Capítulo 1

# Fundamentos de programación en Java

## 1.1. Empezar con Java

Java es ante todo un lenguaje de programación moderno. Fue diseñado en la década de los noventa, y eso se nota en cuanto uno empieza a trabajar con él, nos proporciona una potencia, una robustez y una seguridad que muy pocos lenguajes pueden igualar, sin olvidar su rasgo más conocido: es totalmente portable. Todas estas características y otras que iremos descubriendo a lo largo de estas páginas, hacen de Java un lenguaje necesario que cubre un hueco enorme en el panorama de la programación moderna.

Si bien Java tiene su base en lenguajes como C y C++, los supera con creces y sería un error pensar que es una simple evolución de éstos. Java tiene entidad propia y características novedosas y potentes que hacen de él no sólo una apuesta de futuro, sino también un lenguaje con un presente claro y asentado. No obstante, toda la potencia que Java proporciona tiene un coste; es necesario asimilar muchos conceptos, técnicas y herramientas que en muchos casos son totalmente nuevas y hacen que la curva de aprendizaje sea pronunciada. Sin embargo, una vez superados los primeros escollos, los resultados son espectaculares y merece la pena el esfuerzo.

### 1.1.1. Un poco de historia

En 1991 un grupo de ingenieros de Sun Microsystems liderados por Patrick Naughton y James Gosling comienza el desarrollo de un lenguaje destinado a generar programas independientes de la plataforma en la que se ejecutan. Su objetivo inicial nada tiene que ver con lo que hoy en día es Java, sus creadores buscaban un lenguaje para programar los controladores utilizados en la

electrónica de consumo. Existen infinidad de tipos de CPU distintas, y generar código para cada una de ellas requiere un compilador especial y el desarrollo de compiladores es caro.

Después de dieciocho meses de desarrollo aparece la primera versión de un lenguaje llamado OAK que más tarde cambiaría de nombre para convertirse en Java. La versión de 1992 está ampliada, cambiada y madurada, y a principios de 1996 sale a la luz la primera versión de Java. Los inicios son difíciles, no se encuentran los apoyos necesarios en Sun y el primer producto que sale del proyecto, un mando a distancia muy poderoso y avanzado, no encuentra comprador. Pero el rumbo de Java cambiaría debido a una tecnología completamente ajena a los controladores de electrodomésticos: Internet.

Mientras Java se estaba desarrollando, el mundo de las comunicaciones crecía a una velocidad de vértigo, Internet y principalmente el mundo World Wide Web dejaban los laboratorios de las universidades y llegaban a todos los rincones del planeta. Se iniciaba una nueva era y Java tuvo la suerte de estar allí y aprovechar la oportunidad. En 1993 con el fenómeno Internet en marcha, los desarrolladores de Java dan un giro en su desarrollo al darse cuenta de que el problema de la portabilidad de código de los controladores es el mismo que se produce en Internet, una red heterogénea y que crece sin parar, y dirigen sus esfuerzos hacia allí. En 1995 se libera una versión de HotJava, un navegador escrito totalmente en Java y es en ese mismo año cuando se produce el anuncio por parte de Netscape de que su navegador sería compatible con Java. Desde ahí otras grandes empresas se unen y Java se expande rápidamente.

No obstante, las primeras versiones de Java fueron incompletas, lentas y con errores. Han tenido que pasar varios años de desarrollo y trabajo para que Java sea un lenguaje perfectamente asentado y lleno de posibilidades. Actualmente es ampliamente utilizado en entornos tanto relacionados con Internet como completamente ajenos a la Red.

El mundo Java está en constante desarrollo, las nuevas tecnologías surgen y se desarrollan a gran velocidad haciendo de Java un lenguaje cada día mejor y que cubre prácticamente todas las áreas de la computación y las comunicaciones, desde teléfonos móviles hasta servidores de aplicaciones usan Java.

## 1.1.2. Versiones de Java

En algunos momentos, dado lo cambiante de la tecnología Java y sobre todo por la ingente cantidad de siglas que aparecen relacionadas directa o indirectamente con ella, surgen confusiones a raíz de las denominaciones de los productos o incluso sobre las versiones de los mismos.

Java ha cambiado a lo largo de los años, habiéndose liberado a día de hoy cerca de cuarenta versiones del entorno de desarrollo y de ejecución. Todas las versiones se pueden agrupar en tres grandes grupos. Cada uno de estos grupos representa un salto cuantitativo y cualitativo del producto.

- **Java 1.0.** Como se ha comentado anteriormente, la primera versión de Java se libera en 1996, nace la versión 1.0. Dos meses después aparece la versión 1.02, solucionando algunos problemas. Es el inicio del lenguaje y en estos momentos poco más que un sencillo applet era posible hacer con éste.

- **Java 1.1.** La siguiente revisión importante es la 1.1; el lenguaje empieza a tomar forma, la biblioteca de clases que acompaña al lenguaje es cada vez más completa.
- **Java 2.** Es en 1998 cuando Java da un verdadero paso adelante con la aparición de la versión 1.2; con ella nace Java 2 y es cuando el lenguaje se estabiliza definitivamente.

Dentro de Java 2 se han liberado hasta el momento tres grupos de versiones, la propia 1.2, la 1.3 y recientemente la 1.4. Cada una de ellas proporciona un pequeño avance sobre lo definido por Java 2, soluciona problemas, incrementa la velocidad y sobre todo hace crecer la biblioteca de clases. En este momento el lenguaje está estable y todos los esfuerzos se centran en ampliar las bibliotecas y proporcionar nuevas API para controlar, tratar o manejar cualquier tipo de dispositivo, dato o estructura imaginable.

Existe también cierta confusión con la denominación de los productos o tecnologías relacionados con Java. El entorno de desarrollo y ejecución de Java utilizado en este libro es el Java 2 Standard Edition (J2SE), que permite la creación de programas y de applets y su ejecución en la máquina virtual Java. En el momento de escribir este libro, la última versión estable de Java es la J2SE 1.4.1.

## 1.1.3. Compilación y ejecución en Java

El proceso de compilación y ejecución en Java requiere de la utilización de dos componentes del entorno de desarrollo; por un lado debemos compilar el código java y por otro debemos ejecutar el programa generado. En otros lenguajes de programación el resultado de la compilación es directamente ejecutable por el sistema operativo; pero en Java, el resultado de la compilación es un código que no es ejecutable por un procesador concreto, es un código que es interpretado por una máquina virtual que lo hace totalmente independiente del hardware en el que se ejecute ésta máquina virtual. Este código se denomina normalmente `bytecode` y la máquina virtual es conocida como JVM.

El `bytecode` generado por la compilación de un programa en Java es exactamente igual, independientemente de la plataforma en la que se ha generado, y por ello es posible, por ejemplo, compilar un programa en una máquina Sun basada en tecnología Sparc y después ejecutar el programa en una máquina Linux basada en tecnología Intel. En el proceso de ejecución es la JVM la que toma ese `bytecode`, lo interpreta y lo ejecuta teniendo en cuenta las particularidades del sistema operativo. El resultado: total portabilidad del código que generamos. Evidentemente esta capacidad de Java lo hace muy útil en entornos de ejecución heterogéneos como es Internet.

El conjunto de herramientas utilizadas en la compilación de un programa Java se conocen genéricamente como SDK *(Software Development Kit)* o entorno de desarrollo; la máquina virtual Java y todas las clases necesarias para la ejecución de un programa se conoce como JRE *(Java Runtime Environment)*. El SDK de Java contiene el JRE, no tendría sentido poder compilar un programa y no poder ejecutarlo después; pero el JRE se distribuye por separado, existen personas que sólo quieren ejecutar programas y no les interesa el desarrollo.

Existen versiones del SDK para plataformas Solaris, Windows, Linux y Mac principalmente, pero es posible encontrar máquinas virtuales en otros muchos sistemas operativos, recordemos que Java se diseñó pensando en ejecutar programas en entornos heterogéneos.

## Compilación

El proceso de compilación en Java es similar al de otros lenguajes de programación; la principal diferencia es que en lugar de generar código dependiente de un determinado procesador, como haría un compilador de C++ por ejemplo, genera código para un procesador que no existe realmente, es virtual: la JVM.

Una característica del SDK de Java que sorprende a los programadores que se topan la primera vez con Java es que el SDK de Java no tiene un entorno gráfico, ni tan siquiera un entorno para editar los programas y compilarlos, todo se hace desde la línea de comandos, a la manera tradicional. Esto al principio es engorroso e incómodo pero tiene un beneficio claro: sólo debemos preocuparnos por conocer Java, no es necesario gastar tiempo en entender un entorno de desarrollo complejo antes de incluso saber escribir nuestro primer programa. Con el tiempo, y cuando dominemos el lenguaje, podremos elegir entre los entornos gráficos y no gráficos para desarrollar en Java. Para escribir un programa en Java, incluso con los gráficos más complejos, sólo es necesario un editor de texto y el SDK.

Para poder compilar nuestro primer programa en Java necesitamos escribirlo; y como no podía ser de otra forma será sencillo y sólo mostrará un mensaje de bienvenida a la programación en Java. A pesar de que el programa es de sólo cinco líneas, encierra conceptos importantes, la mayoría de ellos deben ser explicados posteriormente; pero vamos a ver por encima la estructura del programa sabiendo que algunos puntos del programa quedarán oscuros en este momento.

```
public class Inicio {
     public static void main(String [] args){
        System.out.println("¡Bienvenido a Java!");
     }
}
```

En primer lugar debemos escribir el programa con nuestro editor de textos favorito y guardarlo en un fichero con el nombre `Inicio.java`. Es necesario que el nombre del fichero coincida con el de la clase siempre que ésta sea una `public`. Dentro de un fichero `.java` (unidad de compilación) pueden aparecer tantas clases como queramos; pero sólo una de ellas puede ser `public`.

En Java todo son clases y si queremos hacer un programa que sólo escriba un mensaje por pantalla debemos escribir una clase, en nuestro caso la clase se llama `Inicio`. Es importante resaltar que Java es sensible a mayúsculas y minúsculas, y por tanto, la clase `Inicio` es distinta de la clase `inicio`. Una clase se define utilizando la palabra reservada `class` y comprende todo el código encerrado entre las dos llaves más externas.

Dentro de la clase `Inicio` vemos el método `main`. Este método es especial. Por él comienza la ejecución de cualquier programa en Java, siempre tiene esta estructura y es conveniente res-

petarla para evitar problemas. Sin entrar en muchos detalles vamos a comentar brevemente cada una de las partes del método `main`.

- **`public`**. Indica que el método es público y que puede accederse desde fuera de la clase que lo define. El método `main` tiene que ser invocado por la máquina virtual Java, por lo que debe ser público.
- **`static`**. Indica que el método es estático y que no es necesario que exista una instancia de la clase `Inicio` para poder ser ejecutado. Esto también es necesario ya que como hemos dicho el método es llamado desde la JVM.
- **`void`**. Informa al compilador de que el método `main` no devuelve ningún valor tras su ejecución.
- **`String[] args`**. Define un parámetro del método `main`. Utilizaremos los parámetros para enviar y recoger información de los métodos. En este caso `args` contendrá los posibles parámetros de la línea de comando de la ejecución de la clase `Inicio`.

Lógicamente toda esta información es confusa en este momento, no se preocupe, no es necesario comprender totalmente todos estos conceptos para continuar. Algunas veces, es necesario introducir conceptos que no es posible explicar hasta más adelante con el fin de poder continuar. En este momento, es necesario escribir un programa en Java sin saber nada de Java. Es lógico que no se entienda todo.

Por último, dentro del método `main` nos encontramos una línea de código que será ejecutada cuando se invoque el método. Nos limitaremos a decir que esa línea permite imprimir en la salida estándar, normalmente el monitor de nuestro computador, el mensaje `"¡Bienvenido a Java!"`. Para ello utiliza el método `println` del objeto `out` de la clase `System`. Una vez más todo esto se explicará más adelante.

Para compilar utilizaremos el compilador `javac`. La ejecución de `javac` dependerá del sistema operativo en el que estemos trabajando, pero en general se realizará desde la línea de comandos de nuestro sistema operativo.

```
En Windows
C:\> javac Inicio.java
En Solaris/GNU Linux/Unix
$ javac Inicio.java
```

Si no se encuentran errores en la compilación, el resultado de ésta será un fichero con el `bytecode` correspondiente a la compilación de `Inicio.java`; este resultado se almacena en un fichero con extensión `.class` que tiene como nombre el mismo que el fichero fuente. Por tanto, en nuestro directorio tendremos un fichero llamado `Inicio.class`. Ya tenemos compilado nuestro primer programa en Java, ahora tenemos que ejecutarlo.

## Ejecución

La ejecución de un programa Java involucra a la máquina virtual que es la encargada de interpretar y ejecutar cada una de las instrucciones (`bytecode`) contenidas en el fichero `.class`.

