medio en la consecución de los fines planteados porque es el máximo exponente de la Orientación a Objetos, tanto por su purismo, como por la trayectoria y expansión que ha tenido, tiene, y tendrá.

En todos los temas, se ha intentado presentar al principio ejemplos lo más breves y sencillos posibles, a efectos de facilitar al lector la comprensión de los mecanismos básicos a tratar en cada momento. Así mismo, también se han presentado aplicaciones de mayor complejidad, en que se integran dichos mecanismos básicos. Se pretende que, con dicha integración en aplicaciones de mayor extensión, el lector tome una perspectiva global de dichos mecanismos.

Asociados a la mayor parte de ejercicios ejemplo, se aportan esquemas y gráficos, en un intento de clarificar al lector las referencias y objetos intervinientes, y también el cómo se transfieren dichas referencias entre las diferentes capas de la aplicación, cuando ello tiene lugar.

Se ha intentado, en la medida de lo posible, secuenciar contenidos, de tal modo que todos los conceptos, cuestiones, estrategias, etc. aplicados en un tema hubiesen sido ya tratados en temas anteriores.

En pro de minimizar la extensión de las líneas de código de las aplicaciones aportadas, se han producido dos tipos de actuaciones:

1. Los comentarios en el código de las clases están reducidos a la mínima expresión, incluso en la mayoría de los casos, son inexistentes. Así mismo, también se han reducido otros recursos que proporcionan legibilidad al programa, tales como líneas en blanco adicionales.

2. En ejercicios sencillos, se ha concentrado en el método main la mayor parte de actuaciones posibles, sin perder de vista que ello podría suponer alejarse del concepto de modularidad.

En el primer caso, la ausencia de comentarios en el código, es suplida por las debidas explicaciones en el tema relacionadas con dicho código. Y en el segundo caso se actúa "monolíticamente" para que el lector tenga concentradas, en el método en cuestión, la mayor parte de actuaciones posibles.

Por otra parte, también hemos de señalar que, en algunas aplicaciones ejemplo, sobre todo, en los últimos temas del libro, puede tener la impresión el lector, de que, algunas de ellas se podrían haber resuelto mediante una implementación más sencilla. Dicho "suplemento de complejidad" obedece a la intención didáctica de introducir al lector en el modelo de desarrollo software arquitectura a tres capas. El marco de organización que aporta dicho modelo se rentabiliza en aplicaciones de gran extensión. Así mismo, la aplicación de dicho marco de trabajo en estas aplicaciones, permitirá también al lector, reforzar conceptos y mecanismos tratados en temas anteriores.

Finalmente, tan solo matizar que, es el máximo deseo del autor que el lector llegue a asimilar los principios y mecanismos fundamentales de la Programación Orientada a Objetos, que como ya se ha mencionado, es el objetivo principal de este libro.

1

TIPOS DE DATOS. OPERADORES. EXPRESIONES

1.1 ESTRUCTURA DE UNA APLICACIÓN JAVA. CONTEXTO DE TRABAJO

Cuando nos enfrentamos a un nuevo lenguaje, la primera inquietud que se nos plantea es: ¿cuál es la mínima expresión de código en este nuevo lenguaje que me permita lanzar mi primera ejecución?

A continuación, se expone la mínima expresión de código Java susceptible de ser ejecutada:

```
public class MiPrimeraAplicacion {

    public static void main(String[] args) {
        System.out.println(“Hola”);
    }
}
```

El resultado de la ejecución, como se podrá comprobar, es la salida (visualización) por la consola establecida por el IDE, o el contexto de intérprete de comandos, que estemos utilizando de la cadena:

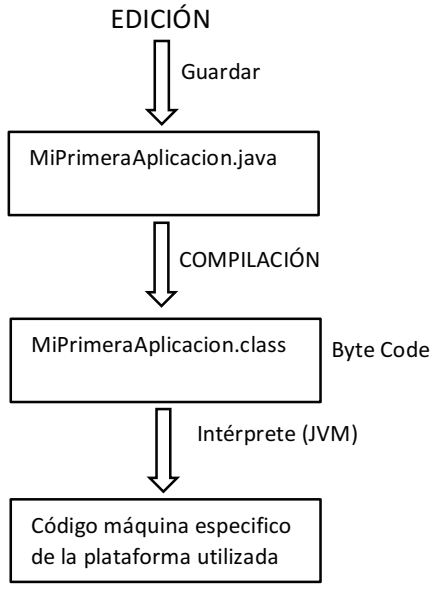
Hola

Dicha visualización es consecuencia de la ejecución del método *println( )*.

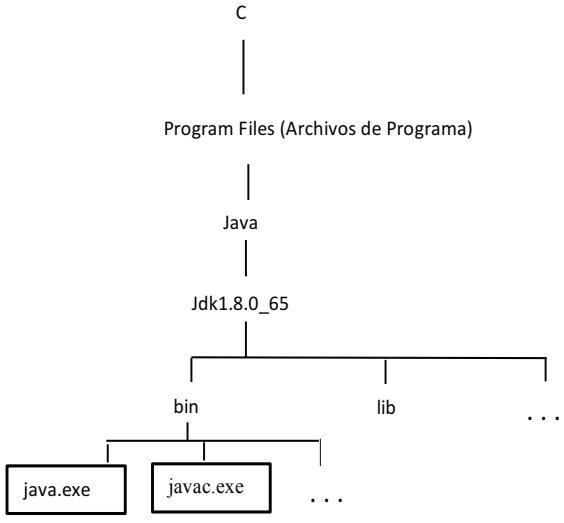
Antes de entrar en más consideraciones y profundidad, procede proporcionar al lector los medios para probar la ejecución de este primer programa. Recomendamos al lector, para ello, la utilización de algún entorno de desarrollo integrado (IDE) de aplicaciones Java. Se trata de software que ofrece al desarrollador una serie de herramientas que proporcionan mayor comodidad y rapidez, entre otras, tareas básicas y fundamentales como son la edición del código, compilación, y ejecución. Las más utilizadas son NetBeans, y Eclipse; pero hay otras más. Una de las formas de obtener los ejecutables para la instalación de un IDE, es descargárselo de Internet. Pero con total independencia de que el lector siga las recomendaciones de utilizar un IDE, vamos a proporcionar conceptos sobre los procesos fundamentales relacionados con la compilación y ejecución de aplicaciones Java.

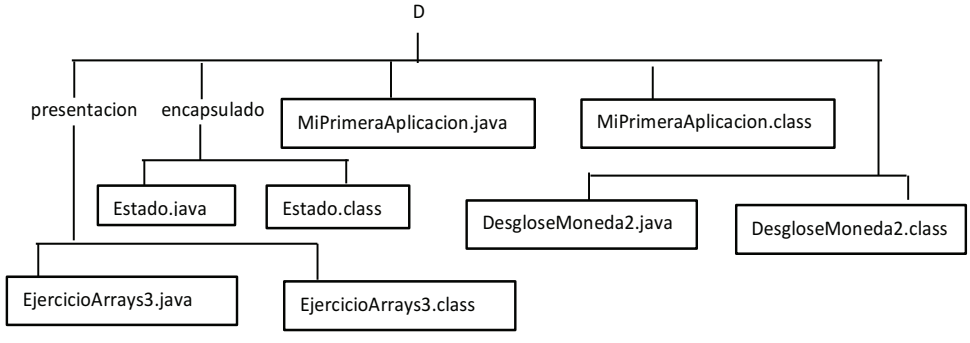
En principio, hemos de mencionar que, independientemente de que se utilice un IDE, o no se utilice, debe procederse, con anterioridad a la instalación del "sustrato de herramientas básico", el JDK (Java Development Kit). Análogamente a los IDE, una de las posibilidades de obtener el ejecutable para la instalación del JDK, es descargárselo desde Internet. Dependiendo del IDE, el JDK puede ir incorporado al software de instalación del IDE, o por separado. Caso de que JDK, e IDE se presenten en ejecutables de instalación por separado. Es necesario instalar primero el JDK. La instalación de ambos suele conllevar un procedimiento muy sencillo, basado en el seguimiento de un asistente, de tal modo, que asumiendo los parámetros que nos sugiere por defecto, la instalación es exitosa en la mayoría de las ocasiones. Solamente con el JDK, ya podemos compilar, y ejecutar aplicaciones Java desde el intérprete de comandos del sistema, o incluso crear iconos en el escritorio para proceder a la invocación de aplicaciones Java. El procedimiento para proceder a lo que estamos describiendo se detallará a continuación de la descripción de los procesos asociados al JDK. Una vez instalado el JDK, ya podemos proceder a la instalación del IDE, que se "acopla" a un JDK previamente instalado, que hay que identificar (normalmente, mediante la ruta del sistema de archivos donde está instalado), durante el proceso de instalación.

