UD01 – ELEMENTOS DE UN PROGRAMA INFORMÁTICO

Contenido

1 Lenguajes de programación. Lenguajes de bajo y alto nivel

1.1 ¿Qué es un programa?

Un primer paso para poder empezar a estudiar cómo hay que hacer un programa informático es tener claro qué es un programa. En contraste con otros términos usados en informática, es posible referirse a un "programa" en el lenguaje coloquial sin tener que estar hablando necesariamente de ordenadores. Os podría estar refiriéndose al programa de un ciclo de conferencias o de cine. Pero, a pesar de no tratarse de un contexto informático, este uso ya os aporta una idea general de su significado: un conjunto de eventos ordenados de forma que suceden de forma secuencial en el tiempo, uno tras otro.

Otro uso habitual, ahora ya sí vinculado al contexto de las máquinas y los autómatas, podría ser para referirse al programa de una lavadora o de un robot de cocina. En este caso, sin embargo, lo que sucede es un conjunto no tanto de eventos, sino de órdenes que el electrodoméstico

sigue ordenadamente. Una vez seleccionado el programa que queremos, el electrodoméstico hace todas las tareas correspondientes de manera autónoma.

Entrando ya, ahora sí, en el mundo de los ordenadores, la forma en que se estructura el tipo de tareas que estos pueden hacer tiene mucho en común con los programas de electrodomésticos. En este caso, sin embargo, en lugar de transformar ingredientes (o lavar ropa sucia, si se tratase de una lavadora), lo que la computadora transforma es información o datos. ***Un programa informático no es más que una serie de órdenes que se llevan a cabo secuencialmente, aplicadas sobre un conjunto de datos.***

¿Qué datos procesa un programa informático? Bueno, esto dependerá del tipo de programa:

- Un editor procesa los datos de un documento de texto.
- Una hoja de cálculo procesa datos numéricos.
- Un videojuego procesa los datos que dicen la forma y ubicación de enemigos y jugadores, etc.
- Un navegador web procesa las órdenes del usuario y los datos que recibe desde un servidor en Internet.

Por lo tanto, la tarea de un programador informático es escoger qué órdenes constituirán un programa de ordenador, en qué orden se deben llevar a cabo y sobre qué datos hay que aplicarlas, para que el programa lleve a cabo la tarea que ha de resolver. La dificultad de todo ello será más grande o pequeña dependiendo de la complejidad misma de lo que es necesario que el programa haga. No es lo mismo establecer qué debe hacer el ordenador para resolver una multiplicación de tres números que para procesar textos o visualizar páginas en Internet.

Por "ejecutar un programa" se entiende hacer que el ordenador siga todas sus órdenes, desde la primera hasta la última.

Por otra parte, una vez hecho el programa, cada vez que lo ejecute, el ordenador cumplirá todas las órdenes del programa.

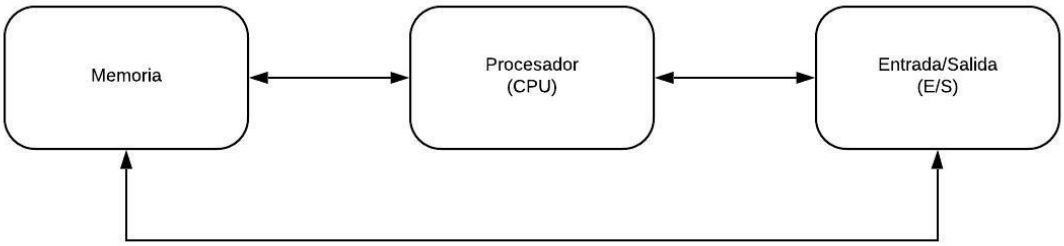
De hecho, un ordenador es incapaz de hacer absolutamente nada por sí mismo, siempre hay que decirle qué debe hacer. Y eso se le llama mediante la ejecución de programas. Aunque desde el punto de vista del usuario puede parecer que cuando se pone en marcha un ordenador este funciona sin ejecutar ningún programa concreto, hay que tener en cuenta que su sistema operativo es un programa que está siempre en ejecución.

1.2 Tipo de órdenes que acepta un ordenador

Para llevar a cabo la tarea encomendada, un ordenador puede aceptar diferentes tipos de órdenes. Estas se encuentran limitadas a las capacidades de los componentes que lo conforman. Por lo tanto, es importante tener presente este hecho para saber qué se puede pedir en el ordenador al crear un programa.

La estructura interna del ordenador se divide en una serie de componentes, todos comunicados entre sí, tal como muestra la figura 1 de manera muy simplista, pero suficiente para empezar. Cada orden de un programa está vinculada de una manera u otra a alguno de estos componentes.

Figura 1 Componentes básicos de un ordenador



El **procesador** (o CPU, *Central Processing Unit*) es el centro neurálgico del ordenador y el elemento que es capaz de llevar a cabo las órdenes de manipulación y transformación de los datos. Un conjunto de datos se puede transformar de muchas maneras, según las capacidades que ofrezca cada procesador. Sin embargo, hay muchas transformaciones que todos pueden hacer. Un ejemplo es la realización de operaciones aritméticas (suma, resta, multiplicación, división), tal como hacen las calculadoras.

La **memoria** permite almacenar datos mientras éstas no están siendo directamente manipuladas por el procesador. Cualquier dato que debe ser tratado por un programa estará en la memoria. Mediante el programa se puede ordenar al procesador que guarde ciertos datos o que los recupere en cualquier momento. Normalmente, cuando se habla de memoria a este nivel nos referimos a memoria dinámica o RAM (*Random Access Memory*, memoria de acceso aleatorio). Esta memoria no es persistente y una vez termina la ejecución del programa todos los datos con las que trataba desvanecen. Por lo tanto, la información no se guardará entre sucesivas ejecuciones diferentes de un mismo programa.

En ciertos contextos es posible que nos encontramos también con memoria ROM (*Read-Only Memory*, memoria de sólo lectura). En ésta, los datos están predefinidas de fábrica y no se puede almacenar nada, sólo podemos leer lo que contiene. Hay que decir que no es el caso más habitual.

El sistema de **entrada / salida** (abreviado como E / S) permite el intercambio de datos con el exterior del ordenador, más allá del procesador y la memoria. Esto permite traducir la información procesada en acciones de control sobre cualquier periférico conectado al ordenador. Un ejemplo típico es establecer una vía de diálogo con el usuario, ya sea por medio del teclado o del ratón para pedirle información, como por la pantalla, para mostrar los resultados del programa. Este sistema es clave para convertir un ordenador en una herramienta de propósito general, ya que lo capacita para controlar todo tipo de aparatos diseñados para conectarse.

Otra posibilidad importante de la computadora, dadas las limitaciones del sistema de memoria, es poder interactuar con el hardware de almacenamiento persistente de datos, como un disco duro.

Partiendo de esta descripción de las tareas que puede llevar a cabo un ordenador según los elementos que lo componen, un ejemplo de programa para multiplicar dos números es el mostrado en la tabla 1. expresado en lenguaje natural (el lenguaje natural es aquel que

empleamos los humanos para comunicarnos habitualmente). Insistir en que los datos deben estar siempre almacenados en la memoria para poder operar.

Tabla1 Un programa que multiplica dos números usando lenguaje natural

Orden	**Elemento que la efectúa**
1. Lee un número del teclado.	E / S (teclado)
2. Guardar el número en la memoria.	memoria
3. Lee otro número del teclado.	E / S (teclado)
4. Guardar el número en la memoria.	memoria
5. Recupera los números de la memoria y haz la multiplicación.	procesador
6. Guardar el resultado en la memoria.	memoria
7. Muestra el resultado en la pantalla.	E / S (pantalla)

1.3 Crear un programa ejecutable

Para crear un programa hay que establecer qué órdenes deben darse en el ordenador y en qué secuencia. Ahora bien, hoy en día los ordenadores todavía no entienden el lenguaje natural (como se utiliza en la tabla 1), ya que está lleno de ambigüedades y aspectos semánticos que pueden depender del contexto.

Para especificar las órdenes que debe seguir un ordenador el que se usa es un **lenguaje de programación**. Se trata de un lenguaje artificial diseñado expresamente para crear algoritmos que puedan ser llevados a cabo por el ordenador. (Por *artificial* entendemos lo que no ha evolucionado a partir del uso entre humanos, sino que ha sido creado expresamente, en este caso para ser usado con los ordenadores).

Existen muchos lenguajes de programación, cada uno con sus características propias, de manera que unos son más o menos indicados para resolver un tipo de tareas u otras.

Los lenguajes de programación están formados por un conjunto de símbolos (llamado alfabeto), reglas gramaticales (léxico/morfológicas y sintácticas) y reglas semánticas, que en conjunto definen las estructuras válidas en el lenguaje y su significado. De esta forma la computadora puede interpretar correctamente cada orden que se le da.

En un lenguaje de programación determinado, el conjunto de órdenes concretas que se pide al ordenador que haga se denomina conjunto de **instrucciones**.

Normalmente, el conjunto de instrucciones de un programa se almacena dentro de un conjunto de archivos. Estos archivos los edita el programador para crear o modificar el programa.

Los lenguajes de programación se pueden clasificar en diferentes categorías según sus características. De hecho, algunas de las propiedades del lenguaje de programación tienen consecuencias importantes sobre el proceso a seguir para poder crear un programa y para ejecutarlo.

Hay dos maneras de clasificar los lenguajes de programación:

- Según si se trata de un lenguaje compilado o interpretado. Esta propiedad afecta los pasos a seguir para llegar a obtener un archivo ejecutable. O sea, un fichero con el mismo formato que el de las aplicaciones que puede tener instaladas en su equipo.
- Según si se trata de un lenguaje de nivel alto o bajo. Esta propiedad indica el grado de abstracción del programa y si sus instrucciones están más o menos estrechamente vinculadas al funcionamiento del hardware de un ordenador.

1.3.1 El lenguaje máquina

El lenguaje de máquina o código de máquina es el lenguaje que elegiríamos si quisiéramos hacer un programa que trabajara directamente sobre el procesador.

En este lenguaje, cada una de las instrucciones se representa con una secuencia binaria, en ceros (0) y unos (1), y todo el conjunto de instrucciones del programa queda almacenado de manera consecutiva dentro de un fichero de datos en binario.

Cuando se pide la ejecución de un programa en lenguaje de máquina, este se carga en la memoria del ordenador. A continuación, el procesador va leyendo una por una cada una de las instrucciones, las decodifica y las convierte en señales eléctricas de control sobre los elementos del ordenador para que hagan la tarea solicitada. A muy bajo nivel, casi se puede llegar a establecer una correspondencia entre los 0 y 1 de cada instrucción y el estado resultante de los transistores* dentro de los chips internos del procesador.

El transistor es el componente básico de un sistema digital. Se puede considerar como un interruptor, donde 1 indica que pasa corriente y 0 que no pasa.

El conjunto de instrucciones que es capaz de decodificar correctamente un procesador y convertir en señales de control es específico para cada modelo y está definido por su fabricante. El diseñador de cada procesador se inventó la sintaxis y las instrucciones del código máquina de acuerdo con sus necesidades cuando diseñó el hardware. Por tanto, las instrucciones en formato binario que puede decodificar un tipo de procesador pueden ser totalmente incompatibles con las que puede decodificar otro. Esto es lógico, ya que sus circuitos son diferentes y, por tanto, las señales eléctricas de control que debe generar son diferentes para cada caso. Dos secuencias de 0 y 1 iguales pueden tener efectos totalmente diferentes en dos procesadores de modelos diferentes,

Un programa escrito en lenguaje de máquina es específico para un tipo de procesador concreto. No se puede ejecutar sobre ningún otro procesador, a menos que sean compatibles. Un procesador concreto sólo entiende directamente el lenguaje de máquina especificado por su fabricante.

Aunque, como se puede ver, en realidad hay muchos lenguajes de máquina diferentes, se usa esta terminología para englobar a todos. Si se quiere concretar más se puede decir "lenguaje de máquina del procesador X".

Ahora bien, estrictamente hablando, si optaran por hacer un programa en lenguaje de máquina, nunca lo haría generando directamente archivos con todo de secuencias binarias. Sólo tiene que imaginar el aspecto de un programa de este tipo en formato impreso, consistente en una enorme ristra de 0 y 1. Sería totalmente incomprensible y prácticamente imposible de analizar. En realidad, lo que se usa es un sistema auxiliar de mnemotécnicos en el que se asigna a cada instrucción en binario un identificador en formato de texto legible, más fácilmente

comprensible para los humanos. De este modo, es posible generar un programa a partir de ficheros en formato texto.

Esta recopilación de mnemotécnicos es lo que se conoce como el **lenguaje ensamblador**.

A título ilustrativo, la tabla 2 muestra las diferencias de aspecto entre un lenguaje de máquina y ensamblador equivalentes para un procesador de modelo 6502. Sin entrar en más detalles, es importante mencionar que tanto en lenguaje de máquina como en ensamblador cada una de las instrucciones se corresponde a una tarea muy simple sobre uno de sus componentes. Hacer que el ordenador realice tareas complejas implica tener que generar muchas instrucciones en estos lenguajes.

Tabla 2 Equivalencia entre un lenguaje ensamblador y su lenguaje de máquina asociado

instrucción ensamblador	**Lenguaje de máquina equivalente**
LDA # 6	1010100100000110
CMP & 3500	110011010000000000110101
LDA & 70	1010010101110000
INX	11101111

1.3.2 Lenguajes compilados

Para crear un programa lo que haremos es crear un archivo y escribir el conjunto de instrucciones que queremos que el ordenador ejecute. Para empezar, bastará con un editor de texto simple (como el Bloc de Notas (Windows) o Gedit (Linux)).

Una vez se ha terminado de escribir el programa, el conjunto de archivos de texto resultante donde se encuentran las instrucciones se dice que contiene el **código fuente**.

Este sistema de programar más cómodo para los humanos hace surgir un problema, y es que los archivos de código fuente no contienen lenguaje de máquina y, por tanto, resultan incomprensibles para el procesador. No se le puede pedir que la ejecute directamente; esto sólo es posible usando lenguaje de máquina. Para poder generar código máquina hay que hacer un proceso de traducción desde los mnemotécnicos que contiene cada archivo a las secuencias binarias que entiende el procesador.