Para ejecutar el programa tenemos que utilizar el entorno de ejecución de Java *(Java Runtime Environment*, JRE). El JRE nos permite ejecutar el `bytecode` de nuestros programas en la máquina virtual Java. Para ejecutar un programa en Java tenemos que invocar el entorno de ejecución pasándole como parámetro el nombre de la clase que queremos ejecutar.

**En Windows**

```
C:> java Inicio
```

**En Solaris/GNU Linux/Unix**

```
$ java Inicio
```

Si todo ha ido bien veremos el resultado de la ejecución después de la ejecución del programa. Es necesario que nos encontremos en el mismo directorio que contiene el fichero `.class` para que la JVM lo encuentre. También es importante recordar que no debemos poner la extensión del fichero, ya que no estamos indicando el nombre de un fichero sino el nombre de una clase, la máquina virtual Java se encargará de encontrar el fichero con el `bytecode`.

## 1.2. Fundamentos del lenguaje

Cuando se comienza a estudiar un nuevo lenguaje de programación es necesario ver dos bloques fundamentales. Por un lado necesitamos conocer qué datos es capaz de manejar, qué posibilidades de manejo de esos datos nos proporciona y por otro lado, qué herramientas para controlar la ejecución y la interacción con el usuario nos ofrece. En esta sección vamos a ver todo esto, pero al contrario que en otros lenguajes, en Java no haremos nada más que empezar a ver sus posibilidades. Tendremos que llegar a conocer y dominar otros muchos elementos, relacionados con la programación orientada a objetos la mayoría, antes de poder decir que lo conocemos.

Comenzaremos viendo la forma de representar y manejar la información en Java, después descubriremos cómo podemos controlar el flujo de ejecución de nuestros programas y finalizaremos con unos breves conceptos de entrada/salida a consola y métodos.

### 1.2.1. Tipos básicos

Java es un lenguaje fuertemente tipado, todas las variables que se definen tienen un tipo declarado y este tipo es controlado y comprobado en todas las operaciones y expresiones. A pesar de que en algunos momentos tantas comprobaciones pueden ser un poco frustrantes, éstas hacen de Java un lenguaje muy poco propenso a errores exotéricos, derivados de un tipo mal expresado o incorrectamente usado, que en otros lenguajes se producen con relativa frecuencia.

Disponemos de ocho tipos básicos divididos en cuatro bloques dependiendo de su naturaleza. En un principio puede parecer que un lenguaje tan completo como Java debería tener más tipos de datos, pero estos ocho tipos cubren perfectamente las necesidades de representación de la información, ya que sirven de base para crear estructuras más complejas y potentes.

Los bloques en los que se encuentran divididos los tipos básicos en Java son los siguientes:

- Enteros. Son cuatro tipos que nos permiten representar números enteros.
- Coma flotante. Son dos tipos usados para representar datos reales.
- Caracteres. Un tipo que nos permite representar caracteres de cualquier idioma mundial.
- Lógicos. Un tipo para representar valores lógicos.

A diferencia de lo que ocurre en otros lenguajes, los tipos básicos en Java siempre tienen los mismos tamaños y capacidades, independientemente del entorno en el que estemos trabajando. Esto garantiza que no será necesario comprobar la arquitectura en la que nos encontramos para

**Tabla 1.1.** Tipos enteros.

| Nombre | Tamaño | Rango |
|---|---|---|
| `long` | 64 bits | –9.233.372.036.854.775.808L a 9.233.372.036.854.775.807L |
| `int` | 32 bits | –2.147.483.648 a 2.147.483.647 |
| `short` | 16 bits | –32.768 a 32767 |
| `byte` | 8 bits | –128 a 127 |

decidirnos por un tamaño de entero o por otro, un tipo `int` tendrá 32 bits en un PC y en una estación Sun.

## Enteros

Los números enteros en Java son siempre con signo, no existen tipos enteros sin signo ni modificadores para eliminarlo. Los cuatro tipos enteros: `byte`, `short`, `int` y `long`, se muestran en la Tabla 1.1. con su tamaño y su rango de valores representables.

A diferencia de lo que ocurre en otros lenguajes, los tipos básicos en Java siempre tienen las mismas capacidades de almacenamiento, independientemente del entorno en el que estemos trabajando. Esto garantiza que no será necesario comprobar la arquitectura en la que nos encontramos para decidirnos por un tamaño de entero o por otro.

Por defecto las constantes enteras son de tipo `int`. Si queremos forzar que una constante de tipo entero sea tomada como un `long` debemos añadir al final una L.

**Tabla 1.2.** Tipos en coma flotante.

| Nombre | Tamaño | Rango |
|---|---|---|
| `float` | 32 bits | ±3.40282347E+38F |
| `Double` | 64 bits | ± 1.79769313486231570E+308 |

## Coma flotante

Los dos tipos utilizados en Java para representar valores reales son: `float` y `double`, en la Tabla 1.2. podemos ver sus características de almacenamiento.

El tipo `double` es, generalmente, más usado que el `float`, pero éste es un poco más rápido en las operaciones y ocupa menos, por lo que puede ser muy útil en algunas circunstancias en las que la velocidad de cálculo sea prioritaria.

Al igual que las constantes enteras son por defecto de tipo `int`, las constantes reales son por defecto de tipo `double`. Podemos forzar un tipo `float` si añadimos al final del número una F.

Los tipos `float` y `double` disponen de tres valores especiales: infinito positivo, infinito negativo y NaN *(Not a number)*. Estos valores nos permiten representar situaciones como desbordamientos y errores.

```
public class Rangos {
  public static void main(String [] args){
    System.out.println(Math.sqrt(-1));
    System.out.println(1.1e200*1.1e200);
    System.out.println(-1.1e200*1.1e200);
  }
}
```

La ejecución del programa anterior da como resultado la impresión de los tres valores especiales. El método `sqrt()` de la clase `Math` nos permite calcular la raíz cuadrada de un número.

```
NaN
Infinity
-Infinity
```

## Caracteres

En Java los caracteres no se almacenan en un byte como en la mayoría de los lenguajes de programación. En Java se usa Unicode para representar los caracteres y por ello se emplean 16 bits para almacenar cada carácter. Al utilizar dos bytes para almacenar cada carácter, disponemos de un total de 65.535 posibilidades, suficiente para representar todos los caracteres de todos los len-

**Tabla 1.3.** Secuencias de escape.

| Secuencia | Descripción |
|---|---|
| \b | Retroceso |
| \t | Tabulador |
| \r | Retorno de carro |
| \n | Nueva línea |
| \’ | Comilla simple |
| \” | Comilla doble |
| \\ | Barra invertida |

guajes del planeta. El estándar ASCII/ANSI es un subconjunto de Unicode y ocupa las primeras 256 posiciones de la tabla de códigos, con lo que es posible la compatibilidad entre los dos sistemas de representación.

Muchas veces es necesario representar caracteres en forma de constante. En Java las constantes de tipo carácter se representan entre comillas simples. Existen secuencias de escape para representar algunos caracteres especiales como el retorno de carro, el tabulador, etc. como se muestra en la Tabla 1.3.

Podemos también expresar caracteres a través de su código Unicode o su código ASCII/ANSI tradicional. Para ello utilizamos ’uxxxx’ donde xxxx es el código Unicode del carácter en hexadecimal. También podemos utilizar el código ASCII/ANSI en octal de la forma ’ooo’ o en hexadecimal con la expresión ’0xnn’.

```
‘A’   ‘\u0041’   ‘\0x41’   ‘\101’
```

## Lógicos

En Java existe un tipo especial para representar valores lógicos, el tipo `boolean`. Este tipo de datos sólo puede tomar dos valores: verdadero y falso. El tipo `boolean` se emplea en todas las estucturas condicionales y es el resultado de las operaciones realizadas utilizando operadores relacionales.

Existen dos palabras reservadas para representar los valores lógicos, `true` y `false`, que pueden utilizarse libremente en los programas. A diferencia de otros lenguajes, el tipo `boolean` es un tipo distinto de los demás y, por tanto, incompatible con el resto. Java es rígido en esto y si espera un tipo `boolean` no aceptará un `int` en su lugar.

```
boolean b;
```

```
b=true;
if (b) System.out.println("Es cierto");
```

## Envoltorios

**Tabla 1.4.** Envoltorios.

| Tipo | Envoltorio |
|---|---|
| `int` | `Integer` |
| `long` | `Long` |
| `float` | `Float` |
| `double` | `Double` |
| `short` | `Short` |
| `byte` | `Byte` |
| `char` | `Character` |
| `boolean` | `Boolean` |
| `void` | `Void` |

En Java todo son clases y objetos, excepto los tipos básicos. Esto hace que en algunas circunstancias tengamos que convertir estos tipos básicos en objetos. Para realizar esta conversión utilizamos envoltorios que recubren el tipo básico con una clase, a partir de este momento el tipo básico envuelto se convierte en un objeto.

Existen nueve envoltorios para los tipos básicos de Java, como se puede ver en la Tabla 1.4., cada uno de ellos envuelve un tipo básico y nos permite trabajar con objetos en lugar de con tipos básicos.

Una de las razones más importantes para utilizar envoltorios es poder emplear las clases de utilidad que Java proporciona en su bilbioteca de clases. Estas clases necesitan utilizar objetos para funcionar y no aceptan tipos de datos básicos. Si queremos por ejemplo crear una pila de números reales, tendremos que envolver los números reales para tener objetos que poder guardar en la pila.

En los envoltorios existen algunos métodos que nos permiten convertir cadenas de caracteres en tipos básicos. Así, podemos convertir la cadena "123" en el número entero 123 utilizando el método `parseInt()` de la clase `Integer`.

```
int num=Integer.parseInt(“123”);
```

En Java 2, dentro de cada clase envoltorio, excepto `Boolean`, `Character` y `Void`, existe un método `parse` que nos permite convertir una cadena en el tipo básico correspondiente.

```
public class Envoltorios {
  public static void main(String [] args){
    System.out.println(Integer.parseInt(“124”));
    System.out.println(Long.parseLong(“165”));
    System.out.println(Byte.parseByte(“21”));
    System.out.println(Short.parseShort(“45”));
    System.out.println(Float.parseFloat(“45.89”));
    System.out.println(Double.parseDouble(“1.5e8”));
  }
}
```

Los tipos básicos envueltos por las clases proporcionadas por Java son inmutables, es decir, que no pueden modificar su valor sin destruir el objeto. Existen situaciones, como se verá más tarde, en las que es necesario cambiar este comportamiento y deberemos definir nuestros propios envoltorios.

**Tabla 1.5.** Tipos de literal.

| Tipo | Literal | Comentarios |
|---|---|---|
| `int` | 123 | Todos los enteros por defecto son `int` |
| `long` | 123L | Es necesario indicar una L |
| `char` | ‘a’ | Comillas simples |
| `float` | 5.9F | Es posible usar también la notación exponencial 1.8E9 |
| `double` | 7.9 | Todos los reales por defecto son `double`. Se pueden finalizar con una D |
| `boolean` | true | `true` y `false` son los únicos valores válidos |
| `String` | “hola” | Comillas dobles |

## 1.2.2. Literales y constantes

Un literal es un dato que se considera constante y que es expresado de diferentes formas dependiendo de su naturaleza con el fin de evitar ambigüedades. En Java disponemos de tantos tipos de literales como tipos de datos básicos. Además podemos escribir literales de tipo cadena de

caracteres, que internamente Java convierte al tipo `String`, utilizado en Java para manejar las cadenas. Todos los tipos de literal se encuentran en la Tabla 1.5.

Es posible utilizar constantes numéricas expresadas en octal y en hexadecimal. Para indicar que una constante está expresada en octal, ésta debe comenzar por 0, si comienza por 0x o por 0X será una constante hexadecimal.

```
Octal          015
Decimal         13
Hexadecimal  0x0D
```

Existe la posibilidad de definir constantes mediante el uso de la palabra reservada `final` situada antes del tipo en la definición de una variable. Al utilizar `final` en la declaración hacemos que esta variable únicamente pueda cambiar su valor una vez y, por tanto, la convertimos en constante.

```
final double PI=3.141592653589793;
```

En Java las constantes se suelen escribir en mayúsculas para diferenciarlas de las variables.

## 1.2.3. Variables

La forma más sencilla de almacenar información en Java es utilizar variables. Posteriormente veremos que disponemos de elementos más complejos para representar información, pero las variables son la base de todos ellos.

En Java antes de usar cualquier variable, independientemente de su tipo, es necesario declararla. Desde ese momento puede ser usada sin más restricciones que las impuestas por su tipo, su ámbito y su tiempo de vida, características todas ellas que exploraremos seguidamente.