A diferencia de otros lenguajes, Java, es a la vez, un lenguaje compilado, e interpretado, es decir, que se requiere de ambos procesos. Para mejor comprensión del lector presentamos el siguiente esquema, con todos los pasos inherentes al proceso de edición, compilación, y ejecución de un programa Java, en que dicho proceso lo aplicamos al programa que hemos presentado **MiPrimeraAplicacion**:



En primer lugar, como en cualquier otro lenguaje, procede la utilización de un editor de textos para generar el programa fuente, que, al guardarlo en el sistema de archivos, debe tener, como nombre, el mismo de la clase, y extensión .java, con lo que en el caso de nuestro ejemplo, será **MiPrimeraAplicacion.java**. Dicho fuente, ya podemos someterlo a compilación, generando un fichero con el mismo nombre de la clase, pero extensión .class, en nuestro caso, MiPrimeraAplicación.class. Al contenido de este archivo se le llama Byte Code, que a diferencia de muchos lenguajes de programación, no supone un código máquina específico para una plataforma concreta y específica, sino que es un código máquina para una máquina cuya existencia es solamente virtual. Para poder ser ejecutado, el Byte Code, es necesario someterlo a un proceso de interpretación a la plataforma concreta a que se vaya a destinar. Este proceso corre a cargo de la JVM (Java Virtual Machine), lo cual nos hace llegar a la conclusión de que existe una JVM diferente para cada plataforma. Mostraremos en detalle a continuación, todos los pasos a seguir para, partiendo del fichero fuente, llegar a ejecutar el programa en una plataforma Windows. En el siguiente esquema exponemos los subárboles de directorios implicados en todo ello en la máquina Windows que se ha utilizado para probar el ejemplo, tanto del despliegue ocasionado en el sistema de archivos por la instalación del JDK, como de la ubicación de los fuentes, y los Byte Code, utilizados en este primer ejemplo, y en otro ejemplo en que se ven involucradas dos clases, organizadas en packages.





En el sistema de archivos del ordenador en que se han desarrollado los ejemplos que se incluyen en este libro, nos encontramos con el subárbol

```
C:\Program Files\Java
```

Dicho subárbol responde al despliegue obtenido como consecuencia de la instalación del JDK. En el directorio

```
C:\Program Files\Java\jdk1.8.0_65\bin
```

se ubican los ejecutables asociados a las utilidades del JDK, de los que los más importantes son:

- **javac.exe .-** El compilador Java

  código fuente → Byte Code

- **java.exe .-** El intérprete Java

  Byte Code → código máquina específico para la plataforma utilizada

Para trabajar con las utilidades del JDK, es necesario configurar el entorno, y dicha configuración hay que acometerla en:

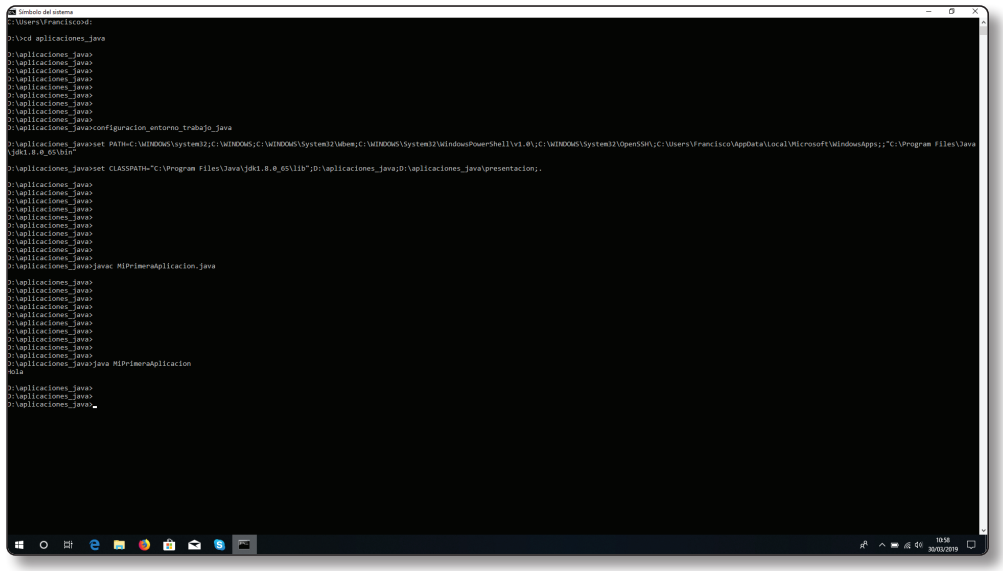
- **PATH .-** En esta variable de entorno, añadimos la ruta donde encontrar los ejecutables. En plataforma Windows, para poder trabajar en línea de comandos:

```
set PATH=%PATH%;"C:\Program Files\Java\jdk1.8.0_65\bin"
```

- **CLASSPATH .-** En esta otra variable de entorno, se establecen las rutas donde la JVM podrá encontrar los Byte Code generados por procesos de compilación, así como otras clases que pueda necesitar, aportados por el JDK:

```
set CLASSPATH="C:\Program Files\Java\jdk1.8.0_65\lib";D:\aplicaciones_
java;D:\aplicaciones_java\presentacion;.
```

Cuando trabajamos en línea de comandos en plataformas Windows, lo práctico es aglutinar la configuración de ambas variables de entornos en un archivo .bat, e invocarlo a ejecución al inicio de la sesión. Ilustramos en la siguiente captura de pantalla actuaciones en línea con lo que se está tratando.



En la sesión correspondiente a dicha captura de pantalla, en primer lugar, nos situamos en el directorio:

D:\aplicaciones_java

Y a continuación realizamos las siguientes acciones:

1. Invocar a ejecución el archivo configuración_entorno_trabajo_java.bat, que aglutina la configuración de las variables de entorno descritas.

2. Compilar el programa fuente **MiPrimeraAplicacion**, guardado en el archivo MiPrimeraAplicacion.java:

   javac MiPrimeraAplicacion.java

   Obsérvese que al compilador ha de proporcionarse el nombre completo del archivo, incluida la extensión .java. Como consecuencia de esta actuación se generará el archivo correspondiente al Byte Code **MiPrimeraAplicacion.class**.

3. Invocar al intérprete para que traduzca al código máquina de la plataforma que estamos utilizando, y ejecute, el Byte Code generado en el paso anterior:

   Estamos proporcionando al intérprete, el archivo **MiPrimeraAplicacion.class**, pero como se observará, no se detalla la extensión:

java MiPrimeraAplicacion

Observaremos que se visualiza

Hola

que es, al fin y al cabo, la única acción consecuencia de la ejecución del programa.