El proceso llamado ***compilación*** es la traducción del código fuente de los archivos del programa en archivos en formato binario que contienen las instrucciones en un formato que el procesador puede entender. El contenido de estos archivos se denomina **código objeto**. El programa que hace este proceso se denomina **compilador**.

El código objeto de las instrucciones en la tabla 2 tiene este aspecto:

```
1010100100000011011 0011010000000000011 0101101001010111100 0011101111
```

Para el caso del ensamblador el proceso de compilación es muy sencillo, ya que es una mera traducción inmediata de cada mnemotécnico a la secuencia binaria que le corresponde. En principio, con esto ya habría suficiente para poder hacer cualquier programa, pero ceñirse sólo al uso de lenguaje ensamblador conlleva ciertas ventajas e inconvenientes que hacen que en realidad no sea usado normalmente, sólo en casos muy concretos.

Cabe destacar que con ensamblador el programador tiene control absoluto del hardware del ordenador a nivel muy bajo. Prácticamente se puede decir que controla cada señal eléctrica y los valores de los transistores dentro de los chips. Esto permite llegar a hacer programas muy eficientes en el que el ordenador hace exactamente lo que se le indica sin ningún tipo de ambigüedad. En contraposición, los programas en ensamblador sólo funcionan para un tipo de procesador concreto, no son portables. Si hay que hacer para otra arquitectura, normalmente hay que empezar de cero. Además -y es el motivo de más peso para pensárselo dos veces si se quiere usar este lenguaje- crear un programa complejo requiere un grado enorme de experiencia sobre cómo funciona el hardware del procesador, y la longitud sería considerable. Esto hace que sean programas complicados de entender y que haya que dedicar mucho tiempo a hacerlos.

Se considera que el código de máquina y el ensamblador son los lenguajes de nivel más bajo que existen, ya que sus instrucciones dependen directamente del tipo de procesador.

Actualmente, para generar la inmensa mayoría de programas se utilizan los llamados lenguajes de alto nivel. Estos ofrecen un conjunto de instrucciones que son fáciles de entender para el ser humano y, por tanto, poseen un grado de abstracción más alto que el lenguaje ensamblador (ya que no están vinculados a un modelo de procesador concreto). Cada una de las instrucciones se corresponde a una orden genérica en la que lo más importante es su aspecto funcional (que se quiere hacer), sin importar cómo se materializa esto en el hardware del ordenador ni mucho menos en señales eléctricas. En cualquier caso, hay que advertir que esta clasificación no siempre es absoluta. Se puede decir que hay lenguajes de "nivel más alto o bajo que otros".

El proceso para generar un programa a partir de un lenguaje de nivel alto es muy parecido al que hay que seguir para hacerlo usando el lenguaje ensamblador, ya que las instrucciones también se escriben en formato texto dentro de archivos de código fuente. La ventaja adicional es que el formato y la sintaxis ya no están ligados al procesador, y, por tanto, pueden tener el formato que quiera el inventor del lenguaje sin que deba tener en cuenta el hardware de los ordenadores donde se ejecutará. Normalmente, las instrucciones y la sintaxis han sido elegidas para facilitar la tarea de creación y comprensión del código fuente del programa.

De hecho, en los lenguajes de nivel alto más frecuentes, entre los que está los que se aprenderán a usar en este módulo, las instrucciones dentro de un programa se escriben como una secuencia de sentencias.

Una **sentencia** es el elemento mínimo de un lenguaje de programación, a menudo identificado por una cadena de texto especial, que sirve para describir exactamente una acción que el programa tiene que hacer.

Por tanto, a partir de ahora se usará el término *sentencia* en lugar de *instrucción* cuando el texto se refiera a un lenguaje de este tipo.

2 Ensambladores, compiladores e intérpretes

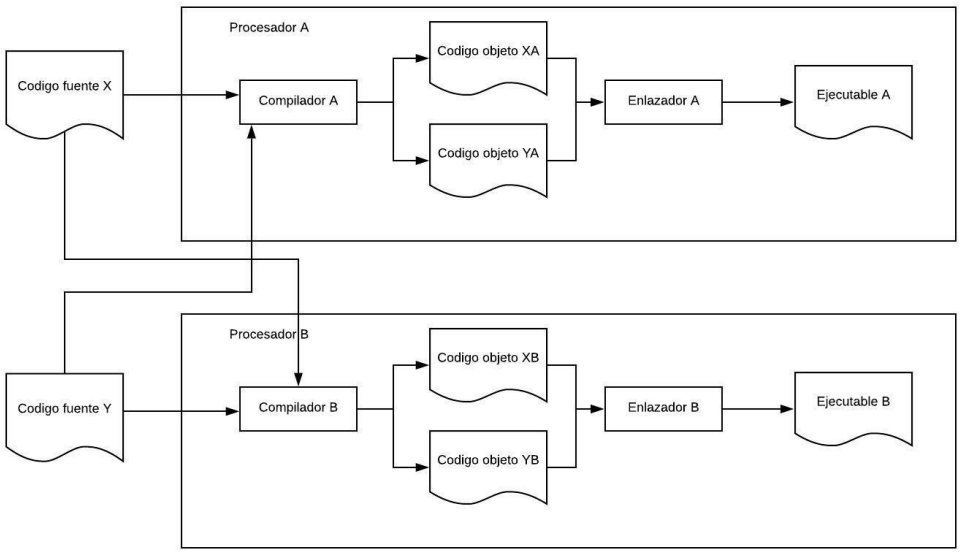
2.1 Lenguajes compilados

Una vez se han terminado de generar los archivos con su código fuente, estos se han de compilar para traducirlos a código objeto.

El proceso de traducción a código objeto será bastante más complicado en los lenguajes de alto nivel que desde ensamblador. El compilador de un lenguaje de nivel alto es un programa mucho más complejo. En cuanto al proceso de compilación, una consecuencia adicional de que el lenguaje no dependa directamente del tipo de procesador es que desde un mismo código fuente se puede generar código objeto para diferentes procesadores. Por lo tanto, sólo es necesario disponer de un compilador diferente para cada tipo de procesador que se quiera soportar.

Ya que para cada archivo de código fuente se genera un fichero de código objeto, tras el proceso de compilación hay un paso adicional llamado *enlazamiento* (*link*), en el que estos dos códigos se combinan para generar un único archivo ejecutable. Coloquialmente, cuando le pedimos que compila un programa ya se suele dar por hecho que también se enlazará, en su caso. Aun así, formalmente se consideran dos pasos diferenciados.

Figura 2 Proceso de compilación (y enlazado) del código fuente



Errores de compilación

El compilador es fundamental para generar un programa en un lenguaje compilado, ya sea de nivel alto o bajo. Para poder hacer su trabajo de manera satisfactoria y generar código objeto a partir del código fuente necesario que las instrucciones sigan perfectamente la sintaxis del lenguaje elegido. Por ejemplo, hay que usar sólo las instrucciones especificadas en el lenguaje y hacerlo en el formato adecuado. Si no es así, el compilador es incapaz de entender el orden que se quiere dar al ordenador y no sabe cómo traducirla a lenguaje máquina.

Cuando el compilador detecta que una parte del código fuente no sigue las normas del lenguaje, el proceso de compilación se interrumpe y anuncia que hay un **error de compilación**.

Cuando esto ocurre, habrá que repasar el código fuente e intentar averiguar dónde está el error. Normalmente, el compilador da algún mensaje sobre el que considera que está mal.

Hay que ser conscientes de que un programador puede llegar a dedicar una buena parte del tiempo de la generación del programa en la resolución de errores de compilación, aunque hoy en día los entornos de desarrollo facilitan enormemente esta tarea.

Ahora bien, que un programa compile correctamente sólo significa que se ha escrito de acuerdo con las normas del lenguaje de programación, pero no aporta ninguna garantía de que sea correcto, es decir, que haga correctamente la tarea para la que se ha ideado.

2.2 Lenguajes interpretados

En contraposición a los lenguajes compilados, tenemos los lenguajes interpretados. Como en el caso de los lenguajes compilados, los programas también se escriben en archivos de texto que contienen código fuente. La diferencia surge inmediatamente después de terminar de escribirlos, en la forma en que se genera un archivo ejecutable. La cuestión es que, precisamente, ni se genera ningún código objeto ni ningún archivo ejecutable. Se trabaja directamente con el archivo de código fuente. Cuando el trabajo escrito, el proceso de creación del programa ejecutable ha finalizado.

Un intérprete no traduce el código fuente del programa en código objeto y entonces lo ejecuta. Lo que hace es ejecutar diferentes instrucciones de su propio código según va leyendo las instrucciones del código fuente.

El intérprete va leyendo las instrucciones del código fuente una por una y las procesando de manera que actúa de una manera o de otra, es decir, ejecuta una parte u otra de su propio código objeto según el tipo de instrucción leída. A fin de cuentas, sería un programa que imita el comportamiento de un procesador, pero a escala de software.

Un programa en un lenguaje interpretado se ejecuta indirectamente, mediante la ayuda de un programa auxiliar llamado **intérprete**, que procesa el código fuente y gestiona la ejecución.

De igual manera que en los lenguajes compilados, puede suceder que el programador haya incluido sin darse cuenta algún error de sintaxis en las instrucciones. En este caso, será el intérprete quien mostrará el error y se negará a ejecutar el programa hasta que haya sido solucionado.

En general la ejecución de programas escritos en lenguajes interpretados es más lenta en que la de los lenguajes compilados.

Por sus características, los lenguajes interpretados no requieren un proceso posterior de enlazado.

Entre los lenguajes interpretados más conocidos encontramos JavaScript, PHP o Perl. Muchos son lenguajes de *script*, que permiten el control de aplicaciones en un sistema operativo, llevar a cabo procesos por lotes (*batch*) o generar dinámicamente contenido web.

2.3 Lenguajes híbridos

Algunos lenguajes interpretados usan una aproximación híbrida. El código fuente se compila y como resultado se genera un fichero de datos binarios escrito en código. En Java este código intermedio se denomina *bytecode*, sin embargo, no es formalmente código objeto, ya que no es capaz de entenderlo el hardware de ningún procesador. Sólo un intérprete lo puede procesar y ejecutar. Simplemente es una manera de almacenar más eficiente y en menos espacio, en formato binario y no en texto, las instrucciones incluidas en el código fuente. Este es el motivo por el que, a pesar de necesitar un proceso de compilación, estos lenguajes no se consideran realmente compilados.

Entre los lenguajes híbridos encontramos a Java o C#.

3 Entornos de desarrollo

Una vez se ha descrito el proceso general para desarrollar y llegar a ejecutar un programa, se hace evidente que hay que tener instalados y correctamente configurados dos programas completamente diferentes e independientes en su ordenador para desarrollarlos: editor, por una parte, y compilador (incluyendo el enlazador) o intérprete por la otra, según el tipo de lenguaje. Cada vez que desee modificar y probar su programa deberá ir alternando ejecuciones entre ambos. Realmente, sería mucho más cómodo si todo ello se pudiera hacer desde un único programa, que integrase los tres. Un editor avanzado desde el que se pueda compilar, enlazar en su caso, e iniciar la ejecución de código fuente para comprobar si funciona.

Un **IDE** (I*ntegrated Development Environment* o entorno de desarrollo) es una herramienta que integra todo lo necesario para generar programas de ordenador, de manera que el trabajo sea mucho más cómodo.

Algunos ejemplos de IDE son Visual Studio, para los lenguajes C #, C ++ y Visual Basic; Eclipse para Java; Netbeans, para los lenguajes Java y Ruby; Dev-Pascal, para el lenguaje Pascal, o el Dev-C, para el lenguaje C.

La utilización de estas herramientas agiliza increíblemente el trabajo del programador. Además, los IDE más modernos van más allá de integrar editor, compilador y enlazador o intérprete, y aportan otras características que hacen aún más eficiente la tarea de programar. Por ejemplo:

- Posibilidad de hacer resaltar con códigos de colores los diferentes tipos de instrucciones o aspectos relevantes de la sintaxis del lenguaje soportado, para facilitar la comprensión del código fuente.
- Acceso a documentación y ayuda contextual sobre las instrucciones y sintaxis de los lenguajes soportados.
- Detección, y en algunos casos incluso corrección, automática de errores de sintaxis en el código, de manera similar a un procesador de texto. Así, no es necesario compilar para saber que el programa está mal.
- Soporte simultáneo del desarrollo de lenguajes de programación diferentes.
- Un depurador, una herramienta muy útil que permite pausar la ejecución del programa en cualquier momento o hacerla instrucción por instrucción, por lo que permite analizar cómo funciona el programa y detectar errores.
- Sistemas de ayuda para la creación de interfaces gráficas.

4 Introducción a Java

Hay diferentes lenguajes de programación, algunos realmente muy diferentes entre sí. Antes de seguir adelante hay que elegir uno que será el usado para practicar todos los conceptos de programación básica que veremos a partir de ahora. Una vez que se aprende a programar en un lenguaje, dominar otros lenguajes es mucho más fácil, ya que muchos de los conceptos básicos, e incluso algunos aspectos de la sintaxis, se mantienen entre diferentes lenguajes de nivel alto.

Uno de los lenguajes más populares es C. La sintaxis de Java y de C# está basada en la sintaxis de C.

En este módulo, los lenguajes con los que aprenderemos a programar serán **Java** y **C#**.

4.1 Características relevantes del Java

Desafortunadamente, no existe el lenguaje perfecto, sin ningún inconveniente y que sea ideal para crear cualquier tipo de programa. Siempre hay que llegar a un compromiso entre ventajas e inconvenientes. De hecho, la elección misma de qué lenguaje hay que usar puede llegar a condicionar enormemente el proceso de creación de un programa, y no es sensato usar siempre el mismo para resolver cualquier problema. En cualquier caso, vale la pena comentar los motivos por los que se considera interesante usar Java:

- **Es uno de los lenguajes más usados.** Java fue creado en 1995 por la firma Sun Microsystems, que en 2009 fue comprada por la empresa de bases de datos Oracle. Su propósito era ofrecer un lenguaje al menos ligado posible a la arquitectura sobre la que se ejecuta. Esto lo convirtió en sus inicios en el mecanismo más versátil existente para ejecutar aplicaciones sobre navegadores web. Desde entonces, su popularidad ha ido en aumento también como lenguaje para crear aplicaciones de escritorio, y actualmente es uno de los lenguajes más utilizados en este campo. Esto hace que la demanda de profesionales que lo dominen sea muy alta y que tenga una gran aceptación y cantidad de documentación disponible.