### Declaración

La estricta comprobación de los tipos en Java hace que la declaración de variables sea obligatoria; ésta puede realizarse en cualquier parte de una clase o método (su denominación cambia pero son variables en cualquier caso) y a partir de ese momento la variable podrá se utilizada.

La forma de declaración de una variable en Java básicamente indica el nombre y el tipo de la misma, pero puede ir acompañada de más información, como el valor inicial o la declaración de más variables del mismo tipo.

```
tipo identificador[=valor][,identificador[=valor]...];
```

En la declaración anterior, `tipo` es uno de los tipos legales de Java, es decir, tipos básicos, clases o interfaces. Estos dos últimos se verán más adelante. El identificador tiene que ser único y puede contener cualquier carácter UNICODE.

Es posible inicializar la variable con un valor distinto del que Java emplea por defecto. Este valor debe ser del mismo tipo que la variable o un tipo compatible. El valor de la inicialización

puede ser cualquier expresión válida, no tiene por qué ser una constante, es, por tanto, legal utilizar una expresión cuyo valor no sea conocido en tiempo de compilación.

```
int i=0,j;
double d = Math.sqrt(i*5);
```

### Ámbito y tiempo de vida

Todas las variables tienen dos características que definen su comportamiento: su ámbito y su tiempo de vida. Generalmente, estos dos conceptos van unidos y no es posible entender el uno sin el otro, pero son dos características diferentes.

La declaración de una variable se debe producir dentro de un bloque de código y ese bloque de código determina su ámbito, es decir, en qué parte del código la variable puede ser accedida. Un bloque es una porción de código encerrado entre dos llaves ( `{` y `}` ). Hemos visto bloques de código en algunos ejemplos anteriores, cuando definíamos una clase o cuando definíamos el método `main`; pero no son los únicos lugares donde aparecen bloques, la mayor parte de las veces aparecen unidos a sentencias de control, pero pueden aparecer solos para delimitar un fragmento de código o más concretamente un ámbito. Es posible anidar bloques y, por tanto, ámbitos.

```
1. {
2.     int a;
3.     a=9;
4.        {
5.            int b=a+1;
6.        }
7.          a=10;
8. }
```

Vemos en el ejemplo anterior que la variable `a` está definida en el bloque que comienza en la línea 1 y finaliza en la línea 8, por tanto, es legal acceder a la variable `a` en la línea 5. La variable `b` se define dentro del bloque de las líneas 4, 5 y 6, si intentamos acceder a su contenido en la línea 7 se producirá un error ya que su ámbito no llega a la línea 7 del programa.

Dada la naturaleza orientada a objetos de Java, no existe el concepto de ámbito global y local como en otros lenguajes. Existen otros ámbitos más adaptados a la programación orientada a objetos como son el ámbito de clase y el de método; no son los únicos pero si los más claros en este momento. Es sencillo deducir que el ámbito de clase corresponde con las líneas de código de una clase y el ámbito de método con las de un método.

El tiempo de vida de una variable es el tiempo (código) que transcurre entre la declaración de la variable y su destrucción. Generalmente, coincide con el ámbito pero existen variables cuya vida no finaliza con la llave que cierra el ámbito donde fueron definidas.

## 1.2.4. Conversión de tipos

Cuando se evalúa una expresión en la que se mezclan datos con distintos tipos, es necesario realizar conversiones de tipo con el fin de que las operaciones se realicen de una forma coherente; en algunos casos estas conversiones son sencillas ya que afectan a tipos que tienen características comunes, como pueden ser dos tipos enteros; pero en otras ocasiones esto no es tan sencillo y es necesario tomar decisiones que afectan a la fiabilidad de los datos involucrados, por ejemplo, cuando tenemos que convertir un número real en un entero.

Por todo ello, los procesos de conversión de tipo son bastante complejos y no siempre pueden ser automáticos. Cuando es posible realizar la conversión de forma automática, Java la realiza. En los demás casos, no es posible realizar la conversión sin que se pierda información, y es necesario que sea forzada por el programador.

## Conversión automática

La conversión automática es la deseable, en la mayoría de los casos el programador no quiere tener que preocuparse por cambios en los tamaños de los enteros, sólo quiere sumar dos enteros. Existen dos reglas básicas para determinar si se puede realizar una conversión automática de tipos en Java:

- Los dos tipos son compatibles.
- El tipo destino es más grande que el tipo origen.

Cuando se cumplen estas dos condiciones se produce una promoción del tipo origen para adaptarlo al tipo destino. En esta promoción nunca se pierde información, por lo que Java puede realizarlo sin problemas y sin que el programador tenga que indicarlo. Es fácil aumentar el tamaño de un `byte` para convertirlo en un `int`.

Las reglas de compatibilidad son :

- Todos los tipos numéricos son compatibles entre sí, sin importar que sean enteros o reales.
- El tipo `char` es compatible con `int`.
- El tipo `boolean` no es compatible con ningún otro tipo.

Cuando se inicializa una variable `long`, `short` o `byte` con una constante entera, Java realiza la conversión de esa constante entera al tipo destino de forma automática. En este caso la constante entera debe estar en el rango del tipo destino, de no ser así se producirá un error de compilación.

Para ilustrar mejor el comportamiento de Java en las conversiones vamos a estudiar algunos ejemplos, para ello definimos algunas variables y las inicializamos convenientemente.

```
char c='a',c2;
int i=23,i2;
short s;
double d;
```

Veamos las asignaciones:

```
i2=c;
```

La asignación es correcta, los tipos de `i2` y `c` son compatibles y el valor de `c` cabe en el tipo de `i2`.

```
s=c;
```

Esta asignación es incorrecta, recordemos que en Java el tipo `char` utiliza el mismo número de bits que el `short` para guardar la información, pero en el tipo `short` hay signo lo que le impide representar todos los valores posibles de un `char`. En este caso necesitaremos utilizar otro tipo de conversión que veremos a continuación.

```
d=c;
```

En este caso la asignación sí es correcta, los dos tipos son compatibles y el origen cabe en el destino.

```
s=678;
```

El literal entero `678` se convierte automáticamente a `short` y como está dentro del rango permitido se realiza la asignación.

## Conversión explícita. `Casting`

En los casos en los que la conversión de tipos no puede llevarse a cabo de forma automática, pero necesitamos que la conversión se realice, tenemos que forzar el cambio utilizando una conversión explícita o `casting`. Éste se realiza anteponiendo al dato que queremos cambiar el tipo destino encerrado entre paréntesis.

Utilizando `casting` podemos forzar las conversiones a pesar de que se puede perder información en el cambio. Esta posible pérdida de información es la razón de que no sea automática, Java nos cede la responsabilidad de decidir si queremos o no sacrificar parte en el cambio.

La conversión se realizará siguiendo unas sencillas reglas:

- Entre números enteros, si el destino es mayor que el origen, el valor resultante será el resto (módulo) de la división entera del valor con el rango del tipo destino.
- Si el origen es un número real y el destino un entero, la parte decimal se trunca, además si la parte entera restante no cabe en el destino, se aplica en criterio del módulo.
- Entre números reales, se guarda el máximo valor posible.

Veamos unos ejemplos de conversiones que requieren `casting`:

```
double dou=123.67;
int dest=(int) dou;
```

La variable `dest` contendrá `123` después de la asignación ya que se trunca la parte decimal al pasar de un número real a un número entero.

```
dou=3.40282347E+50;
```

```
float fl=(float) dou;
```

En este caso el resultado es un poco más sorprendente, pasamos de un tipo `double` a un tipo `float` y el valor que queremos guardar en la variable `fl` supera el rango de un tipo `float`. El resultado de la asignación es un valor infinito, y para representarlo utiliza el valor especial `Infinity` como veíamos anteriormente.

```
int in=257;
byte b;
b=(byte) in;
```

Después de la asignación, la variable `b` contendrá 1 ya que se aplica el resto de la división entera para calcularlo y `257 mod 256 = 1`.

### Promoción en expresiones

Al evaluar una expresión, tambien se producen conversiones de tipo si es necesario. La regla general cuando esto sucede es que Java convierta automáticamente los operandos al tipo mayor de los presentes en la expresión y utilizará este tipo para evaluar la expresión. Si, por ejemplo, se va a realizar una suma entre un tipo `double` y un tipo `int`, el `int` será promocionado a `double` y la operación será realizada entre dos `double`. El resultado, por tanto, será un `double`.

Las reglas que controlan estas promociones son las siguientes:

- `byte` y `short` se promocionan a `int`.
- Si un valor es `long` la expresión se promociona a `long`.
- Si un valor es `float` la expresión se promociona a `float`.
- Si un valor es `double` la expresión se promociona a `double`.
- Un valor `char` en una expresión numérica se promociona a `int`.

Vamos a realizar una operación utilizando tipos `byte`.

```
byte b,b2,b3;
b2=10;
b3=100;

b=(byte)(b2*b3);
```

A pesar de que `b2` y `b3` son del mismo tipo que `b`, la operación se realiza como si fueran `int` y el resultado de la misma es lógicamente un `int`. Como consecuencia, necesitamos utilizar un `casting` para que la asignación se realice sin problemas.

## 1.2.5. Uso básico de cadenas de caracteres

En Java no existe un tipo básico para almacenar cadenas de caracteres, a pesar de que es uno de los tipos más usados en la mayoría de los programas; en su defecto se utiliza una clase de la biblioteca estándar, la clase `String`. Cada vez que necesitamos una cadena de caracteres instanciamos un objeto de la clase `String`. El concepto de instanciación, de objetos y clases se verá posteriormente. Instanciar un objeto es muy parecido (visto desde fuera) a declarar una variable de un tipo básico.

Al no ser el tipo `String` un tipo básico hace que el manejo de cadenas en toda su potencia requiera manejar clases, objetos y métodos, y por ello vamos a ver cómo utilizar cadenas de forma muy simple y posteriormente ampliaremos la información.

Es posible definir literales de tipo cadena entrecomillando texto con comillas dobles, imprimir con `System.out.println()` estas cadenas e incluso concatenarlas usando el operador `+`.

```
System.out.println("Hola"+" mundo");
```

También podemos definir "variables" de tipo cadena (realmente son objetos) y usarlas para guardar cadenas de caracteres y realizar con ellas operaciones básicas de asignación y concatenación.

```
1. String a,b;
2.
3. a="Hola";
4. b=" mundo";
5. String c=a+b;
6. System.out.println(c);
```

En el ejemplo podemos ver en la línea 1 que la declaración de una "variable" de tipo `String` es similar a la declaración de cualquier variable de tipo básico. En las líneas 3 y 4 se realizan asignaciones de literales a las variables definidas. Las líneas 5 y 6 crean una nueva cadena como resultado de la concatenación de `a` y `b` e imprimen el resultado respectivamente.

Como se puede ver en el ejemplo anterior, el tratamiento de cadenas es sencillo en Java, pero no se debe perder de vista que no estamos tratando con tipos básicos y que como se verá posteriormente las operaciones que se realizan con cadenas implican operaciones entre objetos más complicadas que las realizadas con tipos básicos.

## 1.2.6. Arrays

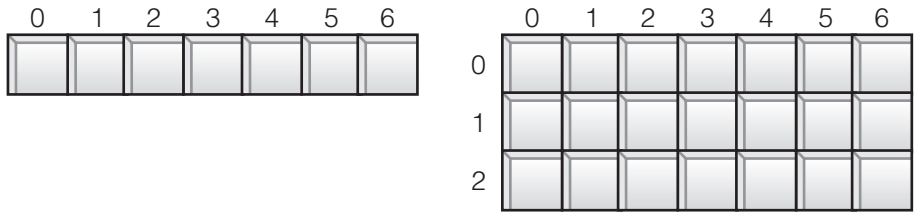



**Figura 1.1.** Arrays de una y dos dimensiones.

En ocasiones nos encontramos con un conjunto de datos homogéneos y relacionados entre sí que debemos almacenar. Con lo que hemos visto hasta ahora, la única solución para almacenar esta información es declarando variables independientes y guardando en cada una de ellas un dato. Con un número de variables pequeño esta solución es relativamente buena, pero qué ocurre si tenemos que manejar los datos referentes a las horas de luz en Madrid durante un mes; tendríamos que definir una treintena de variables y el programa se volvería complicado de leer y muy propenso a errores. Para poder afrontar este tipo de problemas necesitamos un tipo de datos más poderoso que los tipos básicos. Necesitamos un array.

Un array (o matriz) es un conjunto de datos homogéneos que ocupan posiciones de memoria contiguas y que es posible referenciar a través de un nombre único. Cada uno de los elementos de los que está compuesto el array es direccionable individualmente a través del nombre de ésta y un índice que indica el desplazamiento a lo largo de los elementos de los que está compuesta.