A efectos de mostrar aspectos a tener en cuenta cuando desarrollamos en línea de comandos aplicaciones organizadas en diferentes packages, aportamos a continuación el código de las clases **EjerciciosArrays3**, y **Estado**; y detallaremos como compilar y ejecutar dicha aplicación desde la línea de comandos, así como crear un icono en el escritorio para su invocación a ejecución. También aportamos la clase **DesgloseMoneda2**, que también utilizaremos para exponer como crear un icono para su invocación a ejecución desde el escritorio. Dichas clases corresponden a ejercicios ejemplos utilizado en temas posteriores. Por tanto, el lector deberá demorar la comprensión de los mismos, al momento en que aborde dichos temas. En el código correspondiente a dichos ejemplos, se ha añadido al final, en relación al que aparece en temas posteriores, la petición de la introducción de la tecla Intro, a efectos de poder visualizar el resultado de la ejecución antes de que desaparezca la ventana asociada a dicha ejecución, dado que se proponen como ejemplos a ser invocados desde iconos del escritorio. La distribución en directorios de estas clases ejemplo, puede observarse en el esquema previamente expuesto a tal efecto.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class DesgloseMoneda2 {

    public static void main(String[] args) {

        BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
        int importe = 0, numeroBilletes50 = 0, numeroBilletes20 = 0, numeroBi-
lletes10 = 0, numeroBilletes5 = 0, numeroMonedas2 = 0, numeroMonedas1;
        int numeroUnidades = 0;
          .
          .
          .
        numeroMonedas1 = numeroUnidades;
        System.out.println("Obtenemos también "+numeroMonedas1+" monedas de 1");

        try {
```

```
                System.out.print("Pulse INTRO para finalizar ");
                String teclaFin = bufferedReader.readLine();
        } catch (Exception excepcion)
        { System.out.println(excepcion.getMessage());
        }
    }
}
```

```
import encapsuladores.Estado;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class EjerciciosArrays3 {

    public static void main(String[] args) {

        //     DECLARACION, INSTANCIACION DEL ARRAY, E INSTANCIACION DE OBJETOS
        Estado[] estados = {new Estado(1, "España",   46570000), new Estado(2,
"Portugal", 10310000), new Estado(3, "Alemania", 82790000), new Estado(4, "Fran-
cia",  67120000), new Estado(5, "Italia",   60590000)};
          .
          .
          .
        System.out.println("El estado de mayor población es "+estados[posicionEs
tadoMasPoblado].getNombre()+ " que cuenta con "+estados[posicionEstadoMasPoblado
].getNumeroHabitantes());
        System.out.println("El estado de menor población es "+estados[posicionEs
tadoMenosPoblado].getNombre()+ " que cuenta con "+estados[posicionEstadoMenosPob
lado].getNumeroHabitantes());
        try {
                BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
                System.out.print("Pulse INTRO para finalizar ");
                String teclaFin = bufferedReader.readLine();
        } catch (Exception excepcion)
        { System.out.println(excepcion.getMessage());
        }
    }
}
```

```
package encapsuladores;

public class Estado implements java.io.Serializable {
```

```
    private int idEstado;
    private String nombre;
    private int numeroHabitantes;

    public Estado() {
    }
            .
          .
          .

    @Override
    public int hashCode() {
        return idEstado + nombre.hashCode() + numeroHabitantes ;
    }
}
```

Dado que dichas clases se utilizan en ejemplos tratados en temas posteriores de este libro, y que en dichos temas se expone el código de completo de las clases en cuestión, en este tema, en pro de minimizar la extensión, se ha suprimido del detalle, la mayor parte del código de las mismas, pero apareciendo reflejados la aportación novedosa de código que permite forzar una pausa ante la petición de la tecla Intro.

Las clases **EjerciciosArrays3**, y **Estado**, están asociadas a la misma aplicación ejemplo. Como se podrá comprobar, cada package se ha de corresponder con un directorio del sistema de archivos. Detallamos a continuación, todas las acciones realizadas para compilar y ejecutar dicha aplicación. Podrá observar el lector, dado que aparece reflejado previamente, en el prompt, el directorio en que nos situamos para lanzar dichas acciones:

1. Compilamos la clase **Estado**:

```
D:\aplicaciones_java>javac .\encapsuladores\Estado.java
```

2. Compilamos la clase **EjerciciosArrays3**:

```
D:\aplicaciones_java>javac .\presentacion\EjerciciosArrays3.java
```

3. Ejecutamos la clase **EjerciciosArrays3**, que es la que contiene el método de inicio de la aplicación *main*, y observamos también la visualización consecuencia de la ejecución de la aplicación:

```
D:\aplicaciones_java>java EjerciciosArrays3
identificador de estado : 1
nombre del estado : España
número de habitantes del estado : 46570000
-----------------------------------------------
identificador de estado : 2
nombre del estado : Portugal
número de habitantes del estado : 10310000
-----------------------------------------------
identificador de estado : 3
nombre del estado : Alemania
número de habitantes del estado : 82790000
-----------------------------------------------
identificador de estado : 4
nombre del estado : Francia
número de habitantes del estado : 67120000
-----------------------------------------------
identificador de estado : 5
nombre del estado : Italia
número de habitantes del estado : 60590000
-----------------------------------------------
El estado de mayor población es Alemania que cuenta con 82790000
El estado de menor población es Portugal que cuenta con 10310000
Pulse INTRO para finalizar
```

Es importante destacar la presencia de:

```
import encapsuladores.Estado;
```

en la clase **EjerciciosArrays3**, necesaria porque se instancian objetos de la clase **Estado**, y esta clase se encuentra en otro package.

Observamos también que en la invocación al intérprete para la ejecución de la aplicación, no ha sido necesario indicar la ruta donde se encuentra la clase EjerciciosArrays3 porque el Byte Code correspondiente a dicha clase se encuentra en una de las rutas establecidas en la variable de entorno CLASSPATH.

Detallamos a continuación los pasos a seguir para crear iconos que permitan la invocación a ejecución desde el escritorio de las aplicaciones **DesgloseMoneda2**, y **EjerciciosArrays3.** Es necesario que todas las clases correspondientes a

dichas aplicaciones estén previamente compiladas, y los Byte Code resultado de dichos procesos de compilación, en el directorio pertinente. Compilamos la clase **DesgloseMoneda2** mediante:

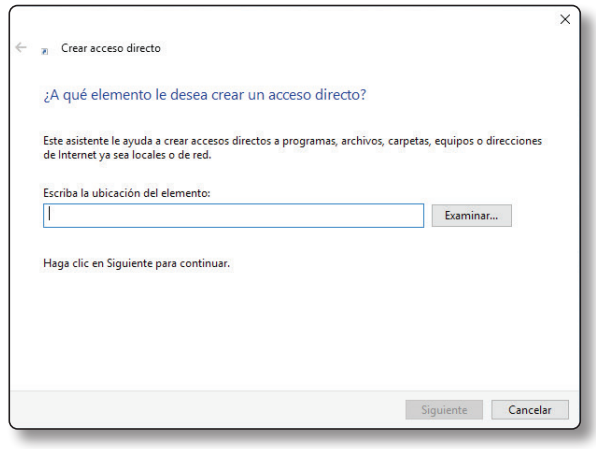
```
D:\aplicaciones_java>javac DesgloseMoneda2.java
```

Y a continuación para la creación del icono:

1. Hacer clic con el botón derecho del ratón sobre espacio libre del escritorio.
2. En el menú contextual que aparece, seleccionar:

```
Nuevo
      Acceso directo
```

3. Aparece la ventana



En el campo de texto en que solicita:

Escriba la ubicación del elemento:

editar:

```
"C:\Program Files\Java\jdk1.8.0_65\bin\java.exe" -classpath "C:\Pro-
gram Files\Java\jdk1.8.0_65\lib";D:\aplicaciones_java DesgloseMoneda2
```

4. Hacer clic en el botón **Siguiente** de dicha ventana.

5. En la siguiente ventana, editamos el campo de texto en que se nos solicita:

   Escriba un nombre para este acceso directo:

   con el texto que pretendemos que aparezca debajo del icono en el escritorio, por ejemplo:

   Desglose de moneda

6. Hacemos clic en el botón **Finalizar**.

Para obtener un icono en el escritorio asociado a la invocación a ejecución de la aplicación **EjerciciosArrays3**, repetimos los pasos de creación del anterior icono, y cuando se nos solicita la ubicación del elemento, editamos:

```
"C:\Program Files\Java\jdk1.8.0_65\bin\java.exe" -classpath "C:\Program Files\
Java\jdk1.8.0_65\lib";D:\aplicaciones_java;D:\aplicaciones_java\presentacion
EjerciciosArrays3
```

y para el nombre de dicho icono, le podemos proporcionar:

Mostrar contenido array

Recordemos que para utilizar el icono que acabamos de crear, los Byte Code correspondientes a todas las clases de la aplicación, deben estar ubicados previamente en el directorio pertinente.

Todo lo que se ha expuesto, relacionado con el desarrollo de aplicaciones en Java en un entorno de línea de comandos, se ha hecho de forma ilustrativa y testimonial, y para que el lector conozca los fundamentos de los procesos asociados a la compilación, e interpretación y ejecución de aplicaciones. Pero, insistimos en la recomendación de utilizar un IDE para el desarrollo de aplicaciones en Java.

Volviendo al primer ejemplo presentado en este tema, vamos a pasar a analizar la estructura de una aplicación en este lenguaje. Hay que señalar, que todavía no hemos entrado en un contexto de objetos.

Nos encontramos, en un principio, con que una aplicación Java, se compone de "unidades" llamadas clases (*class*). Dichas clases actúan, eminentemente, en la mayor parte de las ocasiones, como plantillas a partir de las cuales se instancian "clones" en memoria, tantos como se crea oportuno. A estos "clones", se les llama objetos. Pero todavía no vamos a entrar en estas consideraciones; todas las cuestiones relacionadas con objetos serán abordadas más adelante.

Pero retomemos el estudio en detalle de esta, nuestra primera aplicación. A la definición de la clase le correspondería el siguiente código:

```
class MiPrimeraAplicacion {
}
```

En que nos encontramos la palabra reservada *class*, seguida del nombre que le hemos dado a dicha clase, en este caso, *MiPrimeraAplicacion*. A continuación vemos que la pareja de caracteres *{ }* actúan como delimitadores de todos los elementos que constituirán dicha clase. En este nuestro primer ejemplo, solamente nos encontramos con un elemento constituyente de dicha clase: el método *main*.

Para facilitar la comprensión del ejemplo, vamos a suponer que en lugar de encontrarnos en la única clase que contiene esta aplicación, fuese cualquier otra. De este forma no nos veríamos obligados a este método llamarle *main* sino, por ejemplo, *imprimirCadena*.

En ese caso la mínima expresión en la estructura obligatoria de dicho método sería:

```
void imprimirCadena() {
    }
```

donde *void* corresponde al tipo de dato devuelto por el método, caso de *void* implica ausencia de devolución de dato por parte del método.

imprimirCadena es el nombre que le hemos proporcionado al método

A continuación del nombre del método, hay que ubicar la pareja de caracteres *()*, que actúan como delimitadores de los parámetros que se transfieren al método en su ubicación. En el caso de *imprimirCadena*, nos encontramos ante la ausencia de dichos parámetros. Con lo que llegamos a la conclusión de que la utilización de dichos parámetros es opcional.

Más adelante, también entraremos en consideraciones sobre tipos de datos devueltos por un método, así como parámetros que se transfieren al método en su invocación.

Y finalmente, volvemos a encontrarnos, pero en este caso para el método, al igual que antes hemos descrito para la clase, la pareja de caracteres *{ }*, que actúan, aquí como delimitadores de todo el contenido de dicho método.

Por tratarse en este ejemplo del método *main*, no podemos proporcionarle una estructura mínima como la que acabamos de presentar para *imprimirCadena*, por ser *main* un método único, obligatorio, y especial para la aplicación.

Observando en nuestro ejemplo la jerarquía de inclusiones, determinada por las parejas de *{ }* utilizadas, llegamos a la conclusión de los métodos son uno de los tipos de elementos constitutivos de las clases.

Volviendo al método *main*, podemos decir que supone el punto de entrada, o de "arranque" en la ejecución de código de la aplicación; y nos encontramos con que, dado su carácter especial y exclusivo debe ir acompañado, obligatoria y necesariamente de

- el modificador de ámbito de acceso *public*
- el modificador que determina se corresponde a un contexto de clase *static*
- y que obligatoriamente hay que transferirle un array de *String*.

Obviamente todos estos conceptos y cuestiones serán abordados exhaustivamente con posterioridad.

Como ya hemos comentado, una aplicación Java, se compone de clases; de tal modo que los correspondientes fuentes son archivos con extensión *.java*. Normalmente el código fuente de cada clase se almacena en un archivo con el mismo nombre de la clase, y extensión *.java*, como ya hemos mencionado. Estas a su vez, con fines de organización, pueden agruparse e incluirse en *packages*. De cara al almacenamiento de la aplicación en el sistema de archivos, estos *packages*, son carpetas, en que a su vez están incluidos los archivos correspondientes a las clases. El *package* en que se incluiría una clase, debe hacerse constar en el código fuente de la misma. De tal modo que en el caso de que quisiéramos estructurar esta aplicación inicial en un *package* llamado *miprimieraaplicacion*, en que se incluiría la clase *MiPrimeraAplicacion*, el código de la misma debería quedar:

```
package miprimeraaplicacion;

public class MiPrimeraAplicacion {

    public static void main(String[] args) {
        System.out.println("Hola");
    }

