Tema 2. Introducción al lenguaje Java

Contenido

Objetivos

- Reconocer la importancia de Java
- Ver la importancia de utilizar identificadores significativos
- Distinguir entre tipos predefinidos y de usuario
- Reconocer la diferencia entre los tipos carácter y cadena y cómo se relacionan
- Reconocer la diferencia entre constante y variable
- Ser capaz de describir un objeto de forma abstracta
- Conocer por qué hay diferentes tipos de datos numéricos con distintos rangos de valores
- Comprender la diferencia entre los valores enteros, y coma flotante
- Ver cómo las reglas de precedencia afectan al orden de evaluación de una expresión
- Comprender la conversión de tipo implícita y explícita

1 Introducción al lenguaje Java

1.1 ¿Qué es Java?

Java es un lenguaje de programación, como otros tantos que existen en el mercado. Fue desarrollado por **Sun Microsystems**, que es una empresa conocida principalmente por sus servidores y estaciones de trabajo.

Java se basa en el lenguaje C++, aunque existen diferencias en algunos aspectos básicos que lo convierten en una evolución más cercana a una representación del mundo real que su antecesor.

Entre finales del año 2006 y principios del 2007, Sun liberó el código de Java bajo licencia GPL, convirtiéndose así en un lenguaje de programación libre.

1.2 ¿Por qué estudiar Java?

Java tiene algunas ventajas sobre otros lenguajes de programación que parecen haber convencido a una gran mayoría de desarrolladores y empresas a utilizarlo. Un buen ejemplo de ello lo tenemos en las ofertas de empleo que se ofrecen a diario y que muestran la importancia que este lenguaje ha adquirido en un tiempo relativamente corto (la primera versión de Java apareció en 1991).

También existen estudios (como la lista de TIOBE) donde se calcula la popularidad de una gran cantidad de lenguajes de programación. En 2003 Java ocupaba la tercera posición de popularidad de la lista y, desde principios del año 2006, se mantuvo en primera posición (¿cómo se situará ahora?).

A continuación explicaremos cuáles son esas ventajas que tanta popularidad le han proporcionado.

1.3 Java es fácil de aprender

Si comparamos Java con C, veremos algunas características que corroboran este hecho.

Seguro que un alto porcentaje de los estudiantes a los que se les pregunte qué parte ha sido la más complicada y ardua de estudiar en C contestarán que han sido los punteros.

Pues si es así, a ninguna de estas personas les será indiferente el siguiente hecho: en Java no existen los punteros y la gestión de memoria dinámica se realiza automáticamente. Pero además de esto, una vez conocidas las bases del lenguaje, comprobaréis que el uso del lenguaje Java es más natural que el de C.

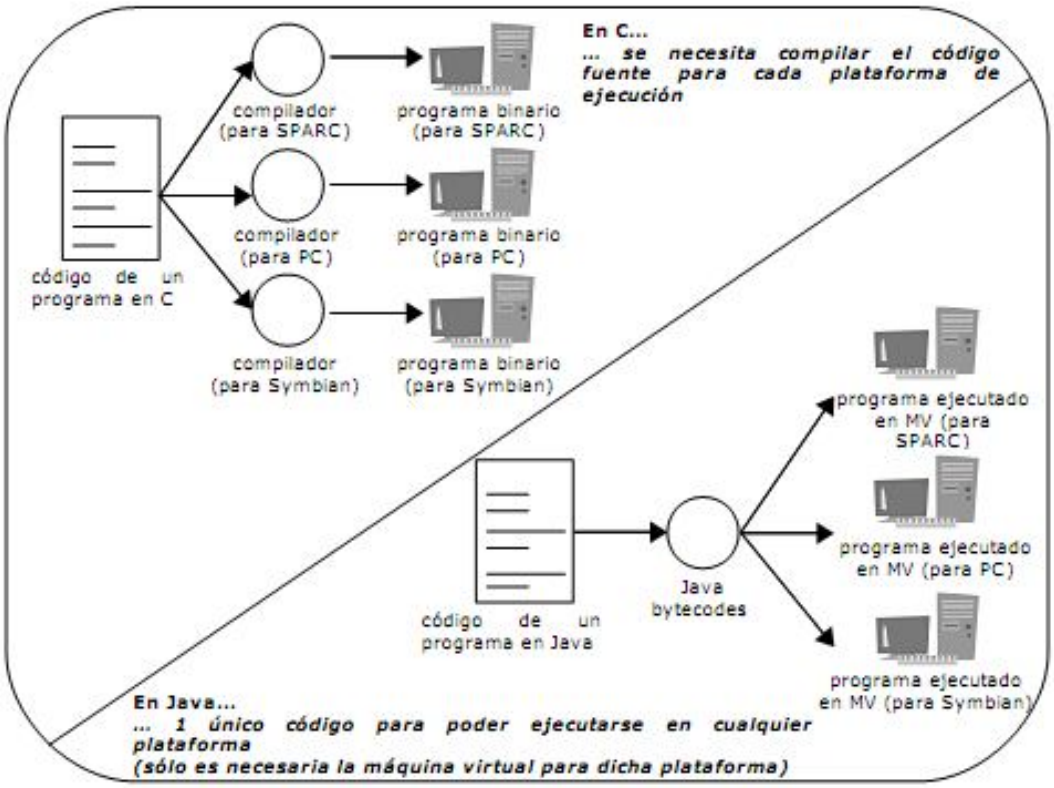
Java es un lenguaje de plataforma independiente.