Podemos definir arrays de cualquier tipo de datos, sea éste un tipo básico o como veremos posteriormente tipos más complejos. También podemos aumentar el número de dimensiones y crear tablas o cubos. No existe una limitación en el número de dimensiones que puede tener un array en Java, pero trabajar con estructuras de más de cuatro dimensiones es algo reservado a muy pocos.

Utilizando arrays, manejar un conjunto de datos relacionados se simplifica y es posible mantener unidos los datos de las horas de luz como veíamos antes, o relacionar las ventas mensuales de varios comerciales en forma de tabla.

## Declaración de arrays de una dimensión

En Java la declaración de arrays es un poco distinta que en otros lenguajes ya que consta de dos partes; por un lado tenemos que declarar una referencia al array que queremos crear. Una vez que tenemos la referencia, asociarle el array, es decir, primero necesitamos una variable para "apuntar" a los datos del array y después reservamos la memoria necesaria para almacenar esos datos.

La declaración de la referencia del array se realiza especificando el tipo del array seguido de `[]`, despúes viene el nombre de la variable.

```
int[] a;
```

Una vez que tenemos la refencia, tenemos que reservar la memoria para almacenar el array, para ello utilizamos el operador `new`. Usaremos `new` cuando queramos crear un nuevo elemento de forma dinámica dentro de nuestros programas, en este momento necesitamos crear un array, posteriormente crearemos objetos. El operador `new` debe ir seguido del tipo del array y entre corchetes la dimensión (número de elementos) del mismo.

```
a=new int[10];
```

Esta sentencia crea un array de diez enteros y éste queda referenciado por la variable `a`. A partir de este momento la variable `a` apunta al array creado y la utilizaremos para acceder al contenido del array.

Es posible realizar las dos sentencias en una, y además existen dos formas de declarar un array. También es posible que el tamaño del array en el momento de la definición no sea conocido en tiempo de compilación.

Como se puede ver en los siguientes ejemplos, la línea 1 declara y define un array de veinte números reales, referenciados por la variable `arr`. En las líneas 3 y 4, creamos el array `arr2` de la misma forma, pero en este caso, el tamaño del mismo viene dado por el valor de una variable lo que hace que su valor no sea conocido hasta que se produzca la ejecución del programa.

```
1. float[] arr=new float[20];
2.
3. int tam=9;
4. float arr2[]=new float[tam];
```

Cuando creamos un array, Java por defecto inicializa todas sus posiciones a 0. Esta inicialización es, además de un trabajo que no tenemos que hacer, una medida de seguridad que evita problemas al acceder a posiciones en las que puede existir cualquier valor existente previamente en la memoria. No obstante, la inicialización por defecto no es siempre la mejor opción y por eso es posible a la hora de declarar el array darle valores por defecto distintos de cero, como podemos ver en la siguiente declaración.

```
int[] numeros={1,2,3,4,5,6,7,8,9};
```

El array `numeros` contendrá, al finalizar su creación, los números del 1 al 9. Como se puede ver, en la declaración de este array no se ha indicado su tamaño, Java es capaz de calcular este dato a partir del número de elementos que encuentre para la inicialización. En este caso, creará un array de nueve posiciones.

El tamaño de un array se indica en el momento de su definición y no puede modificarse posteriormente. Si necesitamos que el array cambie de tamaño, debemos crear un nuevo array y traspasarle la información del array inicial. Es posible conocer el número de elementos de un array utilizando la propiedad `length` como vemos en el siguiente ejemplo que imprimirá en la salida estándar un 25.

```
int[] a=new int[25];
System.out.println(a.length);
```

## Acceso a un array de una dimensión

El acceso al array se realiza, normalmente, a través de la referencia usada en su declaración. Es posible acceder al array en su totalidad, utilizando sólo la referencia del array, o a cada uno de los elementos que lo constituyen mediante un índice único, en este caso se añade el índice a la referencia encerrándolo entre corchetes. El índice del array es un número entero comprendido

entre 0 y la dimensión –1. Java se encarga de comprobar que todos los accesos que se realizan estén comprendidos en ese rango, cualquier intento de acceso fuera de los valores permitidos provocará un error indicado en forma de excepción.

En el siguiente ejemplo podemos ver un sencillo ejemplo de acceso a un array de enteros declarado y definido en la línea 1 del programa.

```
1. int[] a=new int[10];
2.
3. a[0]=1;
4. a[2]=5;
5. a[4]=6;
6.
7. System.out.println(a[2]);
8. System.out.println(a[4]);
9. System.out.println(a[6]);
```

Las líneas 3, 4 y 5 cambian los valores de las posiciones 0, 2 y 4 del array cambiando los ceros de la inicialización por defecto por otros valores. En las líneas 7, 8 y 9 los contendidos de las posiciones 2, 4 y 6 son mostrados por pantalla. El resultado de la ejecución de este fragmento de código se muestra a continuación.

```
5
6
0
```

El último número mostrado es un 0 ya que la posición 6 del array no ha sido cambiada desde su declaración y conserva la inicialización por defecto. Las posiciones 2 y 4 contienen los valores guardados en las asignaciones de las líneas 4 y 5.

Dado que los arrays se manejan utilizando referencias, si asignamos un array a otro, no copiamos los valores de uno en otro, lo que hacemos es apuntar las dos referencias al mismo array.

```
1. int a[]={1,2,3};
2. int b[]={2,4,6};
3. b=a;
4. System.out.println(b[1]);
5. a[1]=99;
6. System.out.println(b[1]);
```

Tras la asignación de la línea 3, `a` y `b` apuntan al mismo array, el que contiene los datos `{1, 2, 3}`, el array al que apuntaba la variable `b`, se pierde y es eliminado por el recolector de basura de Java. El `println` de la línea 4 imprimirá un 2, pero si realizamos una asignación como la de la línea 5 y volvemos a imprimir el contenido de la posición 1 del array `b`, veremos que ha cambiado y se imprimirá un 99. Esto es debido a que `a` y `b` apuntan al mismo conjunto de datos, no hemos realizado un copia de un array en otro.

Si lo que queremos es copiar los valores de un array en otro y que los dos arrays sean independientes, deberemos recorrerlos y copiar cada uno de los elementos de un array en el otro. De este modo, los datos del array origen y los del array destino no ocuparán el mismo lugar de almacenamiento, siendo, por tanto, independientes.

En el siguiente ejemplo vemos cómo copiar un array a otro sin tener problemas.

```
public class  CopiaArraysFor{
   public static void main(String [] args)       {
      int [] origen={1,3,5,7};
      int [] destino= new int[origen.length];
      for (int i=0; i<origen.length ; i++ )
         destino[i]=origen[i];
      System.out.println("Después de la copia");
      System.out.print("origen=[ ");
      for (int i=0; i<origen.length ; i++ )
         System.out.print(origen[i]+" ");
      System.out.println("]");
      System.out.print("destino=[ ");
      for (int i=0; i<destino.length ; i++ )
         System.out.print(destino[i]+" ");
      System.out.println("]");
      origen[1]=99;

      System.out.println("\nDespués de la asignación");
      System.out.print("origen=[ ");
      for (int i=0; i<origen.length ; i++ )
         System.out.print(origen[i]+" ");
      System.out.println("]");
      System.out.print("destino=[ ");
      for (int i=0; i<destino.length ; i++ )
         System.out.print(destino[i]+" ");
      System.out.println("]");
   }
}
```

El resultado de la ejecución del programa anterior se muestra a continuación. Tras realizar la copia del array origen en el array destino, ambos arrays contienen la misma información, pero de forma independiente, de tal forma que si modificamos uno, el otro no se ve afectado.

```
Despues de la copia
origen=[ 1 3 5 7 ]
destino=[ 1 3 5 7 ]
Despues de la asignación
origen=[ 1 99 5 7 ]
destino=[ 1 3 5 7 ]
```

Dado que copiar un array en otro es una tarea muy común, existe un método en la clase `System` que podemos usar para copiar los valores de un array en otro.

```
System.arraycopy(origen,indiceOrigen,destino,indiceDestino,num);
```

Este método copia desde el índice `indiceOrigen` del array origen al array destino, comenzando en la posición `indiceDestino copia num elementos`. El array destino debe tener suficiente espacio para la copia.

```
public class CopiaArray {
   public static void main(String [] args) {
      int[] origen=new int[10];
      int[] destino={1,1,1,1,1};
      origen=new int[] {2,2,2};  // Array anónimo

      System.arraycopy(origen,0,destino,1,origen.length);
      for(int i=0;i<destino.length;i++)
          System.out.println("destino["+i+"]="+destino[i]);
   }
}
```

El resultado de la ejecución del programa será:

```
destino[0]=1
destino[1]=2
destino[2]=2
destino[3]=2
destino[4]=1
```

## Declaración de un array multidimensional

En algunas ocasiones necesitamos representar información relacionada en varias dimensiones, es muy frecuente, por ejemplo, comparar en forma de tabla dos conjuntos de datos relacionados. En este caso necesitamos dos arrays. Es cierto que podríamos utilizar estos dos arrays de forma independiente, pero si además estos dos arrays se encontraran unidos bajo el mismo nombre, su tratamiento se facilitaría enormemente.

Tradicionalmente cuando se piensa en una tabla (array de dos dimensiones) imaginamos un conjunto de arrays de una dimensión unidos. En Java, esto es literalmente cierto. Así, para definir un array bidimensional, definimos un array de arrays. La forma de definirlo se ve en la siguiente línea:

```
int[][] tabla=new int[2][3];
```

Declaramos un array de dos filas y tres columnas, o lo que es lo mismo un array de dos elementos, siendo cada uno de ellos a su vez un array de tres elementos.

Al definir un array multidimensional, sólo es obligatorio indicar el número de filas, después se puede reservar memoria para el resto de forma independiente.

```
int[][] tabla=new int[2][];
tabla[0]=new int[3];
tabla[1]=new int[3];
```

Cuando tenemos más dimensiones todo continúa funcionando de la misma forma, sólo es necesario añadir otro grupo más de corchetes para poder declarar cada nueva dimensión. Veamos un ejemplo de declaración de un array de tres dimensiones.

```
int[][][] tres;

tres=new int[2][][];

tres[0]=new int [2][];
tres[1]=new int [2][];

tres[0][0]=new int[2];
tres[0][1]=new int[2];
```

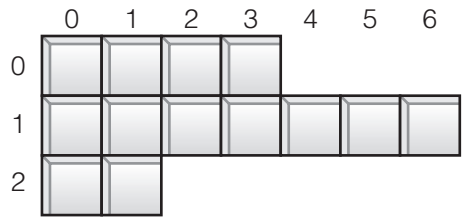



**Figura 1.2.** Array irregular.

```
tres[1][0]=new int[2];
tres[1][1]=new int[2];
```

Evidentemente, no es necesario realizar un proceso tan largo para definr un array de tres dimensiones, podemos hacerlo en una sola línea.

```
int[][][] tres=new [2][2][2];
```

No obstante, la posibilidad de declarar independientemente cada dimensión nos permite crear estructuras irregulares en las que cada fila puede tener una dimensión distinta.

```
int[][] test;
test=new int[3];
```

```
test[0]=new int[4];
test[1]=new int[7];
test[2]=new int[2];
```

## Acceso a un array multidimensional

El acceso a cada uno de los elementos del array es similar al acceso en una dimensión, sólo es necesario añadir un nuevo grupo de corchetes con el índice de la siguiente dimensión.

```
System.out.println(tabla[1][1]);
```

Al igual que con los arrays de una dimensión, en las matrices multidimensionales es posible conocer el número de elementos utilizando la propiedad `length`.

```
public class ArraysDosDim {
   public static void main(String [] args){
  int [][] raro;
  int filas=(int)((Math.random()*10)%5)+1; // Un valor entre 1 y 5
  System.out.println("Creamos un array de "+filas+" filas");
  raro=new int[filas][];

  int columnas;
  for (int i=0; i<raro.length; i++){
     columnas=(int)(Math.random()*10+1); // Un valor entre 1 y 10
     System.out.println("Creamos un array de "+columnas+
                              " columnas para la fila "+i);
     raro[i]=new int[columnas];
  }
  System.out.println("Rellenamos el array con datos aleatorios");
  for (int i=0;i<raro.length ; i++ )
     for (int j=0;j<raro[i].length ; j++)
        raro[i][j]=(int)(Math.random()*10); // Un valor entre 0 y 9
  for (int i=0;i<raro.length ; i++ ){
     for (int j=0;j<raro[i].length ; j++)
        System.out.print(raro[i][j]+"  ");
     System.out.println();
   }

  }
}
```

Un resultado de la ejecución del programa anterior se muestra a continuación.

```
Creamos un array de 5 filas
Creamos un array de 10 columnas para la fila 0
```

```
Creamos un array de 10 columnas para la fila 1
Creamos un array de 5 columnas para la fila 2
Creamos un array de 2 columnas para la fila 3
Creamos un array de 5 columnas para la fila 4
Rellenamos el array con datos aleatorios
6  6  0  8  6  9  5  1  9  2
9  9  8  6  3  8  1  8  6  4
0  8  4  9  4
6  2
3  9  3  0  4
```

**Tabla 1.6.** Operadores aritméticos.

| Operador | Descripción |
|---|---|
| + | Suma |
| - | Resta |
| * | Multiplicación |
| / | División |
| % | Módulo |
| ++ | Incremento |
| -- | Decremento |

También es posible inicializar un array multidimensional en el momento de la declaración indicando los valores para cada una de las dimensiones encerrados entre llaves. Cada pareja de llaves que se abre corresponde con una dimensión y separamos los elementos dentro de la misma dimensión con comas.

```
int[][] tabla={{1,2,3},{4,5,6}};
```

## 1.2.7. Operadores

En Java disponemos de un conjunto de operadores bastante grande, lo que permite realizar prácticamente cualquier operación de una manera directa. Disponemos de operadores para realizar desde operaciones aritméticas sencillas hasta operaciones a nivel de bit.