}
```

Hay que matizar que en el caso que se expone en estos momentos coinciden el nombre del paquete y el de la clase como consecuencia de que en el IDE que se está utilizando, cuando se utiliza el asistente de creación de proyectos, y se crea un proyecto, establece, por defecto, el mismo nombre al proyecto, el paquete inicial, y la clase inicial. En aplicaciones ya de cierta importancia y envergadura, en que los componentes se suelen estructurar atendiendo a criterios, como por ejemplo, los establecidos por la arquitectura a tres capas, los paquetes que aglutinan las clases suelen tener nombres relacionados con las funciones asociadas a la capa en cuestión, como por ejemplo:

- *presentacion*
- *negocio*
- *datos*
- *encapsuladores*
- *excepciones*

Respecto a la nomenclatura en Java, hay que señalar que Java no es indiferente al uso de mayúsculas – minúsculas. Y que existe un estándar de codificación en Java, que no es obligatorio, es decir, el transgredirlo no genera error de sintaxis, pero obviamente, sí es recomendable seguirlo. En relación a los elementos que ya han aparecido, este estándar establece que:

- **nombres de clase:** Todas las iniciales en mayúsculas, y el resto de letras en minúsculas.
- **nombres de método:** Igual que las clases, salvo la primera inicial, que será en minúsculas. Deberán ser verbos en infinitivo.
- **nombres de variables:** Al igual que los métodos, todas las iniciales en mayúsculas, excepto la primera, en minúscula, y el resto de letras en minúsculas.
- **nombres de paquetes:** Todo en minúsculas

1.2 TIPOS DE DATOS

Como cualquier otro lenguaje, Java dispone de un repertorio de tipos de datos que nos permitirán codificar de la forma más adecuada y eficiente cada dato de los que maneje la aplicación, en función de su naturaleza. Hay que señalar que Java es un lenguaje pródigo en tipos de datos; y además dado que es también fuertemente tipado, es decir, que todo dato que vaya a ser utilizado por la aplicación debe ser necesariamente declarado el tipo a que se corresponde, nos conviene conocer todos

esos tipos datos, así como sus características: naturaleza del dato a almacenar, espacio en bytes de memoria que utiliza para ser almacenado, así como el rango de valores que puede almacenar.

Nos encontramos, en una primera aproximación con

- referencias a objetos
- tipos de datos primitivos

Las referencias serán abordadas con posterioridad cuando entremos a "movernos" en un contexto de objetos. En cuanto a los tipos de datos primitivos, podemos reflejar en la siguiente tabla los tipos de datos primitivos con que contamos, así como el resto de características que hemos anunciado.

TIPO	LONGITUD	NATURALEZA	RANGO DE VALORES
boolean	sólo usa 1 bit	lógico	true / false
char	16 bits	carácter	Unicode
byte	8 bits	entero	- 128 … + 127
short	16 bits	entero	- 32.768 … + 32.767
int	32 bits	entero	- 2.147.483.648 … + 2.147.483.647
long	64 bits	entero	- 9.223.372.036.854.775.808… + 9.223.372.036.854.775.807
float	32 bits	real	coma flotante de simple precisión Norma IEEE 754
double	64 bits	real	coma flotante de doble precisión Norma IEEE 754

Las cadenas en Java no son tipo de dato primitivo, sino que son modeladas por la clase *String*, con lo que para su manipulación habrá que recurrir a los métodos de la mencionada clase. En consecuencia, las cadenas no son objeto de tratamiento en este apartado, sino que lo será más adelante.

Las referencias a objetos, cuya única finalidad es manipular y acceder a objetos y sus métodos, serán abordadas, en consecuencia, en el Tema 3: Clases. Objetos. Métodos.

Utilizaremos los tipos de datos primitivos cuando proceda manipular por parte del programa datos cuya naturaleza se detalla en la tercera columna de la tabla anterior. Para ello, deberemos declarar variables, o constantes, lo cual supone que una reserva y ocupación de un espacio de memoria por parte del programa, determinado

por lo detallado en la segunda columna de la tabla. Habrá diferentes lugares en que podamos realizar estas declaraciones, lo cual determinará el ámbito de utilización del dato. Dichos lugares podrán ser:

- variable, o constante local a un bloque de una estructura de control
- variable, o constante local a un método
- parámetro en la invocación a un método (solamente variables)
- variable, o constante, atributo de una clase, u objeto

Dado el carácter inicial de este tema solamente vamos a utilizar en él ejemplos de variables, o constantes locales a un método.

La diferencia entre una variable y una constante reside en que en el espacio de memoria asignado a una variable, puede cambiar el valor almacenado a lo largo de la ejecución del programa; cosa que no ocurre con las constantes, en que solamente se podrá almacenar durante toda la ejecución del programa un solo valor, que no podrá ser sustituido por otro.

Declaramos una variable o constante especificando en primer el lugar el tipo primitivo en función de la naturaleza del dato, y a continuación el nombre que le daremos a dicha variable para ser manipulada por el programa. El criterio que seguiremos para escoger el tipo de dato que asociaremos a la variable, será, teniendo en cuenta la naturaleza del dato, y el máximo valor que podrá almacenar dicha variable, escoger el tipo más ajustado en tamaño. Por ejemplo:

```
int saldo;
```

Podemos declarar más de una variable utilizando una única especificación de tipo, separándolas por “ , “. Por ejemplo:

```
double  precioUnitario,  importeTotal;
```

Posteriormente a la declaración podremos manipular el espacio de memoria asignado a la variable almacenando valores en esa variable, o utilizando dicho valor en la evaluación de expresiones (ello se tratará con más extensión en el apartado relacionado con *operadores*). Vamos a asignar el primer valor a la variable saldo:

```
saldo = 3500;
```

para ello hemos una asignación. De tal modo que en primer lugar especificamos la variable (que identifica el espacio de memoria) que va a ser receptora de la expresión (un literal en este caso) que, a su vez, especificamos a la derecha del " = ".

Obsérvese, aunque todavía no se ha mencionado, que todas las líneas del código del programa deben finalizar en " *;* ".

Podemos comprobar que dicho almacenamiento se ha producido, visualizando el valor depositado en dicha variable:

```
System.out.println(saldo);
```

Podremos sustituir el valor previamente almacenado en la variable por otro valor:

```
saldo = 3800;
```

y al volver a visualizar

```
System.out.println(saldo);
```

comprobaremos que el valor original ha sido sustituido por el nuevo valor.

También se podría haber hecho coincidir la declaración de la variable, con la asignación del primer valor a la misma:

```
int precio = 15;
```

Todas estas actuaciones comportan que el programa está utilizando dos espacios de memoria diferentes, designados como saldo y precio, declarados con el tipo int; lo cual implica que nuestra intención es manipular datos de naturaleza entera, y situados en el rango de valores situado entre - 2.147.483.648 y + 2.147.483.647. Rango de valores que supera con creces el valor asignado en cada caso a la variable. Si los valores que se les vayan a asignar a esas variables fuesen aproximados a los literales utilizados, en aras de la optimización, deberíamos haber declarado:

```
short saldo;
byte precio = 15;
```

Todo ello podemos verlo añadido a nuestro programa inicial:

```
package miprimeraaplicacion;

public class MiPrimeraAplicacion {

    public static void main(String[] args) {
        int precio = 15;
        int saldo;
        System.out.println("Hola");
        saldo = 3500;
        System.out.println(saldo);
        saldo = 3800;
        System.out.println(saldo);
        System.out.println(precio);
    }

}
```

Comprobaremos que las dos declaraciones de las variables se han situado al principio del método *main*. Ello no es obligatorio, de hecho, se podrían haber ubicado las declaraciones en cualquier lugar del código (obviamente siempre previa cada declaración a la utilización posterior de la variable); pero forma parte de los buenos usos y costumbres ubicar al principio todas las declaraciones que se realicen, tanto de tipos primitivos, como de referencias.

Las constantes, en las que como ya se ha indicado, solamente se puede almacenar un solo valor a lo largo de toda la ejecución del bloque en que han sido declaradas, sin poder ser sustituido por otro, se declaran con el modificador final:

```
final byte IVA = 21;
final double PI;
PI = 3.1416;
```

En relación a la convención de nomenclatura, las constantes será con todas las letras en mayúsculas. Y las variables, igual que los métodos: todo en minúsculas, con las iniciales de cada palabra (si es un nombre compuesto) en mayúsculas, excepto la primera inicial, que será en minúscula. Por ejemplo:

```
int saldoClienteBalance;
```

1.3 OPERADORES

En Java tenemos disponibles los operadores cuya clasificación a continuación se detalla. Dicha clasificación está basada en la naturaleza del tipo de dato o actuación que se devuelve por su salida. Como veremos nos encontramos con operadores de 2 entradas, es decir es necesario aplicar 2 operandos; y operadores de 1 sola entrada.

OPERADOR	DESCRIPCIÓN	FORMATO
OPERADORES ARITMETICOS DE 2 ENTRADAS Actúan con 2 operandos de naturaleza numérica y devuelven un valor numérico.		
+	suma	operando1 + operando2
-	resta	operando1 - operando2
*	producto	operando1 * operando2
/	cociente	operando1 / operando2
%	resto (de una división entera)	operando1 % operando2
OPERADORES ARITMETICOS DE 1 ENTRADA Actúan con 1 operando de naturaleza numérica y devuelven un valor numérico.		
-	cambio de signo	-operando
++	incremento en uno	++operando operando++
--	decremento en uno	--operando operando--
OPERADORES RELACIONALES Actúan con dos operandos de igual tipo primitivo ambos y devuelven un valor boolean que es true si se cumple lo especificado en la columna descripción y false en caso contrario.		
= =	igual	operando1 = = operando2
! =	distinto	operando1 ! = operando2
<	menor	operando1 < operando2
< =	menor ó igual	operando1 < = operando2
>	mayor	operando1 > operando2
> =	mayor o igual	operando1 > = operando2
OPERADORES LÓGICOS DE 2 ENTRADAS Actúan con dos operandos que al ser evaluados previamente devuelven un boolean. Devuelven un valor boolean que es true si se cumple lo especificado en la columna descripción y false en caso contrario. Habitualmente, estos operandos, son a su vez expresiones relacionales.		
&&	AND : es *true* el valor de ambos operandos	operando1 && operando2
| |	OR : es *true* uno de los 2 operandos	operando1 | | operando2

OPERADOR LÓGICO DE 1 ENTRADA Actúa con 1 solo operando que al ser evaluado previamente devuelve un boolean. Devuelve un valor boolean.		
!	NOT : devuelve *true* si el operando es *false*, y *false* en caso contrario	! operando
OPERADORES DE DESPLAZAMIENTO DE BITS El primer operando es tipo primitivo de naturaleza entera o char. El segundo operando es un valor numérico de naturaleza entera. En el primer operando es donde se produce el desplazamiento de todos los bits, tantas veces como indique el segundo operando.		
<<	desplazamiento a la izquierda, por la derecha entran siempre un 0	operando1 << operando2
>>	desplazamiento a la derecha, por la derecha entra el bit de signo	operando1 >> operando2
>>>	desplazamiento a la derecha, por la izquierda entran siempre un 0	operando1 >>> operando2
OPERADORES LÓGICOS A NIVEL DE BIT Ambos operandos son de naturaleza entera o char. El operador actúa realizando operaciones lógicas, equivalentes a las presentadas con anterioridad, pero en este caso, bit a bit, entre los que ocupan la misma posición relativa de cada uno de los dos operandos. El resultado es otro valor, del mismo tipo que los operandos, pero con los bits resultantes de la actuación descrita.		
&	AND bit a bit	operando1 & operando2
|	OR bit a bit	operando1 | operando2
^	XOR: OR exclusivo bit a bit	operando1 ^ operando2
~	NOT bit a bit (al igual que su equivalente anteriormente descrito, actúa sobre un solo operando)	~ operando1

Los operadores son elementos del programa, que al actuar con los operandos, nos permitirán definir expresiones. Los operandos utilizados en una expresión podrán ser:

- variables,
- constantes
- literales
- valores devueltos en la ejecución de un método
- …

El objetivo de estas expresiones será que en tiempo de ejecución del programa, al ser evaluadas, proporcionen resultados que serán utilizados con diversos fines:

- asignar el valor resultado a una variable
- transferir dicho valor resultado como parámetro actual al invocar un método

- ser aplicado a *return*, y en consecuencia conformar el valor que devolverá el método al finalizar su ejecución al ser invocado
- ser aplicados al constructor de un objeto en su instanciación
- …

De todos estos ámbitos de utilización de una expresión, en estos momentos, solamente vamos a utilizar ejemplos de la primera variante de utilización, es decir asignar el valor resultado a una variable. Iremos tomando contacto y conciencia de las otras variantes de utilización en posteriores temas del presente libro. Vamos a ilustrar esta primera variante de utilización con un ejemplo:

```
package miprimeraaplicacion;
public class MiPrimeraAplicacion {
    public static void main(String[] args) {
        double precioUnitario, importeTotal;
        byte unidades;
        final byte IVA = 21;
        precioUnitario = 53.78;
        unidades = 10;
        importeTotal = ((precioUnitario * unidades)) + ((IVA / 100.0) * (pre-
cioUnitario * unidades));
    //  --------------       ------------------------------------          ---
-----------------------------------------------
    //      liquido                     importe bruto
importe IVA
        System.out.println("el precio final de la compra es: " + importeTotal);
    }
}
```

Se analizará este ejemplo, como implementación práctica de lo que se ha detallado, para clarificar todos los conceptos e ideas que se han expuesto. Pasemos a ese análisis pormenorizado:

- En primer lugar se declaran las variables *precioUnitario, importeTotal, unidades.*
- Se declara también la constante *IVA*, asignando el valor 21 en la propia declaración.
- Se asignan valores a las variables *precioUnitario*, y *unidades*.

- La siguiente instrucción podemos descomponerla en 3 partes:
  - En primer lugar la variable *importeTotal* que recibirá y almacenará el resultado final de la evaluación de la expresión.
  - A continuación " = " , que implementa el operador asignación. El operador asignación, como ya se ha mencionado con anterioridad, deposita en la variable ubicada a la izquierda, el resultado de la evaluación de la expresión ubicada a la derecha.
  - Finalmente nos encontramos con la expresión, formada por operandos que se pasan a los operadores, para, en tiempo de ejecución, dar lugar a un resultado final. En la evaluación de la expresión, con los operandos tipo variable, se utiliza el valor almacenado en esa variable, en ese momento de la ejecución.

- Finalizamos el código visualizando un mensaje al que concatenamos la variable *importeTotal* cuyo valor almacenado es el que pretendemos visualizar. El operador concatenación " + ", lo utilizamos para "concatenar" cadenas, es decir, poner una cadena a continuación de otra y conformar una sola. Hay que decir que, *importeTotal* no es una cadena, pero que, al utilizarla en la concatenación utilizada, se aplica el valor almacenado convertido en cadena. Con lo que al ejecutar el programa se visualiza por la consola de salida

  el precio final de la compra es: 650.7379999999999

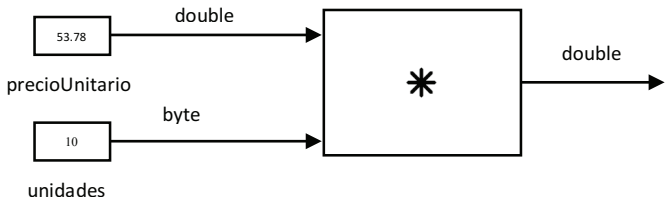
  No nos tiene que preocupar el número de cifras decimales que se visualizan por defecto.

  En temas posteriores trataremos clases cuyo único fin es el formateo de datos.

A continuación, procederemos al análisis exhaustivo de la expresión utilizada, tratando todos sus detalles y pormenores. La expresión utilizada calcula el importe total de una compra sumando al importe bruto de una compra (multiplicando unidades adquiridas por el precio unitario), el importe correspondiente al IVA. Analicemos cada una de las "subexpresiones" que constituyen la expresión total:

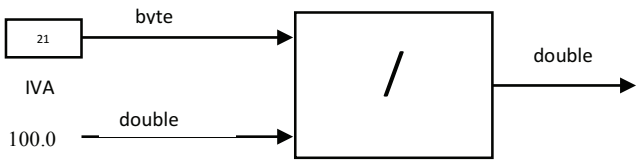
- precioUnitario * unidades

  Esta "subexpresión" aparece en dos ocasiones, y en cada una de ellas se aplica el contenido de dos variables, una *double* y un *byte* como entradas a un operador "*", la salida de dicho operador es un *double*.



- *IVA / 100.0*

  Se aplica a las entradas del operador " / " el valor de una constante *byte* (IVA) y un literal *double* (100.0). Observemos que en este caso, el literal es un dato que podría tener naturaleza entera (podría ser simplemente 100), dada la ausencia de valor en la parte decimal. Se actúa de esta manera para provocar que el operador " / " actúe como división real, y pueda devolver un cociente real (0.21). Caso de que el literal fuese (100), ambos operandos tendrían naturaleza entera, el operador " / " actuaría como división entera, y en consecuencia devolvería un cociente entero, en este caso (0), solamente la parte entera del cociente; y al aplicar este (0) a la "subexpresión" de cálculo del importe bruto, el importe correspondiente al impuesto IVA sería (0); circunstancia ésta nada grata al Ministerio de Hacienda.



Se deduce, por los ejemplos, que cuando a un operador de dos entradas, se aplican por cada una de ellas operandos de diferente tipo, la salida del operador es un dato del tipo del de mayor complejidad o tamaño de los dos aplicados por las entradas.

En esta segunda "subexpresión" podríamos haber recurrido a las siguientes alternativas para conseguir el mismo efecto:

- *IVA / (double)100*
- *(double)IVA / 100*
- *(double)IVA / (double)100)*

En esta última alternativa se aplica *(double)* a ambos operandos, lo cual es innecesario, ya que con aplicarlo a solamente uno de ellos ya es suficiente.

Vamos a explicar qué ha ocurrido. Hemos aplicado el operador *cast* , implementado por *"(tipo de dato)"*, el cual tiene el efecto de un cambio explícito de tipo justo a la entrada del operador, permaneciendo inalterado el tipo original del dato en su almacenamiento en memoria.

También se observa la utilización de *()* , con el objetivo de jerarquizar las "subexpresiones". En este caso no es necesario, de no utilizarlos, el resultado sería el mismo, y solamente se emplean para proporcionar legibilidad a la interpretación "humana" de la expresión. Pero en otros casos sí serían necesarios, operativamente hablando, para "forzar" la prioridad en el orden de evaluación y actuación de los operadores, cuando el orden que nos interesa no viene establecido por la precedencia con que por defecto Java aplica dichos operadores.

En cambio, si la expresión la construimos como a continuación se detalla:

*precioFinal = (precioUnitario * unidades) * (1 + (IVA / (double)100));*

comprobaremos que el resultado es el mismo. Lo único que hemos hecho es aplicar, matemáticamente hablando, "sacar factor común". Pues bien, en este último caso, la aplicación de *()*, sí es necesaria a efectos de priorizar la evaluación de " + ", respecto al " * " más externo; que de no aplicar *()*, se hubiese evaluado con anterioridad.

De no haber aplicado *()*:

*precioFinal = precioUnitario * unidades * 1 + IVA / (double)100;*

el resultado obtenido en la evaluación de la expresión no hubiese sido el deseado.

No podemos finalizar las observaciones sobre el programa que estamos tratando sin mencionar la aparición de " // ". Estos caracteres permiten ubicar comentarios en el código del programa, que no tendrán ninguna repercusión en la ejecución del mismo.

Podríamos también haber expresado estas dos líneas de comentarios utilizando " /* " y " */ ", que actúan como delimitadores de un conjunto de líneas comentario. Para ello, habríamos aplicado:

```
        /*
            -----------   ----------------------------   ------------------------
   --------------------
              liquido               importe bruto                  importe IVA
        */
```

El orden de precedencia establecido por defecto entre los operadores descritos en la tabla expuesta con anterioridad es:

NIVEL DE PRIORIDAD	GRUPO DE OPERADORES
1	Unarios (todos los que actúan sobre un solo operando)
2	Aritméticos: * / %
3	Aritméticos: + -
4	Desplazamiento de bits
5	Relacionales: < <= > >=
6	Relacionales: = = ! =
7	AND bit a bit
8	XOR bit a bit
9	OR bit a bit
10	AND boolean
11	OR boolean

Hay que matizar que el orden de las acciones derivadas de los operadores aritméticos ++ y -- viene determinado por el orden en que aparecen en la expresión. De tal forma que en el caso de

```
operando++
operando- -
```

en primer lugar se evalúa el operando en la expresión de que forme parte, y a continuación se incrementa o decrementa la variable que actúa como operando. En el caso de

```
++operando
- -operando
```

el orden en que se desarrollan las acciones es inverso a lo descrito.

Es muy habitual en programación el uso de asignaciones en que a una variable se le asigna el resultado de una expresión en que su anterior valor actúa como operando. Por ejemplo:

```
numero1 = numero1 + numero2;
```

Dicha asignación es perfectamente aplicable en Java. Pero en este lenguaje, podemos, además implementarlo:

```
numero1 += numero2;
```

obteniendo el mismo efecto.

Exponemos a continuación un programa en que ilustramos dicha equivalencia aplicada a otros operadores:

```
package pruebasasignaciones;
public class PruebasAsignaciones {
    public static void main(String[] args) {
        int numero1 = 20;
        int numero2 = 4;
        System.out.println("Para valores de numero1 : "+numero1+"    numero2 :
"+numero2);
        numero1 += numero2; // equivale a:  numero1 = numero1 + numero2;
        System.out.println("numero1 += numero2 -->> "+numero1);
        numero1 = 20;
        numero2 = 4;
        numero1 -= numero2; // equivale a:  numero1 = numero1 - numero2;
        System.out.println("numero1 -= numero2 -->> "+numero1);
        numero1 = 20;
        numero2 = 4;
        numero1 *= numero2; // equivale a:  numero1 = numero1 * numero2;
        System.out.println("numero1 *= numero2 -->> "+numero1);
        numero1 = 20;
        numero2 = 4;
        numero1 /= numero2; // equivale a:  numero1 = numero1 / numero2;
        System.out.println("numero1 /= numero2 -->> "+numero1);
        numero1 = 20;
        numero2 = 3;
        System.out.println("Para valores de numero1 : "+numero1+"    numero2 :
"+numero2);
        numero1 %= numero2; // equivale a:  numero1 = numero1 % numero2;
        System.out.println("numero1 %= numero2 -->> "+numero1);
    }
}
```

La ejecución del programa, permite visualizar en la consola de salida:

Para valores de numero1 : 20 numero2 : 4
numero1 += numero2 -->> 24
numero1 -= numero2 -->> 16
numero1 *= numero2 -->> 80
numero1 /= numero2 -->> 5
Para valores de numero1 : 20 numero2 : 3
numero1 %= numero2 -->> 2

Ejemplos de utilización de operadores relacionales, y de operadores lógicos con resultado *boolean*, se verán ampliamente más adelante, y sobre todo en el tema relacionado con estructuras de control.

A continuación vamos a exponer varios ejemplos muy breves en que se ilustra la utilización de operadores sobre bits. Para ello tenemos en cuenta que dado un *byte*, el peso específico de cada uno de los bits es:

2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

Para poner a 0 un bit de un dato, hay que aplicar a dicho dato el operador AND binario con otro dato que tenga todos los bits a 1, excepto en la posición que se quiere poner a 0. El resultado será el operando original con todos los bits iguales excepto el que hemos puesto a 0:

```
package trabajandoanivelbit;
public class TrabajandoANivelBit {
    public static void main(String[] args) {
       // Poner un bit a 0
       byte operando1 = 43;                                          //     0  0
1  0  1  0  1  1  ==>>  43 en binario
       byte operando2 = 119;                                        //     0  1
1  1  0  1  1  1  ==>> 119 en binario
       byte resultado;                                                    //  &
-------------------------
       resultado = (byte) (operando1 & operando2);   //       0  0  1  0  0  0
1  1  ==>>  35 en binario
       System.out.println("resultado: "+resultado);
    }
}
```

Para poner a 1 un bit de un dato, hay que aplicar a dicho dato el operador OR binario con otro dato que tenga todos los bits a 0, excepto en la posición que se quiere poner a 1. El resultado será el operando original con todos los bits iguales excepto el que hemos puesto a 1:

```
package trabajandoanivelbit;
public class TrabajandoANivelBit {
    public static void main(String[] args) {
       // Poner un bit a 1
       byte operando1 = 43;                                        //    0  0  1
0  1  0  1  1  ==>>  43 en binario
       byte operando2 = 16;                                        //    0  0  0
1  0  0  0  0  ==>>  16 en binario
       byte resultado;                                                  //  |
-------------------------
       resultado = (byte) (operando1 | operando2);   //     0  0  1  1  1  0  1
```

```
1  ==>>  59 en binario
        System.out.println("resultado: "+resultado);
    }
}
```

A continuación, un ejemplo de desplazamiento de bits:

```
package trabajandoanivelbit;
public class TrabajandoANivelBit {
    public static void main(String[] args) {
       // Desplazamiento de bits hacia la derecha
       byte operando1 = 45;                                  //    0  0  1  0  1  1
0  1  ==>>  45 en binario
       byte resultado;                                            //  al desplazar
todos los bits
                                                                          //  2
posiciones hacia la derecha obtenemos
       resultado = (byte) (operando1 >> 2);      //    0  0  0  0  1  0  1  1
==>>  11 en binario
       System.out.println("resultado: "+resultado);
    }
}
```

Mediante el siguiente programa visualizamos si cada uno de los bits de un byte está a 1 ó a si está a 0. Hay que tener presente que para la comprensión del código del mismo, es necesario haber tratado previamente el tema de Estructuras de control.