- **De nivel alto con compilador estricto**. Java es un lenguaje de nivel bastante alto, con todas las ventajas que ello implica. Adicionalmente, su compilador es especialmente estricto a la hora de hacer comprobaciones sobre la sintaxis empleada y cómo se manipulan los datos que se están tratando en el programa. Si bien esto a veces puede parecer un poco molesto cuando se producen ciertos errores de compilación, en realidad es una ventaja, ya que enseña al programador a tener más grado de control sobre el código fuente que genera, de forma que sea correcto.

- **Multiplataforma**. Uno de los factores decisivos en la popularidad de Java es que sus programas se pueden ejecutar en cualquier plataforma sin necesidad de volver a compilar. Una vez el código fuente se ha compilado una vez, el bytecode resultante puede ser llevado a otras plataformas basadas en otro tipo de procesador y continuará funcionando. Sólo es necesario disponer del intérprete correspondiente para la nueva plataforma. Esto homogeneiza enormemente el aprendizaje del lenguaje independientemente de la plataforma que utiliza para estudiarlo.

- **Orientado a objetos**. Este es el nombre de una metodología avanzada, muy popular y útil, para diseñar programas. Es un mecanismo de nivel muy alto para acercar la forma en que se hacen los programas al método de pensamiento humano. Más adelante en el curso veremos programación orientada a objetos.

- **Interpretado con *bytecode***. Esta es una característica derivada de que sea multiplataforma. Al tratarse de un lenguaje interpretado, es necesario disponer del intérprete correctamente instalado y configurado en cada máquina donde quiera ejecutar su programa. Esto significa que hay un programa más que debemos configurar correctamente en el sistema. Esto también hace que la ejecución de los programas en Java no siga el proceso típico de cualquier otra aplicación (por ejemplo, ejecutarlo desde

línea de comandos o hacer doble clic en la interfaz gráfica). No hay un archivo que se pueda identificar claramente como ejecutable.

4.2 Creación y ejecución de programas Java

Por ahora se trataremos el caso de la creación de programas sencillos que se compongan de un único archivo de código fuente.

Este apartado se centra en mostrar detalladamente cómo se crea y ejecuta un programa en lenguaje Java.

Dado que Java es un lenguaje interpretado, las herramientas que necesita son:

- Un editor de texto simple cualquiera.
- Un compilador del lenguaje Java, para generar *bytecode*.
- Un intérprete de Java, para poder ejecutar los programas.

Disponer de un editor de texto es el menos problemático, ya que todos los sistemas operativos de propósito general suelen tener instalado algún predeterminado. Ahora bien, cuando se edita un archivo de código fuente, hay que asignarle una extensión específica de acuerdo con el lenguaje de programación empleado, de manera que pueda ser fácilmente identificado como tal.

La extensión de los archivos de código fuente en Java es .java.

Otros lenguajes tienen otras extensiones. A título de ejemplo, en lenguaje C los archivos tienen la extensión .c, en ensamblador .asm, en Perl .pl, etc. Es importante que al editar código fuente desde cualquier editor de texto guarda el archivo con esta extensión y no la que te ofrezca por defecto (normalmente, .txt).

En el caso concreto de Java, hay una convención a la hora de dar nombre a un archivo de código fuente. Se suele usar *UpperCamelCase* (notación de camello con mayúsculas). Esta corresponde a usar sólo letras consecutivas sin acentos (ni espacios, subrayados o números), y en el que la inicial de cada palabra usada sea siempre en mayúscula. Esto no es estrictamente imprescindible, pero sí muy recomendable, ya que es el estilo de nomenclatura que siguen todos los programadores de Java y la que se encuentra en documentación, guías u otros programas. Además, en algunos sistemas, el uso de caracteres especiales, como los acentos, puede llevar a errores de compilación.

Algunos ejemplos de nombres de archivos de código fuente aceptables son: Prueba.java, HolaMundo.java, MiPrograma.java, etc.

El compilador y el intérprete de Java son dos programas que deberemos tener instalados en el ordenador para poder programar en Java. Hay varios tipos, de diferentes fabricantes, pero lo más recomendable es instalar el que proporciona de manera gratuita el actual propietario de Java, la empresa Oracle, en su página de descargas. Hay diferentes versiones del entorno de trabajo con Java, una opción es trabajar con la Java SE (*Java Standard Edition*).

La página de descargas de Oracle es https://www.java.com/ES/download/ . Hay versiones de JDK para diferentes procesadores y sistemas operativos. Hasta hace poco la licencia era siempre gratuita, pero esto ha cambiado y está sujeta a restricciones impuestas por Oracle.

También es posible utilizar **OpenJDK** en lugar de Java SE como alternativa open-source.

El compilador de Java está incluido dentro del llamado **JDK** (*Java Development Kit*). Este proporciona una serie de ejecutables vía línea de comandos que sirven para hacer diferentes tareas con código fuente Java.

El programa que pone en marcha el compilador es el ejecutable llamado **javac** (en un sistema Windows, javac.exe). Por lo tanto, para el archivo con código fuente Java llamado HolaMundo.java habría que abrir una línea de instrucciones y ejecutar:

```
javac HolaMundo.java
```

El fichero con *bytecode* resultante del proceso de compilación se llama igual que el archivo de código fuente, pero con la extensión .class.

Una vez disponemos del fichero con *bytecode*, este sólo puede ser ejecutado con la ayuda del intérprete de Java, conocido popularmente como la **JVM** (*Java Virtual Machine)*. Esta incluye dentro del paquete llamado **JRE** (*Java Runtime Environment)*. Asimismo, el JRE está incluido dentro del JDK.

El ejecutable que pone en marcha el intérprete de Java se llama java (en un sistema Windows, java.exe). Una vez se dispone del archivo de *bytecode* HolaMundo.class se puede ejecutar desde la línea de comandos haciendo:

```
java HolaMundo
```

Vemos que no se especifica ninguna extensión. Él solo deduce que la extensión debe ser .class. Inmediatamente, el programa se pondrá en marcha.

Aunque este es el entorno básico de desarrollo de Java, afortunadamente hay varios IDE que soportan este lenguaje y concentran en un único entorno este software. Utilizaremos Eclipse a lo largo del curso para trabajar con Java.

Finalmente, es importante destacar que en el caso de que sólo desee ejecutar un programa que ya ha sido desarrollado por un tercero, sea mediante un IDE o no, la única herramienta que se debe tener instalada en el ordenador para poder ejecutar el archivo .class resultante es la máquina virtual de Java, el JRE, con la versión adecuada. No es necesario instalar nada más.

4.3 Hello, world!

Dentro del ámbito de la programación es tradición que el primer programa que se escribe y se ejecuta cuando se inicia el estudio de un nuevo lenguaje sea el llamado "¡Hola, mundo!" (Originalmente en inglés, "Hello, world! "). Esta tarea es un simple ejercicio de copiar el código fuente del programa, por lo que ni siquiera hay que entender aún la sintaxis del lenguaje. El objetivo principal de este programa es ver que el entorno de trabajo se encuentra correctamente instalado y configurado, ya que por su sencillez es difícil que dé problemas. Además, también os hace servicio como plantilla de la estructura básica de un programa en el lenguaje escogido y permite repasar la estructura y algunos de los elementos básicos.

El código fuente para la versión en Java es el siguiente:

```
// Programa "¡Hola, mundo!" en Java
public class HolaMundo {
  public static void main (String[] args) {
```

```
    System.out.println ("Hola, mundo!");
  }
}
```

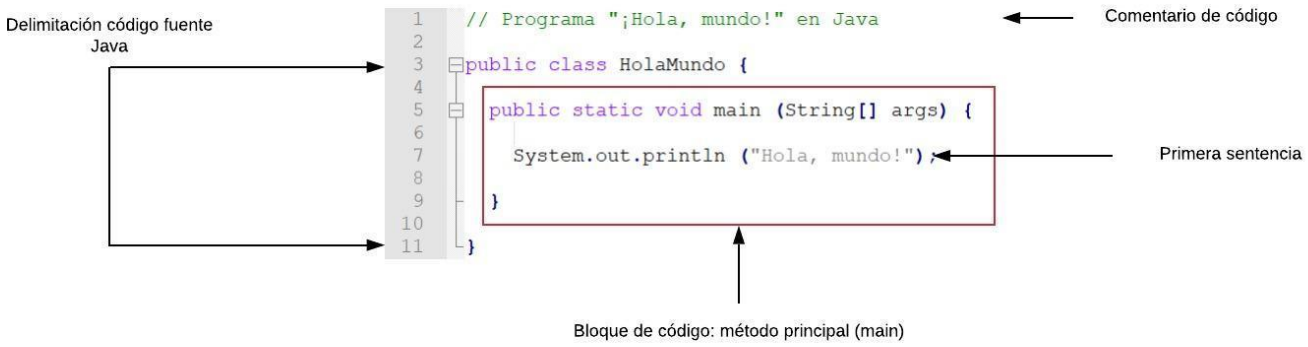
Reto 1: copia el código fuente de este programa en un fichero llamado HolaMundo.java y ejecútalo en tu entorno de trabajo. El resultado de la ejecución debería ser que por pantalla se muestre la frase *Hola, mundo!*.

El lenguaje Java es sensible a mayúsculas y minúsculas. El significado de los términos varía según como están escritos. Por lo tanto, es imprescindible que tanto el código fuente como el nombre del archivo se escriban exactamente tal como se muestran.

Puedes encontrar una lista del código fuente del programa “Hello, world!" para diferentes lenguajes en https://www.scriptol.com/programming/hello-world.php

5 Estructura y bloques fundamentales

Figura 3 Elementos del código fuente del programa "Hola, mundo!"



Bloques de un programa Java

5.1 Paquete

Sería la primera línea de código del fichero, indica el paquete al que pertenece la clase que vamos a escribir a continuación. En el caso del programa “Hola, mundo!” no hemos incluido el programa en ningún paquete.

```
package mipaquete;
```

5.2 Importación de bibliotecas o librerías

Una biblioteca es un conjunto de extensiones al conjunto de instrucciones disponibles cuando genere un programa. Para poder usar estas instrucciones adicionales es necesario que, dentro del código fuente, como preámbulo, se declare la *importación* de la biblioteca. En caso contrario, las extensiones no están disponibles. En el caso del programa "Hola, mundo!", al ser muy sencillo, no es necesario importar ninguna librería. Se podrán importar paquetes completos o clases concretas dentro estos paquetes.

Se usaría la sintaxis:

```
import <nombre librería>;
```

5.3 Indicador de inicio del código fuente. Clase

El código fuente donde empieza realmente el programa en Java comienza con el texto de declaración que se muestra a continuación, en la que <NombreArchivo> puede variar, pero siempre debe corresponder exactamente con el nombre del archivo que lo contiene. Esto es esencial. En otro caso, el compilador nos dará un error. El código del programa se escribirá a continuación, siempre entre llaves, { … }.

```
public class <NombreArchivo> {
  ...
}
```

Este texto declara que este archivo es el inicio de lo que en nomenclatura Java llama una **clase**. Este es un término estrechamente vinculado a la orientación a objetos, pero por ahora lo usaremos simplemente para referirnos a un archivo que contiene código fuente de Java.

5.4 Comentarios al código

Opcionalmente, también se pueden escribir comentarios dentro del código fuente. Se trata de texto que representa anotaciones libres al programa, pero que el compilador no procesa. Normalmente, un comentario se identifica por ser una línea de texto libre que comienza por una combinación de caracteres especiales. Los comentarios pueden estar en cualquier parte del archivo, ya que el compilador los ignora. No sirven realmente para nada en orden a la ejecución del programa, y sólo son de utilidad para quien está editando el fichero de código fuente.

Comentar el código fuente para explicar qué hace cada parte del programa, especialmente en aquellas más complejas, es una tarea muy importante que demuestra si un programador es cuidadoso o no.

En el programa “Hola, mundo!” aparece el siguiente comentario:

```
// Programa "Hola, mundo!" en Java
```

En lenguaje Java los comentarios se escriben o bien precediéndolos con dos barras, en caso de que tengan una sola línea, o bien en el formato siguiente si ocupan más de una línea.

```
/ **
 * Este es el programa "Hola, mundo!"
 * en el lenguaje de programación Java
** /
```

5.5 Indicador de la primera instrucción para ejecutar. Método main.

Para que el ordenador sepa por dónde empezar a ejecutar instrucciones, ante todo debe saber cuál es la primera de todas. Una vez localizada, continuará ejecutando el resto de manera secuencial, por orden de aparición en el código fuente. En algunos lenguajes esto se hace implícitamente, ya que la primera instrucción es directamente la primera línea de texto que aparece en el código fuente. En el caso del Java, hay un texto que lo indica claramente.

En el Java, el bloque de instrucciones en el que se engloba la primera instrucción del programa en la mayoría de lenguajes de programación se denomina el **método principal** (main).

Este método principal engloba todas las instrucciones del programa dentro de un bloque de instrucciones, entre llaves, {…}. Antes de las claves hay una serie de texto que debe escribirse

exactamente tal como se muestra. Si no, el intérprete de Java será incapaz de encontrarlo y de iniciar correctamente la ejecución del programa. Concretamente, dirá que "No encuentra el método pricipal" (*main method not found*).

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

La primera instrucción es la primera que hay escrita justo después de la llave abierta, {. La última instrucción es la escrita inmediatamente antes de la llave cerrada, }.

5.6 Bloques de código o de instrucciones

Las instrucciones o sentencias del programa están escritas una detrás de la otra, normalmente en líneas separadas para hacer el código más fácil de entender. En algunos lenguajes, al final de cada línea hay un delimitador especial, que sirva para indicar cuando termina una sentencia y empieza otra. Con el salto de línea no es suficiente. En el caso del Java, se trata del punto y coma (**;**).

Los programas más simples, como los que veremos por ahora, sólo disponen de un único bloque de instrucciones.