### Operadores aritméticos

Los operadores aritméticos deben usarse con operandos de tipo numérico, sin importar si son enteros o reales, o con operadores de tipo char.

El operador de división produce un resultado entero si los operadores son enteros y un resultado real si los operadores son reales. Posteriormente, veremos cómo usar la conversión de tipos explícita para cambiar este comportamiento.

**Tabla 1.7.** Operadores relacionales.

| Operador | Descripción |
|---|---|
| `= =` | Igual |
| `!=` | Distinto |
| `>` | Mayor que |
| `<` | Menor que |
| `>=` | Mayor o igual |
| `<=` | Menor o igual |

Los operadores de incremento y decremento pueden utilizarse como prefijo o como sufijo. Si lo usamos como prefijo, el operando se incrementa o decrementa antes de que el valor se utilice en la expresión. Si lo usamos como sufijo, la operación de decremento o incremento se realiza después de usar el valor en la expresión.

```
public class Operadores {
   public static void main(String [] args){
      int a=9;
      int b=9;
      System.out.println(a++);
      System.out.println(++b);
   }
}
```

El resultado de la ejecución del programa anterior ilustra perfectamente las diferencias existentes entre el operador de incremento (y también el de decremento) usado como prefijo o como sufijo.

```
9
10
```

## Operadores relacionales

Los operadores relacionales se usan siempre que necesitamos establecer una comparación de cualquier tipo entre dos operandos. El resultado de la evaluación de cualquier operador relacional es un tipo de datos `boolean`.

Los operadores relacionales pueden ser usados sobre cualquier tipo de datos básico. Si son usados sobre tipos más complejos, como arrays u objetos, los resultados no son correctos. Por ejemplo, el operador `==` aplicado sobre dos cadenas de carácteres, únicamente comprueba que

ambas cadenas ocupen la misma posición de memoria. Obviamente ésta no es una buena manera de comprobar si dos cadenas son iguales. Por ello, para comparar la igualdad de dos cadenas utilizaremos el método `equals()`, el cual devolverá `true` si las dos cadenas son iguales y `false` en caso contrario.

Es posible utilizar `equals()` con variables o con constantes:

```
String s="cadena";
String p="otra cadena";
boolean res = "cadena".equals(s);  // True
boolean res2 = s.equals(p);         // False
```

Si necesitamos que la comparación se realice ignorando las diferencias entre mayúsculas y minúsculas podemos utilizar `equalsIgnoreCade()`.

**Tabla 1.8.** Operadores lógicos.

| Operador | Descripción |
|---|---|
| `&` | AND |
| `\|` | OR |
| `^` | XOR |
| `&&` | AND en cortocircuito |
| `\|\|` | OR en cortocircuito |
| `!` | NOT |

```
String cad="Cadena";
String cad2="CADENA";
boolean res=cad.equals(cad2);                  // False
boolean res2=cad.equalsIgnoreCase(cad2);       // True
```

## Operadores lógicos

Los operadores lógicos se aplican sobre operandos `boolean` y como resultado se obtiene un valor también `boolean`. Este tipo de operadores proporciona la capacidad de crear expresiones relacionales más complejas al permitir unir varias expresiones simples en una expresión compleja.

Java dispone de operadores AND y OR en modo normal y en modo cortocircuito. En el modo normal, todos los operandos involucrados en la operación lógica son evaluados independientemente de que el resultado de la operación se conozca de antemano. Esto ocurre, por ejemplo, en el siguiente fragmento de código:

```
int a,b;
boolean r;
a=3;
```

**Tabla 1.9.** Operadores a nivel de bit.

| Operador | Descripción |
|---|---|
| ~ | NOT |
| & | AND |
| \| | OR |
| ^ | XOR |
| >> | Desplazamiento a la derecha |
| >>> | Desplazamiento a la derecha sin signo |
| << | Desplazamiento a la izquierda |

```
b=8;
r=a!=0 | b>a;
```

En este ejemplo, no es necesario evaluar la expresión `b>a`, ya que al ser `a!=0` cierto, la expresión entera lo es. No obstante, al usar el operador `|`, se evalúa completamente la expresión.

Si usamos la variante en cortocircuito, la expresión se evaluará hasta que se conozca el valor seguro del resultado. Esto nos permite realizar comprobaciones ligando el resultado de la parte derecha de una expresión al cumplimiento de la parte izquierda de la misma. Por ejemplo:

```
int num;
num=-9;

if (num<0 && Math.sqrt(num) > 3)
  ...
```

En este ejemplo, si utilizamos el operador `&` en lugar de `&&` obtendremos un valor NaN al tratarse el resultado de un número imaginario.

## Operadores a nivel de bit

Los operadores a nivel de bit se pueden aplicar a todos los tipos enteros, es decir, `int`, `long`, `short`, `char` y `byte`. Como se mencionó anteriormente, en Java todos los enteros tienen signo (con la excepción del tipo `char`), por lo que habrá que tenerlo en cuenta en las operaciones de desplazamiento de bits para evitar sorpresas al recoger el resultado.

En las expresiones los valores `byte` y `short` promocionan a `int`, con lo que pueden producirse resultados inesperados cuando el número que promociona es negativo. En este caso, el signo se extiende rellenando con unos la parte más significativa del número.

```
byte b=64,a;
int i;
i=b << 2;
a=(byte) (b<<2);
```

```
System.out.println("Valor de b:"+b);
System.out.println("Valor de i:"+i);
System.out.println("Valor de a:"+a);
```

El resultado de este fragmento de código es :

```
Valor de b:64
Valor de i:256
Valor de a:0
```

En el ejemplo anterior, el valor de `b` promociona a `int` al realizar el desplazamiento, por lo que al truncarlo para guardar el resultado en `a` se obtiene 0.

Los desplazamientos a la derecha pueden ser con signo o sin signo, por lo que los resultados de algunos desplazamientos también pueden parecer curiosos.

**Tabla 1.10.** Formas reducidas de los operadores.

| Operador | Descripción |
|---|---|
| ~ | NOT |
| += | Suma y asignación |
| -= | Resta y asignación |
| *= | Multiplicación y asignación |
| /= | División y asignación |
| %= | Módulo y asignación |
| &= | AND y asignación |
| \|= | OR y asignación |
| ^= | XOR y asignación |
| <<= | Desplazamiento a la izquierda y asignación |
| >>= | Desplazamiento a la derecha y asignación |
| >>>= | Desplazamiento a la derecha sin signo y asignación |

```
i=-8;
System.out.println(i>>1);
System.out.println(i>>>1);
```

La ejecución del código anterior produce como salida:

```
-4
2147483644
```

## Operadores de asignación

El operador de asignación es el signo igual (=). Asigna la expresión de la derecha a la variable de la izquierda. Además de este operador, existe en Java la posiblidad de escribir en forma reducida operaciones en las que la variable que se encuentra en la parte izquierda de una asignación, aparece también en la parte derecha relacionada con un operador aritmético o lógico. Así, expresiones como :

```
a=a+5;
```

se suelen escribir como

**Tabla 1.11.** Precedencia de los operadores.

| Operador | Asociatividad |
|---|---|
| `() [] .` | izquierda a derecha |
| `++ – ! +(unario) -(unario) () (cast) new` | derecha a izquierda |
| `* / %` | izquierda a derecha |
| `+ -` | izquierda a derecha |
| `>> >>> <<` | izquierda a derecha |
| `> >= <= > instanceof` | izquierda a derecha |
| `== !=` | izquierda a derecha |
| `&` | izquierda a derecha |
| | izquierda a derecha |
| `\|` | izquierda a derecha |
| `&&` | izquierda a derecha |
| `\|\|` | izquierda a derecha |
| `?:` | izquierda a derecha |
| `= += -= *= /= %= &= \|= = <<= >>= >>>=` | izquierda a derecha |

```
a+=5;
```

Todas las formas reducidas se encuentran en la Tabla 1.10.

## El operador ternario

El operador `?:` es el operador ternario. Puede sustituir a una sentencia `if-then-else`. Su sintaxis es:

```
exp1 ? exp2 : exp3;
```

donde, `exp1` es una expresión booleana. Si el resultado de evaluar `exp1` es true, entonces se evalúa `exp2`, en caso contrario la expresión que se evalúa es `exp3`.

Por ejemplo, podemos calcular el valor absoluto de un número con la siguiente expresión:

```
int va, x=-10;
va=(x>=0)? x : -x;
```

### Precedencia de los operadores

La Tabla 1.11. muestra la precedencia de todos los operadores de Java. Esta tabla se aplica en aquellos casos en los que sea necesario decidir qué operador se aplica antes. En caso de que dos operadores tengan la misma precedencia se evaluarán de izquierda a derecha, a no ser que tengan asociatividad derecha a izquierda.

## 1.2.8. Control de flujo

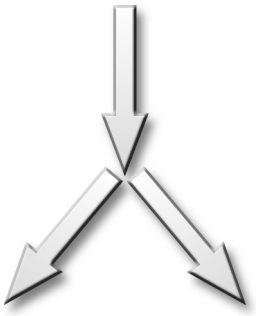
El control de la ejecución del programa se realiza en Java utilizando unas pocas sentencias. Estas sentencias son básicamente las mismas que en C/C++, es un uno de los aspectos que más similitudes se encuentran entre estos dos lenguajes.

### Sentencias condicionales

En Java disponemos de dos sentecias condicionales, la sentencia `if-else` para condiciones simples y la sentencia `switch` para realizar selecciones múltiples.

### Sentencia `if-else`

La sentecia `if-else` sirve para tomar decisiones, es posiblemente la estructura de control más usada en programación. Nos permite decidir entre dos posibles opciones excluyentes.



La sintaxis es la siguiente, donde la parte `else` es opcional:

```
if (expresión)
  sentencia-1
[else
  sentecia-2]
```

La expresión que acompaña al `if` debe producir, al ser evaluada, un valor booleano. Si este valor es `true` entonces la `sentencia-1` es ejecutada; si el valor resultante de la evaluación es false se ejecutará la `sentencia-2`. Dado que la parte `else` es opcional, si el resultado de la evaluación de la expresión es `false`, y no hay parte `else`, la ejecución continuará en la siguiente línea al `if`.

Es posible anidar sentencias `if`. Si lo hacemos tenemos que poner especial cuidado a la hora de emparejar las partes `else` con sus correspondientes `if`. Los `else` se emparejan con el `if` más cercano dentro del mismo bloque que no tenga ya asociado un `else`.

```
int resultado;
int valor=3;
if (valor>3)
  resultado=1;
else if (valor==3)
  resultado=0;
else
  resultado=-1;
```

En todas las sentencias de control es posible sustituir una sentencia por un grupo de sentencias encerradas entre llaves.

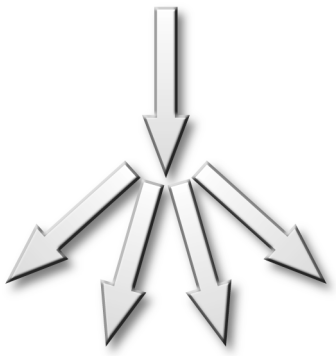
```
int resultado;
int valor=4;
if (valor>3){
   resultado=valor+10;
   resultado*=10;
```

```
}
else{
   resultado=valor-1;
   valor=0;
}
```

### Sentencia `switch`

Existen situaciones en las que una estructura `if-else` se queda corta ya que tenemos que decidirnos entre más de dos alternativas. Si se presenta este problema podemos optar por dos soluciones:

- Utilizar una secuencia de `if` anidados.
- Utilizar la sentencia `switch`.