```
package trabajandoanivelbit;
public class TrabajandoANivelBit {
    public static void main(String[] args) {
       // Visualizar los bits de un byte
       byte datoAVisualizarBits = 45;                            //    0  0  1  0
1  1  0  1  ==>>  45 en binario

       for (byte i = 64 ; i >0  ;  i = (byte)(i/2)){
          byte resultado =  (byte)(datoAVisualizarBits & i);
          if (resultado > 0)
               System.out.print("1  ");
            else
               System.out.print("0  ");
       }
    }
}
```

1.4 JAVA Y LAS MATEMÁTICAS

Java dispone de la clase *Math* para ofrecer al programador todo un conjunto de métodos que implementan funciones matemáticas. Todos ellos son *static*, lo cual implica que son invocados en ausencia de objetos, y con el nombre de la clase según:

Math.metodo(parámetros)

En temas posteriores se abordará el concepto *static*. Presentamos un ejemplo de utilización de los principales métodos de dicha clase:

```
package pruebasmetodosmath;

public class PruebasMetodosMath {

    public static void main(String[] args) {
        System.out.println("POTENCIA                                    Math.
pow(3,2):     "+Math.pow(3,2));
        System.out.println("RAIZ CUADRADA                               Math.
sqrt(144):    "+Math.sqrt(144));
        System.out.println("LOGARITMO DECIMAL                           Math.
log10(10000)  "+Math.log10(10000));
        System.out.println("El número E elevado al valor proporcionado Math.exp(3):
"+Math.exp(3));
        System.out.println("LOGARITMO NEPERIANO   Math.log(20.085536923187668)
"+Math.log(20.085536923187668));
        System.out.println("VALOR de E:     "+Math.E);
        System.out.println("NUMERO ALEATORIO                      Math.random()
"+Math.random());
        // Para la generación de números aleatorios, también se puede recurrir a la
clase Random
        System.out.println("VALOR ABSOLUTO                        Math.abs(-15):
"+Math.abs(-15));
        System.out.println("REDONDEO AL ENTERO MENOR MAS PROXIMO  Math.floor(53.8):
"+Math.floor(53.8));
        System.out.println("REDONDEO AL ENTERO MAYOR MAS PROXIMO  Math.ceil(53.8):
"+Math.ceil(53.8));
        System.out.println("REDONDEO AL ENTERO MAS PROXIMO        Math.round(53.2):
"+Math.round(53.2));
        System.out.println("REDONDEO AL ENTERO MAS PROXIMO        Math.round(53.8):
"+Math.round(53.8));
        System.out.println("REDONDEO AL DOUBLE MAS PROXIMO        Math.rint(53.2):
"+Math.rint(53.2));
        System.out.println("REDONDEO AL DOUBLE MAS PROXIMO        Math.rint(53.8):
"+Math.rint(53.8));
        System.out.println("VALOR MAXIMO   Math.max(15,70):     "+Math.max(15,70));
        System.out.println("VALOR MINIMO   Math.min(15,70):     "+Math.min(15,70));
```

```
        System.out.println("VALOR de PI:     "+Math.PI);
        System.out.println("CONVERSION DE GRADOS SEXAGESIMALES A RADIANES   Math.
toRadians(90):                 "+Math.toRadians(90));
        System.out.println("CONVERSION DE RADIANES A GRADOS SEXAGESIMALES   Math.
toDegrees(1.5707963267948966): "+Math.toDegrees(1.5707963267948966));
        // En la obtención de funciones trigonométricas, el parámetro se aporta
expresado en radianes
        System.out.println("SENO            Math.sin(Math.toRadians(90)):
"+Math.sin(Math.toRadians(90)));
        System.out.println("ARCO SENO       Math.toDegrees(Math.asin(1)):
"+Math.toDegrees(Math.asin(1)));
        System.out.println("COSENO          Math.cos(Math.toRadians(30)):
"+Math.cos(Math.toRadians(30)));
        System.out.println("ARCO COSENO     Math.toDegrees(Math.
acos(0.8660254037844387))): "+Math.toDegrees(Math.acos(0.8660254037844387)));
        System.out.println("TANGENTE        Math.tan(Math.toRadians(45)):
"+Math.tan(Math.toRadians(45)));
        System.out.println("ARCO TANGENTE   Math.toDegrees(Math.atan(1)):
"+Math.toDegrees(Math.atan(1)));
    }
}
```

La ejecución de este programa ejemplo, visualiza en la consola de salida:

```
POTENCIA                                    Math.pow(3,2):     9.0
RAIZ CUADRADA                               Math.sqrt(144):    12.0
LOGARITMO DECIMAL                           Math.log10(10000)  4.0
El número E elevado al valor proporcionado Math.exp(3):       20.085536923187668
LOGARITMO NEPERIANO   Math.log(20.085536923187668)            3.0
VALOR de E:     2.718281828459045
NUMERO ALEATORIO                      Math.random()     0.9660387228619638
VALOR ABSOLUTO                        Math.abs(-15):    15
REDONDEO AL ENTERO MENOR MAS PROXIMO  Math.floor(53.8): 53.0
REDONDEO AL ENTERO MAYOR MAS PROXIMO  Math.ceil(53.8):  54.0
REDONDEO AL ENTERO MAS PROXIMO        Math.round(53.2): 53
REDONDEO AL ENTERO MAS PROXIMO        Math.round(53.8): 54
REDONDEO AL DOUBLE MAS PROXIMO        Math.rint(53.2):  53.0
REDONDEO AL DOUBLE MAS PROXIMO        Math.rint(53.8):  54.0
VALOR MAXIMO   Math.max(15,70):     70
VALOR MINIMO   Math.min(15,70):     15
VALOR de PI:     3.141592653589793
CONVERSION DE GRADOS SEXAGESIMALES A RADIANES   Math.toRadians(90):
1.5707963267948966
CONVERSION DE RADIANES A GRADOS SEXAGESIMALES   Math.toDe-
grees(1.5707963267948966): 90.0
SENO            Math.sin(Math.toRadians(90)):                  1.0
ARCO SENO       Math.toDegrees(Math.asin(1)):                  90.0
```

```
COSENO          Math.cos(Math.toRadians(30)):
0.8660254037844387
ARCO COSENO     Math.toDegrees(Math.acos(0.8660254037844387))):
29.999999999999993
TANGENTE        Math.tan(Math.toRadians(45)):
0.9999999999999999
ARCO TANGENTE   Math.toDegrees(Math.atan(1)):                    45.0
```

En este ejemplo podríamos haber recurrido a haber aplicado importación estática. El código del programa hubiese quedado:

```
packa package pruebasmetodosmath;

import static java.lang.System.*;
import static java.lang.Math.*;

public class PruebasMetodosMath {

    public static void main(String[] args) {
       .     .     .
        out.println("VALOR ABSOLUTO                         abs(-15):    "+abs(-
15));
        out.println("REDONDEO AL ENTERO MENOR MAS PROXIMO  floor(53.8):
"+floor(53.8));
        out.println("REDONDEO AL ENTERO MAYOR MAS PROXIMO  ceil(53.8):
"+ceil(53.8));
    .     .     .
     }
}
```

Hemos añadido las líneas:

```
import static java.lang.System.*;
import static java.lang.Math.*;
```

lo que nos permite prescindir del nombre de la clase en la invocación a los métodos:

```
out.println()
abs()
floor
ceil
```

así como en el resto de métodos de la clase *Math* utilizados en el ejemplo.