Esto quiere decir que un programa en Java puede ejecutarse en cualquier dispositivo: desde una estación de trabajo de última generación hasta un teléfono móvil.

A diferencia de otros lenguajes, el código fuente del programa no se debe compilar de forma diferente según se utilice en uno u otro hardware. A esto se le llama independencia a nivel de código fuente.

Además, al compilar el programa no se genera código binario dependiente del hardware, sino que se crean los llamados bytecodes de Java, que son piezas de código intermedio, es decir, a medio camino entre el código fuente y el código binario de la máquina. A esto se le llama independencia a nivel de código binario.

Claro está, para poder ejecutar este código intermedio, es necesario algún programa que lo "reconozca". A este programa se le llama la máquina virtual de Java, que deberá adaptarse al hardware sobre el que se va a ejecutar nuestro programa. Por tanto, para ejecutar un programa escrito en Java en cualquier dispositivo, será necesario previamente tener instalada la máquina virtual en ese dispositivo.



1.4 Software necesario

Eclipse. Utilizaremos este IDE que se encuentra publicado bajo la licencia libre EPL (Eclipse Public License).

Máquina virtual Java. Paradójicamente, Eclipse es un programa desarrollado en Java. Por tanto, para poder ejecutarlo, es necesario tener instalada la máquina virtual en el ordenador. A la máquina virtual de Java también se la conoce como Java VM, JVM o VM.

La plataforma Java SE está formada principalmente por dos productos: el JDK, que contiene los compiladores y depuradores necesarios para programar, y el JRE, que proporciona las librerías o bibliotecas y la JVM, entre otra serie de componentes.

2 Concepto de Dato

Un **dato** puede ser un carácter leído desde teclado, un número, la información almacenada en un CD, una foto almacenada en un fichero, una canción, el nombre de un alumno almacenado en la memoria del ordenador, etc.

La información en el ordenador no se almacena ni representa al azar, sino que ha de **organizarse y estructurarse de forma adecuada**.

3 Tipos de datos

En este punto analizaremos:

- **Los tipos básicos de datos que incluyen la mayoría de los lenguajes de programación**, (aunque con diferencias significativas entre unos y otros).
- **Las estructuras de datos que pueden crearse a partir de ellos**.
  Como veremos, las estructuras se forman a partir de los tipos elementales, pero también a partir de otras estructuras.
- **La utilidad de estas estructuras**.

Cuando desarrollemos una aplicación tendremos datos que no cambiarán nunca, como el **CIF** de la empresa y otros que cambiarán de valor con muchísima frecuencia, como la hora expresada con precisión de minutos que debe aparecer en una esquina de la ventana de nuestra aplicación.

Lo mismo ocurre en todos los programas. Usaremos datos asociados normalmente a **constantes** y a **variables**.

3.1 Variables

Tanto constantes como variables son **zonas de la memoria** del ordenador a las que se asigna un nombre (**identificador**), a modo de *etiqueta*, y a las que se les asocia un **tipo de dato**, de forma que sólo se podrán almacenar en esa zona de memoria (o posición de memoria) datos de ese tipo.

El identificador de una variable o de una constante sólo puede contener *letras*, *números*, el carácter de *subrayado* y el símbolo *$*. El nombre puede contener cualquier carácter Unicode. La única restricción a la hora de utilizar un identificador es que **no puede empezar por un dígito**.

La diferencia entre constante y variable es:

- **Las constantes reciben valor una vez**, normalmente al principio del programa, **y ya no puede volver a escribirse nada en la posición de memoria que ocupan** (una vez que se les asigna el valor, no se puede modificar, permanece constante). Al declararlas, el nombre de la constante debe ir precedida por la palabra reservada final. Puede declararse sin valor inicial, pero desde el momento que se le asigne un valor, la constante no podrá ser modificada. Ejemplo:

  ```
  final float precioIva;
  precio = 1000;
  precioIva = precio*0.21; // A partir de este momento, precioIva no puede cambiar su valor
  ```

- **Las variables pueden cambiar de valor tantas veces como se desee a lo largo del programa** (podemos volver a escribir otro valor en la zona de memoria que ocupan tantas veces como deseemos). Las variables se declaran indicando el tipo y, a continuación el nombre de la variable. Por ejemplo:

  ```
  int dias;
  ```

En definitiva, **para usar los datos, tenemos que disponer de ellos en la memoria del ordenador**. Es como si la memoria fuese un enorme casillero, en el que cada casilla puede almacenar un dato, y tiene una dirección asignada (realmente un **número binario**) y puedo referirme a una casilla en concreto y al valor que contiene a través de su dirección.