La sintaxis de la sentencia `switch` es la siguiente:

```
switch(expresión) {
   case valor1: sentencia;
                sentencia;
                ...
                [break;]
   case valor2: sentencia;
                sentencia;
                ...
                [break;]
   ...
   [default: sentencia;
             sentencia;]
}
```

El resultado de la expresión debe ser un tipo simple de Java, los valores que acompañan a las partes `case` deben ser constantes y ser tipos compatibles con el resultado de la expresión.

El funcionamento de la sentecia `switch` es muy sencillo, se evalúa la expresión y en caso de coincidir el valor de la expresión con el valor de una de las ramas `case`, se ejecuta el conjunto de sentencias que sigue. El flujo de ejecución continúa desde ese punto, hasta el final de la estructura `switch` o hasta que se encuentra una sentencia `break`. La parte `default` es opcional, si existe, se ejecutan las sentencias que le siguen en caso de que no se encuentre coincidencia entre el resultado de la expresión y todas las partes `case` del `switch`. Si no existe `default` y no se encuentra coincidencia en ninguna rama `case` no se hace nada.

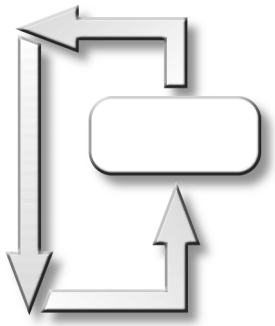
No existe ninguna restricción sobre el orden en el que deben aparecer los `case` y `default`, si bien, las opciones deben ser todas diferentes.

```
class PruebaSwitch {
    public static void main(String args[]){
        int dia=4;
        String diaSemana;
        switch (dia) {
        case 1:
            diaSemana="Lunes";
            break;
        case 2:
            diaSemana="Martes";
            break;
        case 3:
            diaSemana="Miercoles";
            break;
        case 4:
            diaSemana="Jueves";
            break;
        case 5:
            diaSemana="Viernes";
            break;
        case 6:
            diaSemana="Sábado";
            break;
        case 7:
            diaSemana="Domingo";
            break;
        default:
            diaSemana="Dia incorrecto";
        }
        System.out.println(diaSemana);
    }
}
```

## Bucles

### Bucle `while`

La sentencia iterativa más simple es un bucle `while`. En este buble una serie de sentencias se repiten mientras se cumple una determinada condición. Una característica que define a este tipo de bucle es que el cuerpo del bucle (conjunto de sentencias) se ejecuta 0 o *N* veces, ya que si la condición del bucle no se cumple no se entra a ejecutar las sentencias.



La sintaxis Java del bucle `while` es la siguiente:

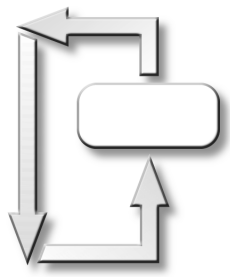
```
while(expresión)
  sentencia
```

Su funcionamiento es sencillo, se evalúa la expresión y si como resultado se obtiene un valor `true`, se ejecuta la sentencia, en caso contrario se continúa la ejecución del programa por la línea siguiente al `while`. Como siempre, es posible sustituir la sentencia, por un grupo de sentencias encerradas por llaves. A la sentencia o conjunto de sentencias que se ejecutan repetidamente dentro del bucle se las conocen como cuerpo del bucle.

```
class PruebaWhile{
     public static void main(String args[]){
          int valor=250;
          while (valor>0){
               System.out.println(valor);
               valor-=10;
          }
  }
}
```

### Bucle `for`

El bucle `for` es el tipo de bucle más potente y versátil de Java. Se comporta como un bucle `while` a la hora de comprobar la condición de control, pero permite realizar asignaciones y cambios en la variable de control dentro del mismo bucle, no obstante, un bucle `for` es equivalente a un bucle `while`.



La sintaxis del bucle `for` es un poco más complicada que la del bucle `while`:

```
for (exp1;exp2;exp3)
   sentencia
```

Puede omitirse cualquiera de las tres expresiones del bucle `for`, pero los puntos y coma deben permanecer. Las expresiones `exp1` y `exp3` son asignaciones o llamadas a función y `exp3` es una expresión condicional. En caso de que no exista `exp2` se considera que la condición es siempre cierta.

`exp1` se utiliza para inicializar la variable que controla el bucle, con `exp2` controlamos la permanencia en el mismo y con `exp3` realizamos modificaciones sobre la variable que controla el bucle para poder llegar a salir de éste.

```
class PruebaFor {
    public static void main(String args[]){
        int i;
        for (i=0;i<100;i++)
            if ((i % 2)==0)
                System.out.println(i + " es divisible entre 2");
     }
}
```

La mayor parte de las veces, la variable de control del bucle se usa solamente dentro de él, por ello, se suele declarar dentro del bucle, con lo que su ámbito y tiempo de vida coinciden con el del bucle.

```
for(int i=0;i<10; i++)
   System.out.println(i);
```

Un bucle `for` puede escribirse como un bucle `while` de la siguiente forma:

```
exp1;
while(exp2){
    proposicion
```

```
    exp3;
}

class PruebaForWhile {
    public static void main(String args[]){
        int i;
        i=0;
        while(i<100) {
            if ((i % 2)==0)
                System.out.println(i + " es divisible entre 2");
            i++;
        }
     }
  }
```

Ocurre también con cierta frecuencia que es necesario utilizar dos o más variables dentro de un bucle `for`:

```
class PruebaFor1{
    public static void main(String args[]){
        int a,c=0;
        a=9;
        for(int b=0;b<a;b++)
           c=a+b;
        System.out.println(c);
    }
}
```

Es posible realizar varias inicializaciones y varias operaciones sobre las variables de control del bucle utilizando el operador coma (`,`) para separar cada una de las operaciones.

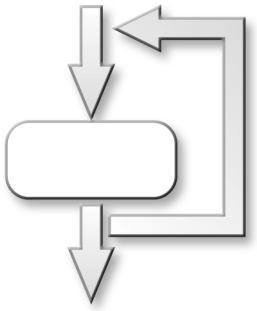
```
class PruebaFor2{

    public static void main(String args[]){
        int c=0;
        for(int b=0,a=9;b<a;b++)
            c=a+b;
        System.out.println(c);
    }
}
```

### Bucle `do-while`

Con los bucles `while` y `for` es posible que el cuerpo del bucle no llegue nunca a ejecutarse, debido a que antes de entrar en ellos se comprueba la condición y si ésta no se cumple se continúa la ejecución sin entrar en el bucle. Existen algunos casos en los que es necesario que el cuerpo del

bucle se ejecute al menos una vez, en estos casos, podemos usar un bucle `do-while` en el que la comprobación de la condición del bucle se evalúa después de ejecutado el cuerpo del bucle.



La sintaxis del bucle `do-while` es la siguiente:

```
do
  sentencia
while(expresión);
```

Este tipo de bucles es particularmente útil para controlar menús, ya que en la impresión del menú (cuerpo del bucle) debe mostrarse al menos una vez, y una vez mostrado es cuando el usuario tiene opción a elegir y su petición puede ser atendida y procesada.

### La sentencia `break`

En algunos casos es necesario salir de un bucle en un momento que no corresponde con la comprobación de la condición de salida del mismo. En estos casos podemos usar `break`. Si dentro de un programa aparece un `break`, el flujo del mismo sale del bucle (o `switch`) más interior, eso quiere decir que si tenemos bucles anidados únicamente saldremos del más interno.

```
class PruebaBreak{
     public static void main(String args[]){
          for (int i=0;i<100;i++) {
               System.out.println(i);
               if (i==50) break;
          }
          System.out.println("Fin del bucle");
    }
}
```

El uso de `break` en los programas sólo debe producirse en situaciones en las que no sea posible salir a través de la condición normal de salida o que para hacerlo sea necesario complicar la lógica del programa demasiado. Generalmente, su uso está desaconsejado.

### La sentencia `continue`

Cuando dentro de un bucle encontramos una sentencia `continue` obligamos al programa a realizar la comprobación de condición de salida en ese momento, sin necesidad de concluir la iteración en curso.

```
class PruebaContinue{
     public static void main(String args[]){
          int suma=0;
          for (int i=0;i<100;i++){
               if ((i%2)==0) continue;
               suma+=i;
          }
          System.out.println("La suma de los impares es "+suma);
    }
}
```

El uso de `continue` no es necesario en la inmensa mayoría de los programas, y lo mismo que ocurría con `break`, sólo en las situaciones de especial complejidad.

## 1.2.9. Entrada/salida básica

La entrada/salida por consola en Java es bastante limitada y no se suele utilizar mucho, ya que normalmente los programas en Java utilizan, para realizar la comunicación con el usuario, interfaces más complicadas que la consola. Esto es debido a que la mayor parte de los programas en Java se ejecutan en entornos gráficos y en ellos tienen la posibilidad de realizar la entrada y la salida de datos de forma más elaborada.

No obstante, en los primeros programas, y en aquellos en los que la interacción con el usuario no es lo primordial, el uso de la entrada/salida por consola es una buena opción.

### Flujos de datos

En Java la entrada/salida se realiza utilizando flujos (`streams`). Podemos entender un flujo como la representación de un productor/consumidor de información. Si disponemos de un flujo al que enviar información y éste es capaz de recogerla, como un sumidero, entonces diremos que es un flujo de salida, si por el contrario es el flujo el que produce la información, como una fuente, entonces tendremos un flujo de entrada.

Todos los flujos están unidos a un elemento, físico o no, que es el origen o el destino de la información que circula por el flujo. Los flujos se comportan de la misma forma, independientemente de dónde estén conectados. Dispondremos de las mismas clases y métodos si conectamos un flujo a un fichero o lo hacemos a una conexión de red.

El uso de flujos en Java hace que la E/S sea homogénea y que no sea necesario escribir código independiente para tratar la entrada por la consola y la salida a un fichero. En Java, el tratamiento es siempre el mismo.

## La entrada/salida de Java

Toda la entrada/salida se define dentro del paquete `java.io`. Dentro de este paquete se encuentran definidas todas las clases necesarias para controlar todas las eventualidades derivadas de la entrada/salida. Existen cerca de 30 clases distintas relacionadas con la entrada/salida; este volumen de clases tiene un doble análisis, por un lado disponemos de una potencia impresionante que cubre prácticamente todo, pero por otro lado es bastante complicado entender y usar la entrada/salida de Java.

Básicamente todas las clases relacionadas con la entrada/salida tienen relación con `InputStream` o con `OutputStream`. La primera clase define los parámetros básicos para la entrada y la segunda para la salida, ambas son abstractas.

## Flujos predefinidos

Todos los programas en Java importan automáticamente el paquete `java.lang`. Este paquete contiene las clases básicas necesarias para escribir programas en Java. Una de estas clases es `System`, que proporciona acceso a algunas características propias del entorno de ejecución del programa, por ejemplo, acceso a la hora del sistema, el entorno de ejecución, etc.

Dentro de `System` existen tres atributos denominados `in`, `out` y `err` que son tres flujos asociados respectivamente a la entrada estándar, la salida estándar y la salida de errores. Estos tres flujos son accesibles desde cualquier parte del programa sin necesidad de declarar ni incluir nada (son `public static`).

`System.in` es un flujo de entrada y está asociado normalmente al teclado, `System.out` y `System.err` son dos flujos de salida y están asociados normalmente con la consola.

## Salida estándar y salida de errores

Podemos enviar información a la consola mediante `System.out` y `System.err`. Utilizaremos `System.out` para enviar los mensajes referentes a la ejecución del programa, y `System.err` para enviar los mensajes producidos por errores en la ejecución. En algunos sistemas operativos no hay diferencia entre la salida estándar y la salida de errores.

Generalmente, utilizaremos los métodos `println` y `print`. La diferencia existente entre ellos radica en que el primero realiza un salto de línea al finalizar la impresión, cosa que `print` no hace.

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

```
}
```

## Entrada estándar

La entrada estándar en Java es bastante más complicada que en otros lenguajes de programación. En Java no disponemos de ningún método genérico de lectura de consola. Además es necesario que incluyamos el paquete `java.io` antes de usar cualquier método de entrada.

Para leer la entrada estándar utilizaremos `System.in`. El método de más bajo nivel de que disponemos es `read()`. Este método nos devolverá un byte del flujo de entrada en forma de entero, un `-1` si no hay más entrada disponible y no lo hará hasta que pulsemos la tecla ENTER. Esto hace de `read` un método poco usado. Además tenemos que tener en cuenta que `read()` puede elevar una excepción de tipo `IOException` como consecuencia de su uso, lo que nos obliga a tomar las medidas necesarias para que esta excepción sea convenientemente tratada o propagada.

```
import java.io.*;
public class Read {
    public static void main(String [] args ) throws IOException {
        char c;
        System.out.print("Teclea un caracter seguido de ENTER : ");
        c=(char) System.in.read();
        System.out.println("Has tecleado: "+c);
    }
 }
```

Utilizando `throws IOException` en el método `main`, avisamos de la posibilidad de que se produzca una excepción que deberá ser tratada por el programa llamante, que en nuestro caso será el sistema operativo. Es necesario realizar un `casting` ya que `read()` devuelve un valor entero.