Las diferentes instrucciones suelen agruparse en **bloques de instrucciones**. El inicio y el fin de cada bloque diferente quedan identificados en Java para que las instrucciones están rodeadas por llaves, {…}.

Por tanto, en este programa sólo hay una única vez primera y última, instrucción.

```
System.out.println("Hola, mundo!");
```

Aunque aún conozcamos ninguna de las instrucciones del Java ni su sintaxis, posiblemente podamos deducir que sirve para ordenar al ordenador que muestre por pantalla el texto escrito entre comillas. Es decir, que es una orden sobre el componente de entrada / salida (consola). Esta es una instrucción muy útil. En este código también se puede apreciar que en el lenguaje Java las instrucciones terminan con un punto y coma (**;**).

6 Tipos de datos

El propósito principal de todo programa de ordenador, en última instancia, es procesar datos de todo tipo. Para lograr esta tarea, el ordenador debe almacenar los datos en la memoria, para que posteriormente el procesador las pueda leer y transformar de acuerdo con los propósitos del programa.

El término **dato** indica toda información que utiliza el ordenador en las ejecuciones de los programas.

Aunque a veces se usa el término *datos*, en plural, como si fuera algo en general, hay que tener en cuenta que dentro de un programa cada dato que se quiere tratar es un elemento individual e independiente. Por ejemplo, en un programa que suma dos números cualesquiera hay tres datos con los que trabaja: los dos operandos iniciales y el resultado final.

Estrictamente hablando, cualquier información se puede transformar en datos que pueda entender y manipular un ordenador. Un dato individual dentro de su programa puede ser un documento de texto, una imagen, un plano de un edificio, una canción, etc. Sólo hay que ver la

inmensa variedad de programas que hay hoy en día para hacerse una idea. Ahora bien, uno de los puntos importantes es que dentro de un programa los datos se pueden clasificar dentro de diferentes categorías: los *tipos de datos*.

Un **tipo de dato** es la definición del conjunto de valores válidos que pueden tomar unos datos y el conjunto de transformaciones que se puede hacer.

Por ejemplo, en su día a día tratamos a menudo con datos que tienen como característica común el hecho de que se pueden representar mediante números: una distancia, una edad, un periodo de tiempo, etc. Se puede decir que una ciudad está a 8 km de otra, que alguien tiene 30 años o que han pasado 15 días desde un evento. Por lo tanto, los valores 8, 30 o 15 formarían parte de un mismo tipo de dato.

También como ejemplo, en contraposición a los datos de tipo numérico, otra información que tratamos habitualmente es texto en palabras: el nombre de una ciudad, de una calle, de una institución, etc. En este caso, puede estar hablando de "Teruel", de "Calle Pablo Monglió", de "IES Segundo de Chomón", etc. En este caso, los datos se representan con símbolos diferentes y expresan otras ideas. Por lo tanto, formarían parte de otro tipo de dato.

Una propiedad importante para ver si ambos datos son de tipos diferentes es si el conjunto de valores con el que se pueden expresar también es diferente.

Como cada dato dentro de su programa siempre debe pertenecer a algún tipo, precisamente parte de la labor de un programador es identificar cuál es el tipo que se acerca más a la información que desea representar y procesar. Establecer cuál es el tipo de un dato implica que se establecen un conjunto de condiciones sobre ese dato a lo largo de todo el programa. Una de las más importantes es el efecto sobre la forma en que este dato se representará, tanto internamente dentro del hardware del ordenador como a la hora de representarla en el código fuente de sus programas. Recordemos que, por su naturaleza de sistema digital, todos los datos que hay dentro de un ordenador se codifican como secuencias binarias.

6.1 Tipos de datos primitivos

Cada lenguaje de programación incorpora sus tipos de datos propios y, aparte, casi siempre ofrece mecanismos para definir otros nuevos partiendo de tipo de datos ya existentes. Por lo tanto, se tiene que elegir entre todos los que ofrece el lenguaje cuál se acerca más a la clase de información que desea tratar. Ahora bien, desgraciadamente no se puede garantizar que las instrucciones básicas definidas en la sintaxis de un lenguaje de programación sean capaces de tratar de manera directa cualquier dato. Esto sólo sucede con un conjunto limitado de tipos de datos muy simples, llamados ***tipos primitivos***.

Los **tipos primitivos de datos** son los que ya están incorporados directamente dentro de un lenguaje de programación, y son usados como piezas básicas para construir otras más complejos.

Los lenguajes de programación identifican cada tipo con una palabra clave propia.

El uso de tipos primitivos puede variar también entre lenguajes. De todos modos, hay cuatro que se puede considerar que, de una manera o de otra, todos los lenguajes los soportan. El motivo es que estos tipos están estrechamente ligados a los tipos de secuencias binarias que normalmente un ordenador puede procesar y representar directamente dentro de su hardware. Se trata de los números **enteros**, los **reales**, los **caracteres** y los **booleanos**.

6.1.1 Booleanos

El tipo de dato **booleano** representa un valor de tipo lógico para establecer la certeza o falsedad de un estado o afirmación.

- Ejemplos de literales de un dato booleano: true (verdadero) o false (falso). No hay otro.
- Ejemplos de datos que se suelen representar con un booleano: interruptor encendido o apagado, estar casado, tener derecho de voto, disponer de carnet de conducir B1, la contraseña es correcta, etc.

Un literal de tipo booleano se representa con las palabras en inglés: ***true***, ***false***. En un IDE, normalmente este texto queda resaltado en un color especial para que quede claro que se ha escrito un literal de tipo booleano. Por ejemplo, el programa siguiente en Java muestra los literales de este tipo por pantalla. Pruébalo en tu entorno de trabajo. Recuerda que debe estar dentro de un archivo llamado LiteralesBooleanos, y que el nombre no puede tener acentos.

```
public class LiteralesBooleanos {

	public static void main(String[] args) {

		System.out.println(true);
		System.out.println(false);

	}

}
```

6.1.2 Enteros

La palabra clave para identificar este tipo de dato en Java es ***int***.

El tipo de dato **entero** representa un valor numérico, positivo o negativo, sin decimal.

- Ejemplos de literales enteros: 3, 0, -345, 138.764, etc.
- Podemos introducir literales enteros en notación binaria y hexadecimal: 0b0001, 0xA, etc.
- Ejemplos de datos que se suelen representar con un entero: edad, día del mes, año, número de hijos, etc.

Un literal de tipo entero se representa simplemente escribiendo un número sin decimales.

```
public class LiteralesEnteros {

	public static void main(String[] args) {

		System.out.println(3);
		System.out.println(0);
		System.out.println(-345);
		System.out.println(138_764);
		System.out.println(0xA);
		System.out.println(0b0011_1010);
	}
}
```

6.1.3 Reales

La palabra clave para identificar este tipo de dato a Java es ***double***.

El tipo de dato **real** representa un valor numérico, positivo o negativo, con decimales.

- Ejemplos de literales reales: 2.25, 4.0, -9653.3333, 100.0003, etc.
- También podemos usar la notación científica, por ejemplo, 1.25e2.
- Ejemplos de datos que se suelen representar con un real: un precio en euros, el récord mundial de los 100 m lisos, la distancia entre dos ciudades, etc.

Para referirse a un dato de tipo real, este incluye siempre sus decimales con un punto (.). En los ejemplos vemos el detalle del valor 4.0. Los números reales representan valores numéricos con decimales, pero en su definición nada impide que el decimal sea 0. Por lo tanto, estrictamente, el valor 4 y el valor 4.0 corresponden a tipos de datos diferentes. El primero es un valor para un tipo de dato entero y el segundo para uno real.

```
public class LiteralesReales {

	public static void main(String[] args) {

		System.out.println(02.25);
		System.out.println(4.0);
		System.out.println(-9653.3333);
		System.out.println(100.0003);
		System.out.println(1.25e2);
	}
}
```

Para saber más acerca de las diferencias entre la representación de enteros y reales, puedes buscar información sobre los formatos "Complemento a Dos" y "IEEE 754".

¿Qué sentido tiene el tipo entero si con el tipo real ya podemos representar cualquier número, con decimales y todo? ¿No es un poco redundante? De entrada, puede parecer que sí, pero hay un motivo para diferenciarlos. Sin entrar en detalles muy técnicos, representar y realizar operaciones con enteros internamente dentro del ordenador (en binario) es mucho más sencillo y rápido que con reales. Además, un entero requiere menos memoria. Es posible que en un programa no sea perceptiblemente más lento o más rápido por el simple hecho de hacer una operación entre dos datos de tipo real en lugar de entero, pero puede serlo si el programa va a procesar una gran cantidad de información o realizar un gran número de cálculos. Es una buena costumbre usar siempre el tipo de dato que se adapte exactamente a sus necesidades. Si un dato no tiene sentido que tenga decimales, como un número de año o de mes, deberíamos usar enteros.

6.1.4 Caracteres

La palabra clave para identificar este tipo de dato en Java es ***char***.

El tipo de dato **carácter** representa una unidad fundamental de texto usada en cualquier alfabeto, un número o un signo de puntuación o exclamación.

- Ejemplos de literales de tipo carácter: 'a', 'A', '4', '>', '?', 'Ñ', etc.
- Ejemplos de datos que se suelen representar con un carácter: cada uno de los símbolos individuales de un alfabeto.

Para referirse a un dato de tipo carácter, ésta se rodea de comillas simples ('). Por tanto, no es lo mismo el carácter '4' y que el valor entero 4, ya que pertenecen a tipos diferentes. El primero lo usará para hacer referencia a una representación textual, mientras que el segundo es el concepto propiamente matemático. Debemos tener cuidado, ya que entre las comillas simples sólo puede haber un solo carácter, o Java dirá que la sintaxis no es correcta.

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

		System.out.println('a');
		System.out.println('A');
		System.out.println('4');
		System.out.println('>');
		System.out.println('?');
		System.out.println('Ñ');

	}
}
```

Sistemas de representación de los caracteres

A la hora de decidir cómo se representa internamente un carácter, se utilizan diferentes tablas de códigos establecidos, entre los que destacan los códigos ASCII y UNICODE. El código ASCII fue uno de los más extendidos, sobre todo por la gran proliferación de ordenadores personales. El código ASCII inicial representaba 128 caracteres en lugar de 256. Cuando se necesitaron caracteres especiales para los diferentes idiomas, se amplió con 128 caracteres más, y se constituyó el código ASCII extendido.

Desgraciadamente, el código ASCII tiene una pega, y es que sólo permite representar alfabetos occidentales, por lo que los programas que la usan para representar sus datos de tipo carácter son incompatibles con sistemas con otros alfabetos, como todos los asiáticos, el cirílico, etc. Por este motivo, posteriormente se creó la tabla de codificación UNICODE, que permite codificar hasta 65.536 caracteres, pero manteniendo la compatibilidad con la codificación ASCII. Esto permite apoyar cualquier lengua actual, e incluso de antiguas, como los jeroglíficos egipcios. Este sistema es el que actualmente usan la mayoría de aplicaciones modernas.

Java representa sus caracteres internamente usando la tabla UNICODE

Para más información sobre los tipos de datos primitivos en Java y sus literales consultar https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

6.2 Tipos de datos complejos

En muchos casos, seguramente no será suficiente sólo escribiendo la representación textual de un tipo de dato simple. Si necesitamos escribir una frase entera del estilo "El resultado obtenido es ...". Es decir, escribir bloques de texto. Partiendo de esta necesidad, tal vez llegaremos a la conclusión de que el tipo de dato carácter no es muy útil, ya que para mostrar frases largas habría que hacerlo carácter por carácter, de manera muy molesta.

En realidad, el tipo de dato carácter no se suele usar directamente, sino que los lenguajes de programación lo usan como base para generar un nuevo tipo de dato más avanzado que sirve para procesar texto.

6.2.1.1 Cadenas de caracteres

La palabra clave para identificar el tipo de dato que representa texto en Java es **String**.

Una cadena de texto (String) es un tipo de dato que permite representar cualquier secuencia de caracteres de longitud arbitraria.

- Ejemplos de valores de una cadena de texto: "¡Hola, mundo!", "El resultado final es ...", etc.

- Ejemplos de datos que se suelen representar con una cadena de texto: cualquier mensaje de texto por pantalla.

Para referirnos a un dato de tipo cadena de texto, la escribimos entre comillas dobles ("). Hay que tener cuidado, ya que no es lo mismo 'a' que "a". La primera es un literal que representa un dato de tipo carácter y la segunda es de tipo cadena de texto.

Las cadenas de texto no son un tipo primitivo.

Cada lenguaje ofrece su manera de gestionar cadenas de texto y representarlas por pantalla. Desde el ejemplo del programa "¡Hola, mundo!", ya hemos visto que las instrucciones de salida básica de Java para pantalla también pueden mostrar cadenas de texto de Java sin problemas, exactamente igual que se hace con los tipos primitivos.