La variable (o constante), además del nombre lleva asociado un tipo de dato. Un tipo de dato no es más que una especificación de los valores que son válidos para esa variable y de las operaciones que se pueden realizar con ellos.

Normas de estilo para nombrar variables

A la hora de nombrar un identificador existen una serie de normas de estilo de uso generalizado que, no siendo obligatorias, se usan en la mayor parte del código Java. Estas reglas para la nomenclatura de variables son las siguientes:

- **Java distingue las mayúsculas de las minúsculas**. Por ejemplo, `Alumno` y `alumno` son variables diferentes.
- **No se suelen utilizar identificadores que comiencen con «$» o «_».** Además el símbolo del dólar, por convenio, no se utiliza nunca.
- **No se puede utilizar palabras reservadas del lenguaje**.

Los **identificadores deben ser lo más descriptivos posibles**. Es mejor usar palabras completas en vez de abreviaturas crípticas. Así nuestro código será más fácil de leer y comprender. En muchos casos también hará que nuestro código se autodocumente. Por ejemplo, si tenemos que darle el nombre a una variable que almacena los datos de un alumno sería recomendable que la misma se llamara algo así como registroAlumno, y no algo poco descriptivo como al33.

Además de lo dicho hasta ahora, en la siguiente tabla hay otras convenciones, que no siendo obligatorias, sí son recomendables a la hora de crear identificadores en Java.

Identificador	Convención	Ejemplo
Nombre de variable	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas	numAlumnos, suma
Nombre de constante	En letras mayúsculas, separando las palabras con el guión bajo. Por convenio el guión bajo no se utiliza en ningún otro sitio	TAM_MAX, PI
Nombre de una clase	Comienza por letra mayúscula	String, MiTipo
Nombre de función	Comienza por letra minúscula	modificaValor, obtieneValor

3.2 Preguntas que deben definir un tipo de datos

Un tipo de datos contesta a dos preguntas:

- **¿Qué valores o datos son válidos?**
- **¿Qué operaciones puedo hacer con esos valores o datos?**

Por ejemplo, el tipo entero permitirá unos valores y unas operaciones distintas a las del tipo real.

A continuación veremos algunos de los **tipos** existentes en la mayoría de los lenguajes de programación aunque NO todos los lenguajes disponen de los mismos tipos de datos.

3.3 Datos simples o básicos o primitivos

Realmente deberíamos hablar más de tipos simples de datos que de datos simples. Usamos la segunda nomenclatura por simplicidad.

¿Qué son los datos simples o primitivos? **Son los datos que suele proporcionar el lenguaje de programación, y que por eso de ellos conocemos:**

- **Cómo se representan internamente (en memoria).**

- **El tamaño exacto que ocupan.**
- **Los operadores que define el lenguaje para ellos.**
- **Que están disponibles sin necesidad de definir nada por parte del usuario.**

Son un conjunto mínimo de tipos de datos, de forma que a partir de ellos se construirán los demás tipos de datos, que por ello recibirán el nombre de datos estructurados.

3.3.1 Cuadro de tipos de datos primitivos en Java

Tipo	Descripción
boolean	Permite representar valores lógicos; Verdadero (**V**) o Falso (**F**).
char	Permite representar un símbolo o carácter UNICODE de 16 bits.
byte	Entero de **8 bits** con signo (representado en complemento a dos). Su rango de valores va desde **-128 (-2^7)** a **+127 (+2^7-1)**.
short	Entero de **16 bits** con signo (complemento a dos). Rango de valores entre **-32.768 (-2^{15})** y **+32.767 (+2^{15}-1)**.
int	Entero de **32 bits** con signo (complemento a dos). Rango de valores entre **-2.147.483.648 (-2^{31})** y **+2.147.483.647 (+2^{31}-1)**.
long	Entero de **64 bits** con signo (complemento a dos). Rango de valores entre **-2^{63}**y **+2^{63}-1**.
float	Número real (en coma flotante) de 32 bits, utilizando la representación IEEE 754-1985.
double	Número real (en coma flotante) de 64 bits, utilizando la representación IEEE 754-1985.

3.3.2 Numéricos

Sería impensable que los ordenadores, que queremos usar para resolver todo tipo de problemas, y además de forma automática no usaran números y operaciones matemáticas. Por eso es imprescindible que cualquier lenguaje defina como tipos básicos algunos tipos de datos numéricos.