Para poder leer cadenas de caracteres tenemos que recurrir a utilizar varias clases de `java.io`. El método que utilizaremos es `readLine()` que pertenece a la clase `BufferedReader`. Para crear el objeto de esta clase utilizamos otra clase intermedia, `InputStreamReader` que nos permite cambiar un objeto de clase `InputStream` en un objeto de tipo `Reader`, necesario para crear un objeto de tipo `BufferedReader`. Veremos posteriormente la entrada/salida con detalle, por el momento el siguiente ejemplo muestra la forma de leer una cadena.

```
import java.io.*;
public class LeerCadena {
   public static void main(String [] args ) throws IOException {
       BufferedReader stdin=new BufferedReader(
                                   new InputStreamReader(System.in));
       String cadena=new String();
```

```
        System.out.print("Escribe una cadena: ");
        cadena=stdin.readLine();
        System.out.println("Has escrito: "+cadena);
    }
}
```

Si necesitamos leer otros tipos de datos debemos hacerlo a través de `readLine()` y posteriormente convertir la cadena leída al tipo deseado.

```
import java.io.*;
public class ESBasica {
    public static void main(String [] args ) throws IOException  {
        BufferedReader stdin=new BufferedReader(
                                new InputStreamReader(System.in));

        // Lectura de un entero. int, short, byte o long
        System.out.print("Escribe un numero entero: ");
        int entero=Integer.parseInt(stdin.readLine());
        System.out.println("Has escrito: "+entero);
        // Lectura de un n·mero real. float o double
        System.out.print("Escribe un numero real: ");
        Float f=new Float(stdin.readLine());
        float real=f.floatValue();
        System.out.println("Has escrito: "+real);
        // Lectura de un valor booleano
        System.out.print("Escribe true o false: ");
        Boolean b=new Boolean(stdin.readLine());
        boolean bool=b.booleanValue();
        System.out.println("Has escrito: "+bool);
    }
}
```

## 1.2.10. Conceptos básicos de atributos y métodos

Los métodos en Java constituyen la lógica de los programas y se encuentran en las clases. La definición de un método es muy sencilla, sigue la siguiente sintaxis:

```
tipo nombre(parametros)
   cuerpo del método
```

donde `tipo` es el valor de retorno del método; este valor es devuelto por el método utilizando la palabra reservada `return`. El nombre del método es el utilizado para su invocación y debe ser un identificador único dentro de la clase.

Normalmente, las llamadas a los métodos se realizan a través de un objeto utilizando la siguiente sintaxis:

```
objeto.metodo();
```

Si la llamada se produce dentro de la misma clase, entonces el nombre del objeto no aparece, opcionalmente se puede poner `this`. El valor de retorno de un método puede ser cualquier tipo válido de Java.

Los datos dentro de una clase se llaman atributos. La definición de un atributo es la misma que la definición de cualquier variable, sólo que se realiza fuera de un método. Podemos definir atributos de cualquier tipo de datos válido. Los métodos de la clase pueden acceder a los atributos de la misma como si fueran variables definidas dentro de los mismos.

En algunas ocasiones, no resulta conveniente tener que crear un objeto para poder invocar un método. Cuando esto sucede tenemos la posibilidad de utilizar métodos estáticos, también llamados métodos de clase. Un método estático no está unido a una instancia de clase (objeto), es común a todos los objetos instanciados y puede ser invocado incluso cuando no exista ningún objeto de esa clase. Un ejemplo de método estático es `main`, cuando se ejecuta una clase que tiene `main` no existe ningún objeto para invocarlo.

Los métodos estáticos no pueden acceder a métodos no estáticos de una clase sin utilizar un objeto, ya que no es posible garantizar que exista un objeto de esa clase y los métodos no estáticos necesitan un objeto para poder ser ejecutados.

```
class Metodos {
      static void imprimirPares(){
         for (int i=1;i<=25;i++)
            if (i%2==0)
               System.out.println(i);
      }
      static void imprimirPrimos(){
         System.out.println("1\n2");
         for (int i=2;i<25;i++){
            int divisor=2;
            while (i%divisor!=0 && divisor<i-1) divisor++;
            if (divisor==i-1)
               System.out.println(i);
      }
   }
   public static void main (String [] args) {
      System.out.println("Números pares hasta 25:");
      imprimirPares();
```

```
      System.out.println("Números primos hasta 25:");
      imprimirPrimos();
   }
}
```

Tanto de atributos como de métodos se tratará con mucha más extensión posteriormente, pero es necesario analizar en este momento algunos conceptos ya que serán necesarios para los ejemplos posteriores.

## Paso de parámetros

Los parámetros indican qué información y de qué tipo se pasa al método. Es posible pasar como parámetro cualquier tipo de datos, tanto los tipos de datos básicos como los objetos más complicados, pero todos tienen que cumplir que el tipo pasado como parámetro debe coincidir con el tipo esperado.

Existen dos tipos de paso de parámetros: por valor y por referencia. En el primero, los valores que recibe el método son copias de los valores originales pasados al mismo, por tanto, cualquier modificación que se realice sobre ellos sólo será visible en el ámbito del método. En el segundo, el paso se realiza pasando una referencia al dato, esto hace que dentro del método se trabaje con el dato verdadero, no con una copia y las modificaciones no se refieran únicamente al ámbito del método.

En Java, todos los tipos básicos pasan por valor y todos los objetos pasan por referencia. Si pasamos un dato de tipo básico como parámetro, lo estamos pasando por valor y si este parámetro es modificado dentro del método, esa modificación se perderá al salir de ésta. En el siguiente ejemplo, se intenta obtener en un parámetro de la llamada el contador de números primos.

```
class Primos {
      static void imprimirPrimos(int max,int num){
        num=2;
        System.out.println("1\n2");
        for (int i=3;i<=max;i++){
              int divisor=2;
              while (i%divisor!=0 && divisor<i-1)
                     divisor++;
              if (divisor==i-1){
                     System.out.println(i);
                     num++;
              }
        }
      }
      public static void main (String [] args) {
        int maximo=25,numero=0;
        System.out.println("Números primos hasta "+maximo+":");
```

```
        // No obtendremos el valor que queremos.
        imprimirPrimos(maximo,numero);
        System.out.println("Total "+numero+" primos"); // Mal
      }
}
```

El resultado de la ejecución es claro, el valor de `numero` no ha cambiado dentro de `main`, a pesar de que el parámetro `num` sí lo ha hecho.

```
Números primos hasta 20:
1
2
3
5
7
11
13
17
19
23
Total 0 primos
```

Si necesitamos pasar por referencia un tipo básico debemos utilizar un envoltorio, recubrir el tipo básico con una capa de objeto, y de esta forma forzar su paso como parámetro por referencia.

```
class Primos2 {
  static class Envoltorio {
    int dato;
  }
  static void imprimirPrimos(int max,Envoltorio num){
    num.dato=2;
    System.out.println("1\n2");
    for (int i=3;i<=max;i++){
      int divisor=2;
      while (i%divisor!=0 && divisor<i-1)
        divisor++;
      if (divisor==i-1){
        System.out.println(i);
        num.dato++;
        }
      }
  }

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

```
    int maximo=25;
    Envoltorio numero=new Envoltorio();
    numero.dato=0;

    System.out.println("Números primos hasta "+maximo+":");
    imprimirPrimos(maximo,numero);
    System.out.println("Total "+numero.dato+" primos");
  }
}
```

En este caso, el programa funciona correctamente. Al encerrar el tipo `int` dentro de un objeto, es posible modificar su contenido dentro del método y que ese valor modificado llegue al método `main`.

```
Números primos hasta 25:
1
2
3
5
7
11
13
17
19
23
Total 10 primos
```

# 1.3. Ejercicios resueltos

## Ejercicio 1.1.

**Enunciado** Implementar un programa en Java que obtenga el número primo inferior a un valor introducido por teclado. Se recomienda implementar un método que determine si un número es primo o no, y utilizarlo para ir probando números desde el valor introducido hacia abajo.

**Solución**

```
import java.io.*;
public class PrimoMenor{
public static void main(String args[]) {
  int numero=0;
  int ultimoPrimo=1;
  int cadaNumero=numero;

  InputStreamReader isr=new InputStreamReader(System.in);
```

```
  BufferedReader br=new BufferedReader(isr);
  String valor;
  try {
    System.out.println("Introducir numero: ");
    valor=br.readLine();
      numero=Integer.valueOf(valor).intValue();
    }
  catch (IOException ioex)  {
  System.out.println("error de entrada");
  }

  while (cadaNumero>=2){
  if (esPrimo(cadaNumero)){
    ultimoPrimo=cadaNumero;
  }
  cadaNumero—;
   }
  System.out.println("el numero primo inferior o igual a "+numero +" es:
                      +ultimoPrimo);
}
static boolean esPrimo (int par){
    int resto = 1;
    int n=2;
    while((n<par/2) && (resto!=0)){
      resto= (int) (par % n);
      n++;
    }
    if (resto!=0)
      return true;
    else
      return false;
    }
}
```

## Ejercicio 1.2.

**Enunciado** Escribir un programa que calcule y presente en pantalla el "triángulo de Tartaglia". Se muestra a continuación un ejemplo para 6 filas:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

$$\begin{bmatrix} 1 & 0 & 1 \\ 2 & 1 & 3 \\ 0 & 1 & 1 \end{bmatrix}^9 = \begin{bmatrix} 2187 & 2187 & 4374 \\ 8748 & 8748 & 17496 \\ 4374 & 4374 & 8748 \end{bmatrix}$$

```
1 5 10 10  5  1
1 6 15 20 15 6 1
```

Para ello construya un array bidimensional triangular donde almacenar el triángulo de Tartaglia hasta la fila *n* y a continuación representarlo en pantalla. Obsérvese que en una fila, el valor en cada columna es el siguiente:

1 si es la primera o última columna de esa fila.

La suma de los valores en la fila anterior correspondientes a la misma columna y la situada a la izquierda.

**Solución**

```
import java.io.*;
public class TrianguloTartaglia{
   public static void main (String args[]) throws IOException
   {
     BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
     System.out.println("Introducir n: ");
     String   valor=br.readLine();
     int   n=Integer.parseInt(valor);

      int triangulo[][]=new int[n+1][];
      int k,l,m;
      for (k=0;k<n+1;k++){
         triangulo[k]=new int[k+1];
         for (l=k;l>=0;l–){
            if ((l==k)||(l==0))
               triangulo[k][l]=1;
            else
               triangulo[k][l]=triangulo[k-1][l]+triangulo[k-1][l-1];
         }

      }
      for (k=0;k<n+1;k++ ){
         for(l=0;l<k+1;l++)
            System.out.print(triangulo[k][l]+" ");
         System.out.print('\n');
      }
   }
}
```

## Ejercicio 1.3.

**Enunciado** Escribir un programa en Java que lea una matriz de teclado (por filas) y, a continuación, calcule la potencia *n*-ésima (*n* también pedido por teclado) y la presente en pantalla.

**Ejemplo**

**Solución**

```
import java.io.*;
import java.util.StringTokenizer;
public class PotenciaMatriz {
   public static void main(String args[]) throws IOException{
      int n, p;
      int i,j;
      InputStreamReader isr=new InputStreamReader(System.in);
      BufferedReader br=new BufferedReader(isr);
      String valor, cadaNum;

      System.out.println("Introducir tamaño: ");
      valor=br.readLine();
      n=Integer.valueOf(valor).intValue();

      int[][] m = new int[n][n];
      int[][] resultado = new int[n][n];

      //lectura de m
      for (i=0;i<n;i++){
         System.out.println("introduzca fila "+i);
         valor=br.readLine();
         StringTokenizer st=new StringTokenizer(valor);
         for (j=0;j<n;j++){
            if (i==j)
               resultado[i][j]=1;
            else
               resultado[i][j]=0;
            cadaNum=st.nextToken();
            m[i][j]=Integer.parseInt(cadaNum);
         }
      }
      System.out.println("matriz introducida:");
      ver(m);
      System.out.println("potencia ");
      valor=br.readLine();
      p=Integer.valueOf(valor).intValue();

      for (i=0;i<p;i++)
         resultado=producto(resultado,m);

      System.out.println("resultado:");
      ver(resultado);