```
System.out.println("¡Hola, mundo!");
```

7 Operadores y expresiones

Dentro de un programa los datos acabarán usando alguno de los cuatro tipos primitivos. Una vez se ha elegido qué tipo usará para representar una información concreta, ya se ha limitado el conjunto de valores que puede tomar. Ahora bien, otro punto muy importante de esta decisión es que también se habrá fijado de qué manera es posible transformarlas. Para cada tipo de dato, sus valores sólo se pueden transformar de unas maneras muy concretas: mediante el que formalmente se conoce como ***operaciones***. Cada tipo de dato establece el conjunto de operaciones admisibles sobre los valores que comprende.

Un lenguaje de programación que sólo permite operaciones entre datos del mismo tipo se llama **fuertemente tipado**. Java lo es.

Normalmente, sólo se pueden hacer operaciones entre datos que pertenezcan al mismo tipo. Como se suele decir, no se pueden sumar peras con manzanas.

El símbolo como se identifica cada una de las operaciones dentro de un tipo de dato es su **operador**. Los datos sobre las que se aplica una operación son sus **operandos**.

7.1 Operadores booleanos

El tipo de dato booleano debe su nombre al álgebra de Boole, que es el conjunto de normas y operaciones que rigen cómo se pueden combinar los valores verdadero y falso.

Las operaciones básicas que se pueden hacer sobre datos de este tipo son de dos tipos: lógicas y relacionales.

Los operadores lógicos que se pueden aplicar son:

- **NOT** representado por **!**
- **AND** representado por **&&**
- **OR** representado por **||**

- **XOR** representado por **^**

Para este tipo de operaciones, una manera sencilla de plasmar su resultado al ser aplicados sobre datos de tipo booleano es usar una tabla de verdad.

Las **tablas de verdad** son una representación que nos permite averiguar, con claridad y fiabilidad, todos los resultados posibles de una operación a partir de todas las combinaciones de los valores posibles de los operandos.

A	B	A&&B	A||B	!A	A^B
false	false	false	false	true	false
false	true	false	true	true	true
true	false	false	true	false	true
true	true	true	true	false	false

Resumiendo:

- La negación da como resultado el valor contrario.
- AND es cierto si los dos operandos son ciertos.
- OR es cierto si alguno de los operandos lo es.
- XOR es cierta sólo si uno de los operandos es cierto, pero no si lo son los dos.

Visto con un ejemplo más concreto, las operaciones lógicas siguen el razonamiento siguiente. Suponga que A y B son interruptores que pueden estar encendidos (true) o no (false). La negación es como conmutar un interruptor. Si estaba encendido pasa a estar apagado y viceversa. El operador AND es como preguntar "¿Es cierto que los dos interruptores están encendidos?". Basta que uno de los dos esté apagado para que la respuesta sea que no (false). El operador OR es como preguntar "¿Es cierto que alguno de los dos interruptores está encendido?". Si uno de los dos, lo que sea, lo está, la respuesta será sí (true). El operador XOR es como preguntar "¿Es cierto que uno de los dos interruptores está encendido y el otro apagado?"

```
public class OperadoresBooleanos {

	public static void main(String[] args) {

		System.out.println(true && true);
		System.out.println(true && false);
		System.out.println(true || false);
		System.out.println(false || false);

		System.out.println(!true);
		System.out.println(true ^ false);
		System.out.println(true ^ true);
		System.out.println(false ^ false);
	}

}
```

7.2 Operadores relacionales

Revisando algunas definiciones matemáticas, nos damos cuenta que los números conforman un conjunto ordenado. Cada uno tiene una posición relativa. Sabemos que el 2 es menor que el 4 y que el 6 es mayor que el 1. Al comparar dos números, realizamos una función de relación.

En java disponemos de los operadores relacionales para verificar si se cumple una relación. Por ejemplo el operador de equivalencia (==) nos devuelve un valor de verdadero si los operandos son iguales. Estas operaciones comparan dos valores numéricos y devuelven un valor booleano.

Operador	**Utilización**	**Resultado**
>	A > B	verdadero si A es mayor que B
>=	A >= B	verdadero si A es mayor o igual que B
<	A < B	verdadero si A es menor que B
<=	A <= B	verdadero si A es menor o igual que B
==	A == B	verdadero si A es igual a B
!=	A != B	verdadero si A es distinto de B

También podemos comparar char, ya que internamente está guardados como números. La comparación dará como resultado el orden alfabético.

Entre los booleanos solo se permiten los operadores de equivalencia, es igual (==) o es distinto (!=)

```
public class Relaciones {

	public static void main(String[] args) {

		System.out.println(-3 > 5);
		System.out.println(6 == 6);
		System.out.println(3.0 >= .25);
		System.out.println(3.6 != 5.6);
		System.out.println('A' < 'B');
		System.out.println(true == false);
		System.out.println(true != false);

	}

}
```

7.3 Operadores aritméticos

Los operadores aritméticos entre enteros soportados en Java son:

- el cambio de signo (**-**, operador unario)
- la suma (**+**)
- la resta (**-**, operador binario)
- la multiplicación (*****)
- la división (**/**).
- el módulo o resto de la división (**%**)

Cualquier operación entre dos datos de tipo entero siempre da como resultado un nuevo dato también de tipo entero. Ahora bien, como que los números enteros no disponen de decimales, hay que tener presente que, en el caso de la división, el resultado se trunca y se pierden todos los decimales. El operador módulo (%) devuelve el resto de la operación de división entre enteros.

Las operaciones que se pueden hacer entre datos de tipo real son exactamente las mismas que entre enteros. La única excepción es la operación módulo, que deja de tener sentido, ya que ahora la división sí se hace con cálculo de decimales

```
public class OperadoresArimeticos {

	public static void main(String[] args) {
		// Operaciones aritméticas entre enteros
		System.out.println("Operaciones aritméticas entre
enteros");
		System.out.println(-3);
		System.out.println(-3 + 5);
		System.out.println(6 - 6);
		System.out.println(3 * 5);
		System.out.println(20 / 4);
		System.out.println(21 / 4);
		System.out.println(21 % 4);
		// Operaciones aritméticas entre reales
		System.out.println("Operaciones aritméticas entre
reales");
		System.out.println(-3.1);
		System.out.println(-3.5 + 5.0);
		System.out.println(6.9 - .9);
		System.out.println(3.0 * 5.5);
		System.out.println(20.0 / 4.0);
		System.out.println(21.0 / 4.0);

	}

}
```

7.4 Construcción de expresiones

Como hemos visto, dado un conjunto de datos de un tipo concreto, el mecanismo principal para transformarlas y obtener otras nuevas es la aplicación de operaciones entre varios operandos. Hasta el momento, sin embargo, los ejemplos de aplicación de operaciones se han limitado a una única operación, con el número de operandos correspondientes según si ésta era unaria o binaria (uno o dos). Ahora bien, en muchos casos, es útil poder aplicar de una sola vez un conjunto de operaciones diferentes sobre una serie de datos.

Una **expresión** es una combinación cualquiera de operadores y operandos.

Para construir una expresión, es necesario que ésta sea correcta en dos niveles, sintáctica y semánticamente, de forma que se respete el significado de los datos usados como operandos y sus operadores. En cualquier caso, las expresiones siempre se escriben en una sola línea, como cualquier texto legible en español, ordenando los elementos de izquierda a derecha y de arriba a abajo.

Desde el punto de vista **sintáctico**, las normas básicas de construcción de expresiones usando literales son las siguientes:

1. Se considera que un literal solo es en sí mismo una expresión.
2. Dada una expresión correcta E, también lo es escribirla entre paréntesis: (E).
3. Dada una expresión correcta E y un operador unario cualquier *op* , *op* E es una expresión correcta.
4. Dadas dos expresiones correctas E1 y E2 y un operador binario cualquier *op* , E1 *op* E2 es una expresión correcta.

Estas normas son un caso general válido para representar cualquier combinación de literales y operaciones, cualquiera que sea la longitud. Ejemplos de expresiones que las siguen, mostradas de manera que se aplican de manera acumulativa, son:

- 6, Para la primera regla.
- (6 + 5), Para la segunda regla, en la que E es 6 + 5.
- -4, Para la tercera regla, en la que *op* es -y E es 4.
- 6 + 5, Para la cuarta regla, en la que E1 es 6, E2 es 5 y *op* es +.
- (6 + 5) * -4, Para la cuarta regla, en la que E1 es (6 + 5), E2 es -4 y *op* es *.

Desde la vertiente **semántica**, las normas que siempre debe cumplir una expresión son las siguientes:

1. Cualquier operación siempre debe ser entre datos del mismo tipo.
2. La operación usada debe existir para el tipo de dato.

El primer punto es muy importante y hay que tener cuidado al aplicarlo. Por ejemplo, si partimos de literales, escribir un 6 corresponde al literal que representa el valor numérico "seis" dentro del tipo de dato entero. El texto 6.0 es el mismo, pero para los reales, y ‘6’ (nótese las comillas simples) para los caracteres. Ahora bien, a pesar de que representan el mismo concepto, no es posible realizar operaciones entre estos tres literales, ya que pertenecen a tipos diferentes. Sin embargo, tenemos que ver como con pequeños detalles de escritura se puede establecer a qué tipo de dato pertenece cada literal en su código fuente directamente. De hecho, todos los ejemplos de valores posibles dado un tipo de dato, mostrados en la sección anterior, son literales.

Las expresiones son correctas sintácticamente, pero no semánticamente. Por tanto, no se consideran expresiones válidas.

- 5.3 == ‘4’ : Aunque reales y caracteres disponen de la operación igualdad, los dos literales pertenecen a tipos diferentes (primera regla).
- true + false, La operación suma no existe para los booleanos (segunda regla).
- -‘g', La operación cambio de signo no existe para los caracteres (segunda regla).
- 5 == false, Aunque en booleanos y enteros existe la igualdad, la operación es entre tipos diferentes (primera regla).

- 5 || 4.0, Se hace una operación entre tipos diferentes y, aparte, la operación OR no existe en los enteros ni en los reales (regla primera y segunda).

7.5 Evaluación de expresiones

Entendemos por *evaluar* una expresión ir aplicando todos sus operadores sobre los diferentes datos que la conforman hasta llegar a un resultado final, que es el dato resultante. La evaluación siempre se comienza calculando las expresiones que se encuentran entre paréntesis, en orden de más interno a más externo. Una vez han desaparecido todos los paréntesis, entonces se aplican las operaciones según su orden de precedencia.

El **orden de precedencia** de un conjunto de operadores es la regla usada para establecer de manera no ambigua el orden en que deben resolver las operaciones dentro de una expresión.

Las operaciones con orden de precedencia mayor evalúan antes que las de orden menor. Para las operaciones que se han descrito hasta ahora, este orden es el siguiente. En caso de empate, se resuelve la expresión ordenadamente de izquierda a derecha.

Orden de prioridad de los operadores

Orden	operación	operador
1	Cambio de signo	- (unario)
2	Producto, división y módulo	* / %
3	Suma y resta	+-
4	Relaciones de comparación	> < <= >=
5	Relaciones de igualdad	== !=
6	Negación	! (unario)
7	AND	&&
8	OR	||

Aunque existe este orden, es muy recomendable que todas las expresiones basadas en un operador binario siempre se incluyan entre paréntesis cuando deben combinarse con nuevos operadores para generar expresiones complejas, con vistas a mejorar la legibilidad de la expresión general.

```
public class PrecedenciaOperadores {

	public static void main(String[] args) {
		//Estudia el orden de preferecia de los operadores
		System.out.println(3+5==4*2&&1.2>1.15||'a'=='b');
		System.out.println(3+5==4*2&&1.2<1.15||'a'=='b');
		System.out.println((3+5==4*2)&&(1.2>1.15||'a'=='b'));
		System.out.println((3*5==4+2&&1.2>1.15)||('a'=='b'));
		//Practica tú mismo modificando las expresiones
		//y quitando y poniendo paréntesis

	}
}
```

Veamos un ejemplo de evaluación de la primera expresión del bloque de código anterior

```
Ejemplo de evaluación de expresiones

3+5==4*2&&1.2>1.15||'a'=='b'

3+5==8&&1.2>1.15||'a'=='b'

8==8&&1.2>1.15||'a'=='b'

8==8&&true||'a'=='b'

true&&true||'a'=='b'

true&&true||false

true||false

true
```

7.6 Desbordamientos y errores de precisión

Un hecho muy importante a tener en cuenta cuando evalúe expresiones aritméticas entre datos de tipo numérico (enteros y reales), es que estos tipos tienen una limitación en los valores que pueden representar. Al haber infinitos números, esto quiere decir que para representarlos todos dentro del ordenador necesitarían secuencias binarias de infinita longitud, lo que es imposible. Por lo tanto, los lenguajes de programación sólo pueden representar un rango concreto en de números tanto enteros como reales. Si se supera este rango, se dice que ha producido un desbordamiento aritmético (*aritmetic overflow*) y el resultado de la expresión será incorrecto.

En el caso de los reales, además de existir el infinito, existe la particularidad de que, dado un número real, éste puede tener infinitos decimales. Esto significa que tampoco es posible representar cualquier secuencia de decimales con un número real. Para ciertos valores en realidad se hacen redondeos. La consecuencia directa de este hecho es que al hacer cálculos con datos reales pueden aparecer errores de precisión. Lo tendremos que tener en cuenta en programas de cálculo complejos, en los que cada decimal es importante.

Teniendo en cuenta la circunstancia de que el rango de datos que se puede representar usando tipos numéricos está acotado, muchos lenguajes de programación en realidad dividen los tipos enteros y reales en diferentes categorías, cada una definida con una palabra clave diferente. Cada una de estas categorías se considera un nuevo tipo primitivo diferente. El rasgo distintivo para cada caso es el rango de valores que puede alcanzar y la longitud de su representación interna en binario (número de bits).

La siguiente tabla muestra los tipos primitivos, su tamaño y el rango de números que se pueden representar.

Tipo	**Palabra clave Java**	**Tamaño (bits)**	**Rango**
Byte	byte	8	-128 a 127
entero corto	short	16	-32768 a 32767
entero simple	int	32	-2147483648 a 2147483647
entero largo	long	64	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,808

real de simple precisión	float	32	-3.40292347 * 10 ^38 a 3.40292347 * 10 ^38
real de doble precisión	double	64	-4.94065645841246544 * 10 ^24 a 1.79769313486231570 * 10 ^308

Los literales que usa Java por defecto, y los que se han usado hasta ahora para describir datos de tipo entero y real se corresponden a los tipos entero simple (int) y real de doble precisión (double). Por tanto, a partir de ahora, siempre que se hable de tipos de datos enteros y reales en general, el texto se referirá a estos dos tipos primitivos en concreto.

En Java, un float, puede representar fielmente hasta 6 o 7 decimales. Un double hasta unos 15.

Si se quiere indicar que un literal es de otro tipo diferente de estos dos, hay que indicarlo explícitamente añadiendo al final el carácter 'L', para escribir un long, o 'F', por un float. En Java no se puede explicitar que un literal es un entero corto. Por ejemplo:

```
5400000000L
3.141592F
```

Siempre que se evalúa una operación binaria en la que aparecen literales numéricos de tipo diferente, el tipo del resultado será el mismo que el operando con más rango.

Por ejemplo, el resultado de evaluar la expresión (2 + 5) * 40L es un entero de tipo long (concretamente, 280L).

¿Qué ocurrirá al tratar de ejecutar la siguiente sentencia? ¿Cómo puede corregirse el problema?

```
System.out.println(3000000000);
```

8 Variables

Hasta el momento hemos trabajado con literales dentro del código fuente de los programas. Ahora bien, esta es sólo la punta del iceberg a la hora de operar con datos dentro de un programa. Si nos limitamos a esto, la función del ordenador no va más allá de ser una simple calculadora más complicada de usar. De hecho, los literales sólo son suficientes en los casos que debe representar datos con un valor conocido en el momento de escribir el programa y que nunca variará a lo largo de la ejecución, para ninguna de las ejecuciones.

Ahora bien, en muchos casos, el valor de los datos que se quieren tratar dentro de un programa o bien es desconocido (por ejemplo, se trata precisamente del resultado que se está intentando calcular o dependerá de una respuesta del usuario) o bien el valor irá variando a lo largo del programa (por ejemplo, el precio total para pagar en una tienda virtual cuando aún no se ha finalizado la compra). Quizás incluso conocemos el valor inicial (en una tienda virtual, el precio cuando no se ha comprado nada todavía seguro que es cero), pero no se puede predecir la evolución. Otro caso que nos podemos encontrar es que deseemos guardar los datos resultantes de evaluar una expresión para usarla más adelante, en uno o varios lugares dentro del programa. En ninguno de estos casos no se pueden usar literales y lo que hay que usar es una *variable* almacenada dentro de la memoria del ordenador.

Una **variable** es un dato almacenado en la memoria que puede ver modificado su valor en cualquier momento durante la ejecución del programa.

Antes de ver cómo podemos usar variables, vale la pena hacer un repaso de los aspectos más importantes del funcionamiento de la memoria del ordenador desde el punto de vista de cómo

la usará a la hora de hacer un programa. Esto le ofrecerá una cierta perspectiva sobre algunos de los aspectos más importantes en el uso de las variables.

Supongamos que estamos en la escuela primaria y que un buen día el maestro te hace salir a la pizarra y te dice que tienes que resolver una suma de 4 números enteros de 8 cifras. Primero de todo, lo que harás es buscar un trozo de pizarra que esté libre para poder escribir, que no implique tener que borrar lo que el maestro ha escrito antes. Concretamente, necesitarás suficiente espacio para poder escribir los 4 números que te dictará el profesor y para poder escribir la solución final. Si no fueras capaz de calcular todo de memoria, también tendrías buscar espacios para poder hacer cálculos auxiliares.

Después, a medida que el maestro te va dictando los números, los vas escribiendo en la pizarra para no olvidarte. Una vez los tienes todos apuntados, ya puedes proceder a hacer las operaciones pertinentes y ver si te sale o no. Mientras hagas los cálculos, puedes ir escribiendo, si te va bien, nuevos números que representan cálculos parciales, siempre que tengas espacio en la pizarra. También puedes borrar o modificar en todo momento la información que has escrito en la pizarra. Finalmente, con la ayuda de todos estos datos, obtienes el resultado final. Una vez das el resultado final, y el profesor te dice si es correcto o no puedes eliminar la parte de la pizarra donde has escrito para dejar espacio a tus compañeros y vuelves a tu sitio.

Conceptualmente, la memoria del ordenador tal como la utilizaría un programa no es muy diferente de nosotros y de esta pizarra. Es un espacio disponible donde los diferentes programas pueden guardar libremente datos para poder llevar a cabo su tarea. Ahora bien, los dos puntos más importantes que este símil tiene en común con la memoria de un ordenador, y que debe tener en cuenta, son:

- **La memoria es un espacio compartido** entre todos los programas que se encuentran en ejecución, como el sistema operativo. Por lo tanto, antes de poder almacenar ningún dato, hay que poder buscar un espacio vacío que no se esté usando.
- **Los datos almacenados no son persistentes**, sólo existen mientras el programa está resolviendo la tarea en curso. Al terminar, se borran. En el caso de un programa, sus datos sólo están presentes en la memoria mientras éste está en ejecución. Cuando termina, los datos desaparecen totalmente. La única manera de evitarlo es que, antes de terminar la ejecución, sean traspasadas a un medio persistente (por ejemplo, a un archivo en el disco duro).

Ahora bien, por la naturaleza de sistema digital de un ordenador, la forma en que su memoria organiza todos los datos contenidos no es exactamente como una pizarra. También hay dos aspectos en los que el símil diverge y que hay que tener muy presentes:

- Recordemos que **los datos se representan en formato binario** cuando se encuentran dentro del ordenador. Un ordenador sólo puede representar internamente la información en unos y ceros, de acuerdo con el estado de los transistores de sus chips.
- De hecho, **los datos se organizan en celdas** dentro de la memoria, de una manera más bien parecida a una hoja de cálculo. Cada una puede contener 8 bits de información. Ahora bien, un dato puede ocupar más de una celda y el número exacto que ocupa depende de su tipo y del lenguaje usado. Cada celda individual se indexa con un identificador numérico llamado su *dirección*.

8.1 Declaración de variables

El lenguaje de programación ya se encarga internamente de todos los aspectos de nivel bajo vinculados a la estructura real de la memoria.

De la descripción del funcionamiento de la memoria, el punto más importante es que para poder disponer de una variable dentro de su programa, primero se debe buscar un espacio libre en la memoria, compuesto de varias celdas, que será asignado a esta variable de forma exclusiva. Afortunadamente, los lenguajes de programación de nivel alto ofrecen un sistema relativamente sencillo para hacerlo.

Toda variable dentro del código fuente de un programa debe haber sido **declarada** previamente por el programador antes de poder utilizarla.

Para declarar una variable dentro un programa, basta con especificar tres cosas: el tipo, un identificador o etiqueta único y un valor inicial. Aunque la sintaxis para declarar una variable puede variar según el lenguaje de programación, casi siempre hay que definir estas tres cosas.

El identificador nos permite diferenciar las diferentes variables que haya dentro del código fuente del programa, el tipo delimita qué valores puede almacenar y qué operaciones son aplicables, y el valor inicial el contenido que habrá luego que se ha declarado. Evidentemente, el valor inicial debe ser un dato del mismo tipo que la variable. Una vez completada correctamente la declaración, inmediatamente después de esta línea de código, ya es posible usar la variable refiriéndose al identificador elegido para leer el valor de los datos almacenados o sobrescribirlos.

Es importante inicializar las variables; no todos los lenguajes requieren asignar un valor inicial a una variable. En estos casos, simplemente no se puede predecir cuál es el valor almacenado en la memoria (la secuencia de unos y ceros en ese espacio). Puede ser cualquier valor, por lo que el resultado de cualquier operación en que intervenga esta variable será impredecible. Por este motivo, aunque el compilador acepte la sintaxis como correcta si no se hace, es un hábito muy bueno inicializar siempre las variables.

En caso de duda, siempre puede asignar el 0 como valor inicial.

La sintaxis para declarar una variable en Java es la siguiente:

```
<tipo> identificadorVariable = valorInicial;
```

Estrictamente, se considera que la parte a la izquierda del signo igual es la declaración propiamente de la variable, mientras que la parte derecha es la **inicialización**; la especificación de cuál es el valor inicial.

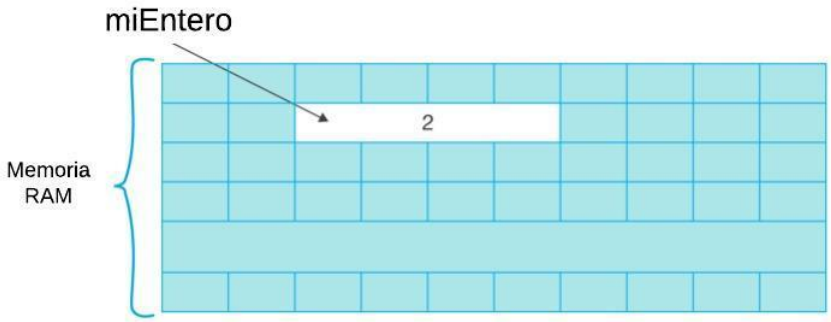
Recordemos que las palabras clave de Java para los cuatro tipos primitivos presentados son boolean, int, double y char.

La manera más sencilla de verlo es mediante un ejemplo concreto:

```
int miEntero=2;
```

Justo después de ejecutar esta instrucción, la memoria quedaría como muestra la figura. En este caso, esta variable ocupa dos celdas, ya que Java almacena los enteros en la memoria usando

32 bits. En cualquier caso, este hecho es totalmente transparente para el programador.



También es posible declarar más de una variable del mismo tipo de dato en una sola línea separando los identificadores e inicializaciones con comas. Por limpieza y para mantener un código más simple y claro se desaconseja esta notación.

```
int i = 2, j = 4;
```

8.2 Identificadores

Coloquialmente, para referirnos a un identificador también solemos usar el término *nombre*; normalmente diremos *"El nombre de una variable"*.

Un identificador toma la forma de un texto arbitrario. Podemos elegir el que más nos guste. De todas formas, es muy recomendable que siempre usemos algún texto que nos ayude a entender fácilmente qué se está almacenando en cada variable, ya que esto ayuda a entender qué hace el código fuente de su programa. Cuando un programa es largo o hace mucho que no se ha editado, incluso puede ser problemático para su propio creador volver a entender para qué servía cada variable.

¿Cuál de los siguientes programas resulta más legible?

```java
public class D {

	public static void main(String[] args) {

		int x = 30;
		int z = 6;
		System.out.print("División de " + x + " por " + z + " es igual a "+ x/z);

	}

}

public class Division {

	public static void main(String[] args) {

		int dividendo = 30;
		int divisor = 6;
		int division = dividendo/divisor;
		System.out.print("División de " + dividendo + " por " + divisor + " es
igual a "+ division);

	}

}
```

Es muy conveniente usar identificadores de variables comprensibles, aunque sean un poco más largos. Evidentemente, la clave de todo siempre está en el equilibrio. Usar un identificador que ocupe cientos letras y no se pueda leer en una sola línea de texto tampoco ayuda mucho.

Desgraciadamente, no cualquier identificador se considera válido. Cada lenguaje de programación impone unas condiciones sobre el formato que considera admisible. Aunque aquí se explica el caso concreto del lenguaje Java, no hay muchas variaciones entre los diferentes lenguajes. Un identificador:

- No puede contener espacios.
- No puede comenzar con un número.
- Se desaconseja usar acentos.
- No puede ser igual que alguna de las palabras clave del lenguaje.

Una **palabra clave (o reservada)** de un lenguaje de programación es aquella que tiene un significado especial dentro de su sintaxis y se usa para componer ciertas partes o instrucciones en el código fuente de un programa.

El conjunto de palabras reservadas en Java se enumera a continuación:

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

En esta lista hay algunas palabras que ya hemos visto, usadas para declarar variables o especificar donde empieza el código fuente o el método principal del programa: boolean, int, double, char, public, class, void, etc.

Convenciones de nomenclatura

Aunque tenemos libertad total para elegir los identificadores de las variables, al igual que ocurre con los nombres de las clases de Java, hay una convención de código para nombrarlos en Java. En este caso hay que usar *lowerCamelCase* (notación de camello minúscula). Esta notación es como *UpperCamelCase*, pero en este caso la primera palabra siempre en minúscula y no en mayúscula.

Ejemplos de identificadores de variables que siguen la convención son: divisor, resultadoDivision, miEntero, etc.

Por último, recordar que Java es sensible a mayúsculas y minúsculas, por lo que miEntero y MiEntero se consideran identificadores totalmente diferentes.

8.3 Uso de variables

La principal valía de una variable dentro de un programa es su capacidad para almacenar datos y recuperar su valor en cualquier momento a lo largo del programa.

Cuando una variable ha sido declarada, su identificador se puede usar exactamente igual que un literal dentro de cualquier expresión. El valor que representará será el del valor almacenado en memoria en ese instante para esa variable.

Ahora bien, hay que tener presente que en la mayoría de lenguajes de programación la declaración de una variable sólo tiene validez desde la línea de código donde se declara hasta encontrar el delimitador que marca el final del bloque de código donde se ha declarada. Fuera de este rango de instrucciones del código fuente, su *ámbito*, es como si no estuviera declarada.

El **ámbito de una variable** es el contexto bajo el cual se considera declarada. La región de código donde se pude usar.

Si se intenta hacer uso de un identificador que no corresponde a ninguna variable declarada (ya sea porque nunca se ha declarado o porque se hace uso fuera de su ámbito), al compilar el código fuente se producirá un error.