   }
   public static int[][] producto (int[][] a, int [][] b){
```

```
      int i,j,k;
      int n=a.length;
      int[][] resultado=new int[n][n];
      for (i=0; i<n; i++)
         for (j=0; j<n; j++){
            resultado[i][j]=0;
            for (k=0; k<n; k++)
               resultado[i][j] = resultado[i][j] + a[i][k]*b[k][j];
         }
      return resultado;
   }
   public static void ver (int[][] a){
      int i,j;
      int n=a.length;
      for (i=0;i<n ;i++ ){
         for (j=0;j<n ;j++ ){
            System.out.print(" "+ a[i][j]);
         }
      System.out.println();
      }
   }
}
```

## Ejercicio 1.4.

**Enunciado** Para almacenar las temperaturas medias de todos los días de un año no bisiesto, declarar un array de dos dimensiones de elementos de tipo `float`, que contenga 12 filas, y cada fila tendrá 28, 30 o 31 días, según corresponda al mes. Con esta estructura, escribir el código para:

- Rellenar la estructura con temperaturas simuladas. Se modelarán las temperaturas con una variación cíclica anual a la que se superponga una variación aleatoria de 3 grados. Se sugiere una expresión como la siguiente:

$$\text{temperatura (día\_año)} = 20 - 10 \times \cos\left(\frac{\text{día\_año} \times 2\pi}{365}\right) \pm 3^{\circ}$$

- Para cada uno de los meses, la temperatura media, máxima, mínima y desviación estándar

$$\textit{media} = \frac{\sum_{i=1}^{N} a_i}{} \qquad \textit{desviación estándar} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} a_i^2 - \frac{1}{N^2}\left(\sum_{i=1}^{N} a_i\right)^2}$$

siendo $a_i$ la temperatura de cada día y N el número de días en un mes

- Para un mes seleccionado por teclado, las temperaturas de todos los días, con 7 días por línea. Representar los valores con un máximo de dos decimales.

**Solución**

```
import java.io.*;
class  Temperaturas{
   public static void main(String [] args) throws IOException {
      int k, l;
      String [] meses={"Enero", "Febr", "Marzo", "Abril", "Mayo", "Junio",
                       "Julio",  "Agosto", "Sept", "Octub", "Nov", "Dic"};
      float temperaturas[][]  = new float[12][];
      temperaturas[0]=new float[31];
      temperaturas[1]=new float[28];
      temperaturas[2]=new float[31];
      temperaturas[3]=new float[30];
      temperaturas[4]=new float[31];
      temperaturas[5]=new float[30];
      temperaturas[6]=new float[31];
      temperaturas[7]=new float[31];
      temperaturas[8]=new float[30];
      temperaturas[9]=new float[31];
      temperaturas[10]=new float[30];
      temperaturas[11]=new float[31];

      int dia=0;
      for (k=0;k<12;k++ ){
         for (l=0;l<temperaturas[k].length;l++ ){
            temperaturas[k][l]=(float)
               (20.0-10*Math.cos(dia*2*Math.PI/365)+4*(Math.random()-0.5));
            dia++;
         }
      }

      float suma, sumaCuad, maximo, minimo, media, desv;
      int mes;

      for (mes=0;mes<12;mes++)     {
         suma=0.0f;
         sumaCuad=0.0f;
         maximo=0.0f;
         minimo=0.0f;

         for (l=0;l<temperaturas[mes].length;l++){
            if (l==0)
               maximo=minimo=temperaturas[mes][l];
            else{
```

```
            if(temperaturas[mes][l]>maximo)
               maximo=temperaturas[mes][l];
            if(temperaturas[mes][l]<minimo)
               minimo=temperaturas[mes][l];
         }
         suma+=temperaturas[mes][l];
         sumaCuad+=(temperaturas[mes][l])*(temperaturas[mes][l]);
      }
      int N=temperaturas[mes].length;
      media=suma/N;
      desv=(float)Math.sqrt(sumaCuad/N-suma*suma/(N*N));

      System.out.print(meses[mes]+":\t media="+formato(media));
      System.out.print("\tdesv: "+formato(desv));
      System.out.print(",\tmaxima: "+formato(maximo));
      System.out.println(",\tminima: "+formato(minimo));
   }// for de meses
   System.out.println("Mes para visualizar temperaturas ");
   InputStreamReader isr=new InputStreamReader(System.in);
   BufferedReader br=new BufferedReader(isr);
   String mes_cadena;

   int mesSelec=Integer.parseInt(br.readLine());

   System.out.println("temperaturas mes "+meses[mesSelec-1]);
   for (l=0;l<temperaturas[mesSelec-1].length;l++){
      System.out.print(formato(temperaturas[mesSelec-1][l])+"\t");
      if((l+1)%7==0)
         System.out.print('\n');
   }
  }
  static double formato(double v){
   return ((int)(100*v))/100.0;
  }
}
```

## Ejercicio 1.5.

**Enunciado** Escribir un procedimiento que intercambie la fila *i*-ésima con la *j*-ésima de una matriz m*n.

**Solución**

```
class Matrices4 {
   public static void main(String [] args){
      int [][]matriz = {{3,5,-1,6},{2,6,8,2},{13,4,1,1},
                        {12,-2,0,2},{1,1,1,1}};
      int i = 2,j = 4,k;
```

```
      int aux;
      for (k=0;k<4;k++){
         aux = matriz[i][k];
         matriz [i][k] = matriz [j][k];
         matriz [j][k] = aux;
      }
      // visualizar la matriz
      for (int m=0;m<5;m++){
         for (int n=0;n<4;n++)
            System.out.print(matriz[m][n] + ", ");
            System.out.println();
      }

   }
}
```

## Ejercicio 1.6.

**Enunciado** Escribir un programa que lea números por teclado mientras sean positivos y calcule su media y varianza.

**Solución**

```
class MediaVarianza {
   public static void main(String [] args){
      System.out.println("numeros");
      double sumax=0.0, sumax2=0.0, media=0.0, desv=0.0, cada=0.0;
      int      numero=0;
      do{
         valor=br.readLine();
         cada=Double.parseDouble(valor);
         if (cada>=0){
            sumax+=cada;
            sumax2+=cada*cada;
            numero++;
         }
      } while(cada>=0);
      if (numero>0){
         media=sumax/numero;
         desv=Math.sqrt(sumax2/numero-media*media);
      }
      System.out.println("numero: "+numero+" media: "+media+" desv: "+desv);
}
```

## Ejercicio 1.7.

**Enunciado** Desarrollar un programa que implemente una calculadora, presentando los siguientes menús al usuario, leyendo del teclado los operandos y opciones seleccionadas, y sacando el resultado por pantalla:

Menú 1:

Indicar numero de operadores:

0 (salir)

1 operador (ir a Menú 1.1)

2 operadores (ir a Menú 1.2)

Menú 1.1

Introducir operando

Operación deseada:

1. logaritmo
2. exponente
3. seno
4. coseno

Menú 1.2

Introducir operando1

**Introducir operando2**

Operación deseada:

1. suma
2. resta
3. producto
4. división

**Solución**

```
import java.io.*;
public class Calculadora {
 public static void main(String args[]) throws IOException{
   int numero=1;
   char menu1, menu2;
   double op1, op2;
   InputStreamReader isr=new InputStreamReader(System.in);
   BufferedReader br=new BufferedReader(isr);
   String valor;

   do{
      System.out.println("Indicar numero de operadores: ");
      System.out.println("0 (salir)");
      System.out.println("1 operador");
```

```
System.out.println("2 operadores");
valor=br.readLine();
menu1=valor.charAt(0);
System.out.println("menu1="+menu1);
switch (menu1){
case '1':
   System.out.println("introducir operando");
   valor=br.readLine();
   op1=Double.parseDouble(valor);

   System.out.println("Operacion deseada:");
   System.out.println("1.- logaritmo");
   System.out.println("2.- exponente");
   System.out.println("3.- seno");
   System.out.println("4.- coseno");
   valor=br.readLine();
   menu2=valor.charAt(0);//Integer.valueOf(valor).intValue();
   switch (menu2){
      case '1':
         System.out.println("log("+op1+")="+Math.log(op1));
       break;
      case '2':
         System.out.println("exp("+op1+")="+Math.exp(op1));
       break;
      case '3':
         System.out.println("sen("+op1+")="+Math.sin(op1));
       break;
      case '4':
         System.out.println("raiz("+op1+")="+Math.sqrt(op1));
      }
   break;
case '2':
   System.out.println("introducir operando1");
   valor=br.readLine();
   op1=Double.parseDouble(valor);
   System.out.println("introducir operando2");
   valor=br.readLine();
   op2=Double.parseDouble(valor);
   System.out.println("Operacion deseada:");
   System.out.println("1.- suma");
   System.out.println("2.- resta");
   System.out.println("3.- producto");
   System.out.println("4.- division");
   valor=br.readLine();
   menu2=valor.charAt(0);//Integer.valueOf(valor).intValue();
```

```
            switch (menu2){
               case '1':
                  System.out.println(op1+"+"+op2+"="+(op1+op2));
                break;
               case '2':
                  System.out.println(op1+"-"+op2+"="+(op1-op2));
                break;
               case '3':
                  System.out.println(op1+"*"+op2+"="+(op1*op2));
                break;
               case '4':
                  System.out.println(op1+"/"+op2+"="+(op1/op2));
               }
            }//switch principal
         }while (menu1 > '0' && menu1 <= '2');

 }
}
```

## Ejercicio 1.8.

**Enunciado** Implementar un programa en Java que permita leer información por la entrada estándar (teclado) hasta que el usuario teclee la cadena "fin". Para mostrar la información por la salida estándar utilizar el flujo predefinido `System.out`.

**Solución**

```
import java.io.*;
public class LeerTeclado {
     public static void main(String [] args) throws IOException{
          //lectura desde teclado
          InputStreamReader flujoEntrada =new InputStreamReader(System.in);
          BufferedReader teclado = new BufferedReader(flujoEntrada);
          System.out.println("Introduce líneas.....");
          System.out.println("Teclea FIN para terminar.");
          boolean terminar=false;
          String linea=null;
          while (!terminar){
               linea = teclado.readLine();
               terminar = linea.equals("fin");
               if (terminar){
                    System.out.println("no más líneas....."+linea);
               }else{
                    System.out.println("has tecleado:: "+linea);
```

```
            }
        }
    }//fin main
} //fin LeerTeclado
```

## Ejercicio 1.9.

**Enunciado** Implementar un programa en Java que permita mostrar información por la salida estándar (monitor). Debe utilizarse tanto el flujo predefinido de Java (`System.out`) como alguna de las clases disponibles en el API de Java 1.2 y que permitirían realizar la misma operación (se recomienda estudiar la clase `java.io.PrintWriter`).

**Solución** La salida estándar está definida en Java como un flujo (`System.out`), que habitualmente es el monitor (realmente se trata de la consola o aplicación que permita mostrar texto a través del monitor, por ejemplo, en el caso del sistema operativo Linux se trataría de un terminal). El siguiente ejemplo muestra dos formas muy simples de escribir por pantalla, la primera (y más habitual) utilizando el método `println()`; (línea 6) del flujo predefinido `System.out` (línea 5). La segunda establece una conexión entre un `PrintWriter` utilizando como flujo de salida el estándar, de los constructores disponibles es importante utilizar: `PrintWriter (OutputStream out, boolean autoFlush)`, y poner el segundo parámetro a `true` para forzar el vaciado automático del `Stream` (en caso contrario no se verá nada por pantalla).

```
import java.io.*;
public class EscribirTeclado {
   public static void main(String [] args) throws IOException{
      //escritura por pantalla
      PrintWriter teclado = new PrintWriter(System.out,true);
      System.out.println("Forma habitual de escribir por la salida estandar");
      String [] lineas = {"en un lugar","de la Mancha", "de cuyo nombre",
                          "no quiero acordarme"};
      for(int i=0; i<lineas.length;i++)
         teclado.println(lineas[i]);
   }//fin main
}//fin escribir teclado
```

## Ejercicio 1.10.

**Enunciado** Escribir un programa que tome un número entero por teclado y muestre por pantalla el número de bits a 1 y a 0 que tiene su representación binaria.

**Solución**

```
import java.io.*;

class ContarBits {
    public static void main(String [] args) throws IOException{
```

```
    InputStreamReader flujoEntrada =new InputStreamReader(System.in);
    BufferedReader teclado = new BufferedReader(flujoEntrada);

    int numero;
    System.out.print("Dame un numero: ");
    numero=Integer.parseInt(teclado.readLine());
    int mascara=1,numeroUnos=0;

    // Cogemos el número de bits por exceso
    int numBits=(int)(Math.log(numero)/Math.log(2))+1;

    // Comprobamos con un AND si el bit está puesto a 1
    for (int i=0;i<=numBits;i++){
      if ((numero & mascara) != 0 )
        numeroUnos++;
        mascara<<=1;
      }
      System.out.print("El numero "+numero+" tiene "+numeroUnos+" bits a 1 y ");
      System.out.println(numBits-numeroUnos+" a 0 ");
  }
}
```