```java
//Movimiento Rectilineo Uniformemente Acelerado
public class MRUA {

	public static void main(String[] args) {

		/*
		Unos estudiantes de la universidad de Stuttgart han conseguido batir el récord
		Mundial de aceleración de 0 a 100 km/h en tan sólo 1,779 segundos
		con un vehículo totalmente eléctrico diseñado para las competiciones de Formula
		Student
		*/

		//Queremos cuál es la aceleración del vehículo, suponiendo que ésta es constante
		//Fórmula a aplicar: Velocidad = VelocidadInicial + Aceleración * Tiempo

		//Velocidad Inicial, en reposo
		double v0 = 0.0;
		//Velocidad final, 100Km/h
		double v = 100.0;
		//Tiempo 1.779s
		double t = 1.779;
		//Declaración de la variable para la aceleración
		double a = 0;

		//Resultado, hay que convertir los segundos en horas para tener las variables en
		//las mismas unidades de medida

		a = (v-v0)/(t/3600.0);

		System.out.println("Aceleración: " + a + " Km/h²");

	}
}
```

En este ejemplo el uso de las variables es correcto, ya que se accede a su identificador en una línea de código que se encuentra entre la línea donde se han declarado y la clave de final de bloque del método principal. El uso se ha hecho dentro de su ámbito.

Ahora bien, el uso de variables no tiene mucho sentido cuando para hacer una operación es podrían usar literales directamente y el resultado sería el mismo, como en el ejemplo anterior. La verdadera utilidad consiste en poder cambiar el valor que hay almacenado en memoria. El valor de una variable se puede modificar en cualquier momento dentro de un programa con el operador de asignación (=):

```
identificadorVariable = expresión;
```

Esta sintaxis ya la habíamos visto, ya que es prácticamente igual a la usada para establecer el valor inicial al declarar la variable. De hecho, una variable también se puede inicializar a partir de una expresión, no es necesario usar un único literal.

Si el tipo del resultado de la expresión y el de la variable en la que se asigna no es lo mismo habrá un error de compilación. En el ejemplo anterior debemos utilizar todos los literales de tipo double.

Al hacer una asignación a una variable, lo primero que sucede es que se evalúa la expresión que hay en la parte derecha. El resultado obtenido entonces se convierte inmediatamente en el nuevo valor almacenado dentro de la variable. El valor anterior se pierde para siempre, ya que queda sobrescrito. Ahora bien, recordemos que el operador de asignación (=), como cualquier otro operador binario, siempre requiere que sus operandos pertenezcan al mismo tipo de

dato. Por lo tanto, sólo es correcto asignar expresiones que evalúen un tipo idéntico al utilizado en declarar la variable.

Atención. No es lo mismo el ***operador de asignación (=)*** que el de ***igualdad (==)***. No hay que confundirlos.

Es muy importante ser conscientes de la orden para resolver una asignación. Primero hay que evaluar la expresión y después se modifica realmente el valor de la variable. Dentro de una expresión se puede usar el identificador de la misma variable a la que se está haciendo la asignación. Por lo tanto, en el lado derecho se usa su valor original, y una vez hecha la asignación, esta habrá modificado de acuerdo con el resultado de evaluar la expresión.

La mejor manera de ver el comportamiento de una variable cuando se modifica el valor es con un ejemplo típico de variable entera contador:

```
contador = contador + 1;
```

8.4 Operadores de asignación

Prácticamente lo hemos utilizado en todos los ejemplos de variables y operadores. Es el operador de asignación. Este aparece con un signo igual (**=**). Cambia el valor de la variable que está a la izquierda por un literal o el resultado de la expresión que se encuentra a la derecha.

Se asigna a la variable a el cálculo de la aceleración:

```
a = (v-v0)/(t/3600.0);
```

Aumentamos en 1 el valor de la variable contador

```
contador = contador + 1;
```

Existe una forma de simplificar la notación anterior. Es la siguiente:

```
contador += 1; //equivalente a contador = contador + 1;
```

Se junta el operador de suma con el de asignación. Este atajo se puede realizar para otras operaciones además de la suma. Es útil cuando la operación contiene como primer operando al valor de la misma variable en la que se almacena el resultado.

Operación	**Operador**	**Utilización**	**Operación equivalente**
Suma	+=	A += B	A = A + B
Resta	-=	A -= B	A = A - B
Multiplicación	*=	A *= B	A = A * B
División	/=	A /= B	A = A / B
Resto de división	%=	A %= B	A = A % B

8.5 Operadores incremento y decremento

Java soporta los operadores incremento (**++**) y decremento (**--**); respectivamente, suman y restan una unidad al valor de la variable a la que se aplican. Ambos operadores pueden ser sufijos, es decir se coloca antes del operando o posfijo que se sitúa detrás.

La sintaxis es la siguiente:

```
int contador = 0;
//Incremento
contador++;
```

```
System.out.println(contador); //mostrará 1
//Decremento
Contador--;
System.out.println(contador); //mostrará 0
```

Cuando el operador va detrás de la variable, primero se evalúa la expresión y después se aplica el incremento. Por ejemplo:

```
int contador = 0;
System.out.println(contador++); //mostrará 0
```

Se mostrará 0 como consecuencia de evaluar la expresión y justo a continuación, antes de ejecutar la línea siguiente, la variable valdrá 1. En cambio, si el incremento antecede a la variable, primero se aplica el incremento y a continuación se evalúa la expresión:

```
int contador = 0;
System.out.println(++contador); //mostrará 1
```

De igual manera funciona para el decremento.

9 Constantes

Al definir variables, existe la posibilidad de asignar un identificador que le permite dar una cierta semántica al valor que almacena. Si elige bien este identificador, puede ser mucho más sencillo entender qué hace el programa cuando lea el código fuente. Desgraciadamente, al usar literales se pierde un poco esta expresividad, ya que sólo se dispone del valor tal cual, pero no se puede saber exactamente la función sin inspeccionar el código con atención. Para solucionar este detalle, los lenguajes de programación permiten reemplazar el uso de literales por constantes.

Una **constante** es un tipo especial de variable que tiene la particularidad de que dentro del código del programa su valor sólo puede ser leído, pero nunca modificado.

El uso de este término es similar al que se hace en matemáticas o física, donde se usa para indicar ciertos valores universales e inmutables: π (3,1415 ...), c (300.000 m/s, la velocidad de la luz), etc.

Hay varias maneras de definir constantes en el Java, veamos la más popular: las constantes se definen fuera del método principal, pero dentro del bloque que identifica un código fuente Java (clase). La sintaxis es idéntica a la definición de la variable, pero antes de la palabra clave para el tipo de dato hay que añadir las palabras reservadas *private static final*, en este orden. Es decir:

```
private static final tipo NOMBRE_CONSTANTE = valor;
```

Al igual que con las variables, el valor inicial puede ser tanto un literal como una expresión. Ahora bien, esta expresión no puede contener variables.

Convenciones de nomenclatura

Para el caso de las constantes, las convenciones de código existentes en Java son diferentes. En este siempre se usan mayúsculas para todas las letras y, en caso de llamarlas con una composición de diferentes palabras, estas se separan con un símbolo de subrayado o guion bajo (*underscore*, _).

Ejemplos de identificadores de constantes que siguen la convención son: CONSTANTE, PI, VELOCIDAD_LUZ, ACELERACION_GRAVEDAD, etc.

Podemos comparar la legibilidad del código de los dos programas. En el primero parece que se aplique una multiplicación arbitraria al final de código, pero en el segundo queda bien claro por qué hay que hacerla.

```
//Sin usar constantes
public class IVA {

	public static void main(String[] args) {
		double precio = 15.50;
		double pvp = precio + (precio * .21);
		System.out.println("Total: " + precio);
		System.out.println("Total IVA incluido: " + pvp);
	}
}
```

```
//Utilizando constantes
public class IVA {
	private static final double IVA = .21;

	public static void main(String[] args) {
		double precio = 15.50;
		double pvp = precio + (precio * IVA);
		System.out.println("Total: " + precio);
		System.out.println("Total IVA incluido: " + pvp);
	}
}
```

Las constantes son también especialmente útiles cuando, aparte de querer etiquetar ciertos literales de manera que el código sea más fácil de entender, el literal en cuestión aparece varias veces en el código fuente. Supongamos que en una versión posterior del programa hay que hacer un cambio en el valor de este literal. Por ejemplo, el IVA a aplicar dentro de un programa de tienda virtual pasa de 0,21% a 0,18%. Si no se ha declarado como constante, habrá que buscar este valor línea por línea y asegurarse de que se ha usado para hacer un cálculo de IVA (tal vez se ha usado este mismo literal, pero con otra finalidad). Si, en cambio, hemos declarado una constante, basta con modificar sólo la declaración de la constante. El cambio se produce automáticamente en todas las expresiones en las que se usa la constante con este identificador.

La declaración de constantes en lugar de usar directamente un literal cuando éste aparece en muchos lugares dentro del código también ofrece otra ventaja sutil. Supongamos que nos equivocamos al escribir alguna de las veces que aparece el literal. En lugar de 0,21% de IVA escribimos 0,31% (ya que las teclas '2' y '3' están juntas). En este caso, el programa compilará perfectamente, pero no hará lo que tiene que hacer correctamente. Lo peor es que, de entrada, no tendremos ni la menor idea de por qué ocurre esto y tendremos que repasar todo el código hasta encontrar el literal incorrecto. En cambio, con una constante, el literal del escriba una única vez y ya está. En cambio, si nos equivocamos al escribir el identificador de la constante, el compilador considerará que está haciendo uso de un identificador no declarado previamente y nos dará un error, indicando exactamente el lugar dentro del código fuente donde nos hemos equivocado. Arreglarlo será muy fácil.

Una pregunta que nos podemos hacer es si no se puede lograr exactamente lo mismo simplemente usando una variable, y no modificar más el valor dentro del programa. Es una buena pregunta, y la respuesta es que al efecto funcional sería el mismo. Ahora bien, definir constantes tiene dos ventajas, pero para entenderlo hay que tener un poco de idea de cómo funciona un compilador. La primera ventaja es que el compilador controla explícitamente que ninguna constante declarada es modificada en alguna instrucción del código fuente programa. Si esto sucede, considera que el código fuente es erróneo y genera un error de compilación. Esto puede ser muy útil para detectar errores y garantizar que sus constantes realmente nunca ven modificado su valor a lo largo de la ejecución del programa a causa de alguna distracción. El segundo es que, en algunos lenguajes, antes de procesar el código fuente, el compilador reemplaza todas las apariciones del identificador de cada constante por el literal que se le ha asignado como valor. Por tanto, al contrario que las variables, las constantes no ocupan espacio realmente a la memoria mientras el programa se ejecuta.

Por lo tanto, es una buena costumbre ceñirse al uso de constantes cuando se está tratando con datos generales que nunca cambian de valor.

10 Conversiones de tipo

Hasta el momento hemos dicho que siempre que hay una asignación de un valor, ya sea en establecer el valor de una variable o el de una constante, los tipos debe ser exactamente igual en ambos lados de la asignación. Bueno, en realidad esto no es estrictamente cierto. Bajo ciertas condiciones es posible hacer asignaciones entre tipos de datos diferentes si sobre el valor en cuestión se puede hacer una conversión de tipo.

Este proceso también se conoce como *casting*.

Una **conversión de tipo** es la transformación de un tipo de dato en otro diferente.

Dentro de los lenguajes de programación hay dos tipos diferentes de conversiones de tipos: las implícitas y explícitas. El primer caso corresponde a aquellas conversiones fáciles de resolver y que el lenguaje de programación es capaz de gestionar automáticamente sin problemas. El segundo caso es más complejo y lo fuerza el programador, si es posible.

10.1 Conversión implícita

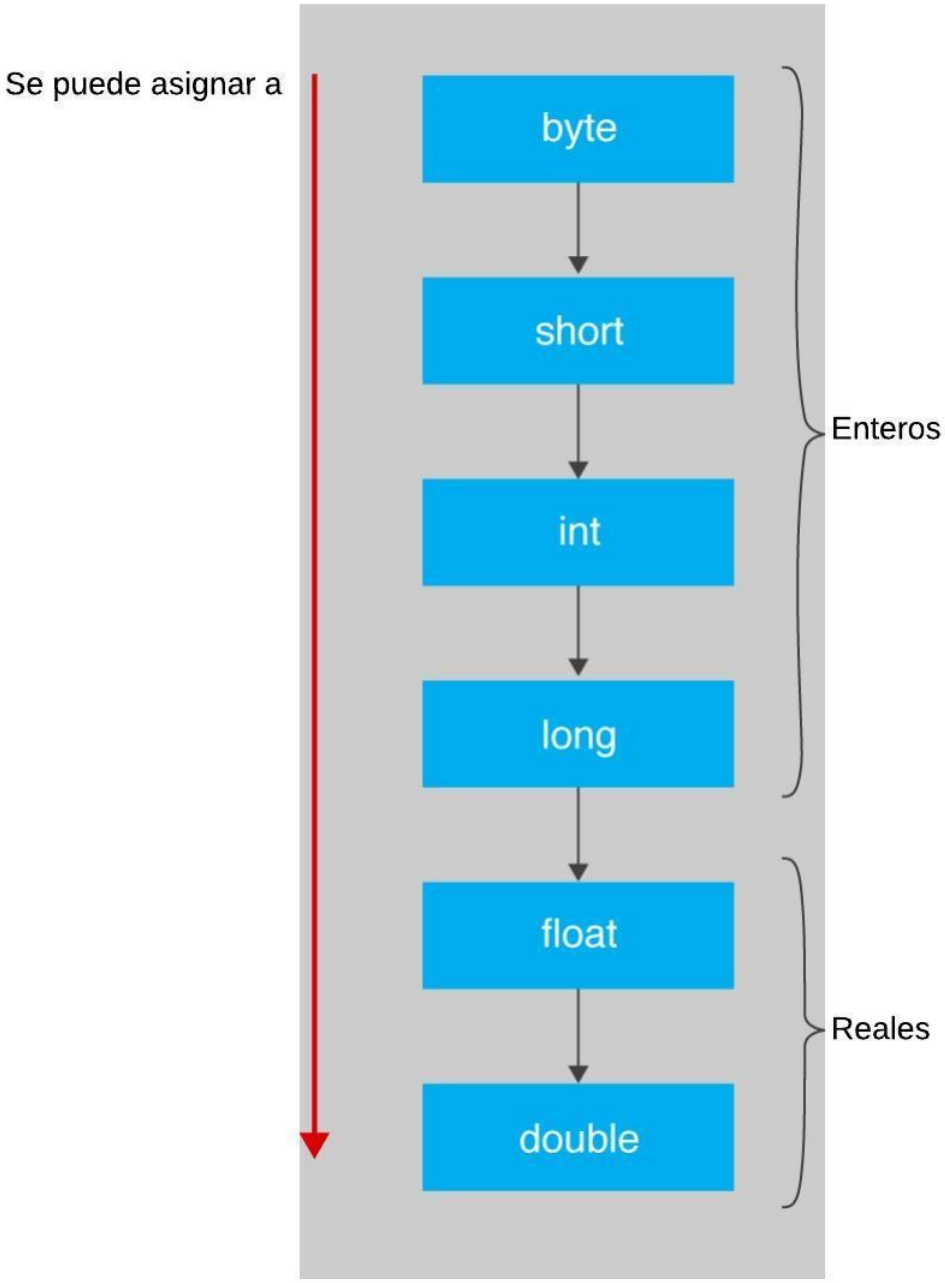
Este caso es el más simple y se fundamenta en el hecho de que hay ciertas compatibilidades entre tipos primitivos de datos diferentes. Lo vemos ilustrado con el siguiente programa:

```
public class ConversionImplicita {

      public static void main(String[] args) {

            // El literal "100" es un entero y "real" es un float.
Son tipos diferentes, no?
          float real =  100 ;
          // Pero, "double" y "real" son dos variables de tipo
diferente, no?
          double doble = real ;
          System.out.println (doble) ;

      }

}
```

La sintaxis de este programa no se corresponde a lo que se ha explicado hasta ahora. En cambio, si intentamos compilarlo y ejecutarlo, funcionará correctamente. Esto es porque en este código están los dos casos de conversiones implícitas más habituales.

El primer caso se corresponde a que todos los valores que pertenecen al tipo de dato entero se pueden traducir muy fácilmente a real. Simplemente hay que considerar que sus decimales son .0 y ya está. Esto es lo que pasa aquí. El literal 100 es un entero (concretamente, un int), pero se asigna sobre una variable de tipo float, por lo que en principio sería incorrecto. Habría que usar el literal que expresa este valor como un real de precisión simple, el 100.0F. Sin embargo, el compilador de Java considera que la asignación es correcta, ya que es capaz de hacer automáticamente esta conversión desde el tipo entero a real.

Relaciones de compatibilidad entre tipos numéricos



El segundo caso está vinculado al rango del tipo de dato numérico. Si un valor es de un tipo primitivo numérico concreto, este puede ser interpretado fácilmente como un dato de los tipos que acepta un rango mayor de valores. Por ejemplo, cualquier valor de tipo int que os podáis imaginar seguro que también está dentro de los valores aceptados entre los long. Igual pasa entre los float y los double. El compilador de Java también se da cuenta de ello y por ello acepta una asignación de un valor float (menos rango) a un double (más rango) sin problemas, ya que sabe hacer la conversión.

10.2 Conversión explícita

En cualquier caso, donde haya asignaciones entre datos de tipo diferente pero el lenguaje elegido sea incapaz de hacer una conversión implícita, el compilador siempre dará error. Ya de por sí, la aparición de este error suele significar que algo no está bien y que hay que modificar

el programa. Pero hay casos en que es permisible eliminar este error especificando el código que se quiere forzar la conversión de los datos a un nuevo tipo de todas formas. Básicamente, lo que hace es obligar al compilador a transformar un dato de un tipo en una diferente "lo mejor posible". Para indicarle ello, ante la expresión para convertir se pone el tipo de dato resultante que se quiere obtener entre paréntesis. La sintaxis es la siguiente:

```
identificador = (tipo) expresión;
```

Al hacer esto, el resultado de la expresión se intentará convertir al tipo de dato especificado, sea cual sea el tipo en que evalúe originalmente. El valor final obtenido dependerá mucho de la situación bajo la cual se hace la conversión, ya que es evidente que convertir datos a tipos totalmente diferentes de su origen no es una acción directa. En cualquier caso, el tipo elegido para la conversión debe ser compatible con el de la variable en la que se asigna, como siempre.

La conversión explícita se debe usar con mucho cuidado y teniendo muy claro qué se está haciendo.

La situación más frecuente en que se usa la conversión explícita es invertir el orden establecido en las relaciones de la figura del apartado anterior, ya que normalmente un lenguaje de programación no permite hacer asignaciones en dirección opuesta a las flechas. Esto se debe a que, por ejemplo, un real o un valor de tipo long muy grande no se pueden traducir normalmente como un entero (int). En el primer caso no hay manera de representar los decimales y en el segundo se sale fuera del rango admisible. Ahora bien, sí hay casos en que esta asignación tiene sentido: cuando los decimales del real son exactamente 0 o cuando el valor del long está dentro del rango aceptable para los enteros. Entonces, usando la conversión explícita podemos evitar que el compilador dé un error y hacer que intente convertir de manera correcta los valores en enteros. Si podemos garantizar que el valor original cumple las condiciones para que la traducción sea correcta, la conversión se completará con éxito. En caso contrario, el valor final asignado será seguro erróneo.

A continuación, se muestra un ejemplo de conversiones explícitas entre tipos que se supone que no son compatibles. Ahora bien, como el programador puede garantizar que el valor real para convertir, 300.0 es representable en diferentes tipos de entero, todo funciona correctamente. Al final se ve por pantalla el valor 300 (o sea, un literal de tipo entero). Pero si eliminamos las conversiones explícitas, (long) y (int), el programa no compilará.

```
public class ConversionExplicita {

	public static void main(String[] args) {
		// Declaración de una variable real
		double dValor = 300.0;
		// Asignación incorrecta. Un real tiene decimales, no?
		long lValor = (long) dValor;
		// Asignación incorrecta. Un entero largo tiene un rango mayor
		// que un entero, no?
		int iValor = (int) lValor;
		System.out.println(iValor);

	}

}
```

Para que este mecanismo funcione es muy importante que estemos seguros de que el valor para convertir realmente pertenece al tipo de dato de destino. En caso contrario, el resultado final no se adaptará fielmente al dato original. Por ejemplo, en el caso de la conversión de cualquier tipo real en entero se perderán todos los decimales (se truncará el número), y en el caso de asignar un valor fuera del rango posible entre cualquier tipo de dato obtendrá un desbordamiento.

Puedes comprobar que ocurre en el programa anterior con un valor de 3000000000.

10.3 Conversión con tipos no numéricos

En ciertos lenguajes de programación también se puede intentar hacer conversiones de tipo en que alguna o ambos datos son de tipo no numérico. Entonces lo que se pide es hacer una operación a bajo nivel con la representación interna, en binario, del dato. Para entender qué sucede en esta situación, hay que recordar que la combinación de unos y ceros como se representa un dato concreto depende de su tipo. Ahora bien, también hay que tener presente que una misma secuencia binaria puede representar valores diferentes para tipos de datos diferentes. Por ejemplo, el valor de tipo carácter ‘a' se codifica internamente con la secuencia binaria 0000000001100001, pero esta secuencia también es usada para codificar el valor de tipo entero '97'. El tipo de dato proporciona un contexto sobre el significado de los datos en binario cuando se lee un dato dentro del ordenador.

Hacer una conversión con datos de tipo no numérico indica al ordenador que tiene que coger el valor binario encontrado en la memoria tal cual e interpretarlo como si fuera otro tipo de dato totalmente diferente.

En la mayoría de casos, estas se deben hacer explícitamente, pero en situaciones muy concretas el lenguaje incluso es capaz de interpretarlas implícitamente. A modo de ejemplo, Java es capaz de llevar a cabo la conversión de carácter a entero, y viceversa, de manera implícita. En otros lenguajes se debe hacer explícita o simplemente no se puede hacer.

```
public class ConversionCaracteres {

	public static void main(String[] args) {
		int entero = 'a';
		char caracter = 98;
		// Escribe el valor en binario del literal 'a'
interpretado como un entero: 97.
		System.out.println(entero);
		// Escribe el valor en binario del literal 98
interpretado como un carácter:
		// 'b'.
		System.out.println(caracter);

	}

}
```

En otros lenguajes, como C, se puede intentar con prácticamente cualquier tipo de dato.

Hay que decir que el uso de este caso es muy marginal. De hecho, aunque esta última opción de operar a bajo nivel, en cada lenguaje hay conversiones explícitas que simplemente son imposibles. Si las quiere hacer el compilador dirá que hay un error igualmente. En Java, por ejemplo, es el caso de convertir un dato del tipo booleano a cualquier otro.

11 Visualización de los datos en Java

La verdad es que no tiene ningún sentido que un programa lleve a cabo el procesamiento de unos datos si esta tarea luego no tiene ninguna repercusión en el sistema de entrada / salida. Sobre todo, teniendo en cuenta que una vez el programa finaliza, todos los valores tratados borran de la memoria. Lo menos que puede hacer es mostrar el resultado de la tarea por pantalla, de manera que el usuario que ha ejecuta el programa sepa cuál ha sido el resultado de todo ello. Otras acciones, más complejas ya, serían guardar los datos en un fichero de manera persistente, en una base de datos o imprimirlos a través de una impresora.

Hay muchas maneras de entrar o mostrar los datos al usuario, basta con ver las interfaces gráficas de los sistemas modernos, pero aquí verá la más simple: cadenas de texto en la consola o por teclado.

11.1 Instrucciones de salida de datos por consola

Aunque hasta el momento ya se ha visto en los ejemplos de apartados anteriores algunas instrucciones que sirven para mostrar cosas en la consola, ha llegado el momento de inspeccionarlas con algo más de detenimiento. Antes de empezar, sin embargo, vale la pena decir que cada lenguaje de programación tiene sus propias instrucciones para visualizar datos por consola. Si bien la idea general puede tener ciertas similitudes, la sintaxis puede ser totalmente diferente.

Principalmente, Java proporciona dos instrucciones para mostrar datos en la consola:

```
// Muestra en la consola el resultado de evaluar la expresión x
System.out.print(x);
// Hace lo mismo, pero al final añade un salto de línea.
System.out.println(x);
```

El texto se muestra en el orden según se vayan ejecutando las instrucciones de salida por consola. No se puede pedir un posicionamiento absoluto a cualquier lugar de la consola. Por lo tanto, cada instrucción comenzará a escribir inmediatamente después de donde la ha dejado la instrucción anterior.

Uno de los aspectos más importantes de representar datos por consola es que este periférico sólo trata con texto. O sea, conjuntos de caracteres en Java (char). Por tanto, en el proceso de salida a la consola todo dato que se quiere mostrar debe sufrir una transformación desde la representación original hasta el conjunto de caracteres que la representan en formato texto.

Supongamos que tenemos una variable de tipo entero en Java (int), en el que hay almacenado el valor 1309. Esto internamente se representa con el valor binario: 0000010100010100. Ahora bien, por la consola lo que se debe mostrar son los 4 caracteres (char) '1', '3', '0', '9'. Es decir, se ordenará al sistema de entrada / salida por consola que muestre los cuatro valores de la representación en binario:

- '1': 00000000000000000110001
- '3': 00000000000000000110011
- '0': 00000000000000000110000
- '9': 00000000000000000111001

Otros lenguajes ofrecen otros mecanismos, algunos no tan simples, para hacer esta transformación.

Por lo tanto, si se quisiéramos escribir un entero, sería necesario un paso de transformación previo a representarlo en diferentes caracteres individuales. Con respecto a este tipo de instrucciones, una de las grandes ventajas de Java, como lenguaje de nivel alto en relación con otros lenguajes de nivel más bajo, es que las dos instrucciones básicas de que dispone pueden aceptar cualquier expresión que al ser evaluada resulte en cualquier tipo primitivo, sea cual sea (entero, real, booleano o carácter). Automáticamente se encargan ellas mismas de transformar el valor almacenado en su representación en formato texto, y la muestran por consola.

A modo de ejemplo, veamos el siguiente programa:

```
// Muestra diferentes expresiones simples por pantalla.
public  class MostrarExpresiones {
  public  static  void main(String[ ] args){
    // Muestra el texto "3" por pantalla.
    System.out.println(3) ;
    // Muestra el texto "a" por pantalla.
    System.out.println('a') ;
    // Muestra el texto "3.1416" por pantalla.
    double x =  3.1416 ;
    System.out.println(x) ;
    // Muestra el texto "true" por pantalla.
    boolean a = true ;
    boolean b = false ;
    System.out.println((a || b) && (true)) ;
  }
}
```

Al ejecutarlo se muestra por consola lo siguiente:

```
3
a
3.1416
true
```

11.2 Cadenas de caracteres

Ya hemos visto que una **cadena de texto** (*String*) es un tipo de dato que permite representar cualquier secuencia de caracteres de longitud arbitraria.

Como tipo de dato, también se pueden declarar variables usando la palabra clave, al igual que con cualquier otro tipo primitivo:

```
String holaMundo = "¡Hola, mundo!";
```

11.2.1 Operadores de las cadenas de texto

En contraposición de los tipos primitivos, no se puede generalizar al hablar de operaciones de cadenas de texto. Cada lenguaje puede ofrecer diferentes (o ninguno). En el caso del Java, las cadenas de texto aceptan el operador suma (**+**), por lo que se pueden generar expresiones de la misma manera que se puede hacer con los tipos primitivos. En este caso, **el resultado de aplicarlo en una operación entre cadenas de texto es que éstas se concatenan**. Se genera una nueva cadena de texto formada por la unión de cada uno de los operandos de manera consecutiva.

```
// Muestra el texto "¡Hola, mundo!" por consola usando concatenación
de cadenas de texto.
public  class HolaMundoConcatenado{
  public  static  void main(String[ ] args){
    String hola = "Hola," ;
    String mundo = "mundo" ;
    String exclamación = "!" ;
    // Muestra el texto "Hola, mundo!" por consola
    System.out.println(hola + mundo + exclamación);
  }
}
```

Las cadenas de texto se pegan directamente una tras otra, sin añadir ningún espacio.

De hecho, para facilitar aún más el trabajo del programador, Java va un poco más allá y permite aplicar este operador entre una cadena de texto y cualquier tipo primitivo. El resultado siempre es una nueva cadena de texto en el que el tipo primitivo se transforma en su representación textual. Por ejemplo:

```
System.out.println(“Muestra el ” + 3);
```

11.2.2 Secuencias de escape

En algunos lenguajes de programación, como en Java, hay un conjunto de literales dentro de los caracteres que son especialmente útiles dentro de las cadenas de texto, ya que permiten representar símbolos o acciones con un significado especial. Estos se denominan *secuencias de escape* y siempre se componen de una barra inversa (****) y un carácter adicional. La siguiente tabla muestra una lista de los más habituales.

Tabla Secuencias de escape típicas

Secuencia de escape	**Acción o símbolo representado**
\ t	Una tabulación.
\ n	Un salto de línea y retorno de carro.
\'	El carácter "comilla simple" (').
\"	El carácter "comillas dobles" (").
\\	El carácter "contrabarra" (\).

Como se puede apreciar, para los dos primeros casos hay que usar una secuencia especial, ya que son acciones que encontramos en un teclado, pero no son representables gráficamente. Por ejemplo, el código:

```
System.out.println("línea 1\n\tlínea2\nlínea 3");
```

mostraría por consola:

```
línea 1
      línea 2
línea 3
```

En el resto de casos, su existencia se debe a la forma en que se representan los literales de caracteres o cadenas de texto. Como el literal de un carácter se rodea con dos comillas simples y el de una cadena de texto con comillas dobles, es la única manera de distinguir en el texto si este símbolo se usa para delimitar un literal o como texto que forma parte. La contrabarra se representa de este modo para distinguirla respecto de cuando se usa, precisamente, para especificar una secuencia de escape.