Los tipos **numéricos** son tipos básicos que nos permiten representar valores de tipo numérico. La mayoría de los lenguajes sólo permiten representar y usar dos grupos de datos numéricos, para **números enteros** y para **números reales**. Cualquier otro tipo numérico como por ejemplo los números Racionales o los números Complejos, deberán ser definidos por el usuario como una estructura a partir de los tipos simples entero y real.

3.3.3 Enteros. Tipos de enteros

Los datos de tipo entero permiten representar números enteros (sin decimales) tanto positivos como negativos. La mayoría de los lenguajes definen varios tipos de enteros, que se diferencian fundamentalmente en el número de **bytes** que se usan para su representación. Cuantos más bytes usemos para representar un número entero, mayor será el rango de números representable.

Por ejemplo, en Java, existen cuatro tipos de números enteros:

Nombre del tipo entero	Bits usados para su representación	Total de números distintos representables	Menor número representable para el tipo	Mayor número representable para el tipo
byte	8 =1 byte	2^8 = 256	-128	+127
short	16 =2 bytes	2^{16} = 65.536	-32.768	+32.767
int	32 = 4 bytes	2^{32} = 4.294.967.296	-2.147.483.648	+2.147.483.647
long	64 = 8 bytes	2^{64} = 1.844.674.407 E+19	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

Otros lenguajes definirán otros datos de tipo entero, pero es usual que al menos todos los lenguajes dispongan de un tipo "entero" y de un tipo "entero largo". Las diferencias entre unos tipos enteros y otros, serán siempre el rango de valores representables.

En la actualidad, la diferencia entre definir todas las variables de un programa por ejemplo de tipo short o definirlas de tipo int puede ser una cantidad de memoria insignificante e irrelevante para el rendimiento de la aplicación. Por eso es usual definir las variables y constantes enteras directamente como int, aunque sepamos que su valor nunca va a ser mayor de 10, por ejemplo. Sin embargo sólo definiremos las variables o constantes como long cuando sepamos con seguridad que vamos a manejar datos o valores realmente grandes, ya que 8 bytes (64 bits) empiezan a ser un tamaño no despreciable.

3.3.4 Reales. Tipos de reales

Los datos de tipo real se usan para representar números reales (con decimales). Al igual que ocurre con los enteros, la mayoría de los lenguajes definen más de un tipo de datos real, que se diferencian igualmente entre sí por el número de bits que se usan para representarlos y almacenarlos.

Cuantos más bits usemos, podremos representar números:

- **Más grandes en valor absoluto** (tanto positivos como negativos)
- **Con mayor precisión** (con más decimales exactos)

La mayoría de los lenguajes definen también como tipos básicos (o simples o primitivos), más de un tipo de números reales, que se diferencian en el número de bits usados para la representación, y por tanto en la precisión de los números representados.

Por ejemplo, en Java se definen como de tipo real los siguientes:

Nombre del tipo real	bits usados para su representación	Total de números distintos representables	Menor número representable para ese tipo (en valor absoluto)	Mayor número representable para ese tipo (en valor absoluto)
float	32 = 4 bytes	2^{32} = 4.294.967.296	1.401298464324817 E-45	3.4028234663852886 E+38
double	64 = 8 bytes	2^{64} = 1.844.674.407 E+19	4.9 E-324	1.7976931348623157 E+308

En el caso de los números reales, cuando no tengamos limitaciones serias de la memoria disponible para ejecutar nuestra aplicación, **se recomienda usar siempre el tipo de mayor precisión**, ya que así, al operar, **los errores cometidos por las sucesivas aproximaciones serán menores**. De hecho, en Java los literales reales se consideran double por defecto, y se recomienda que todas las variables de tipo real se definan como double. La diferencia entre usar 8 bytes para cada variable double en vez de los 4 bytes de float en un programa que use 10.000 variables de tipo real (lo cual es muchísimo) supone usar 80.000 bytes en vez de 40.000, lo que supone una diferencia de menos de 40 Kbytes de memoria. Cuando hoy en día los ordenadores tienen mucho más de un Gigabyte de **memoria RAM**, 40 Kbytes se pueden considerar bien empleados a cambio de mejorar la precisión de los cálculos.

Los operadores disponibles para los tipos reales son las operaciones matemáticas usuales: **suma, resta, multiplicación, división real (con decimales)**

3.3.5 Lógicos o Booleanos

Todos los lenguajes incluyen un tipo de **datos lógicos o booleanos. Son datos que sólo pueden tomar dos valores distintos, (verdadero o falso, sí o no, 0 ó 1.)** Se utilizan normalmente para comprobar el valor de una condición, o una comparación de valores o para distinguir la ocurrencia de un suceso de entre dos posibles.

Los operadores habituales para este tipo de datos son:

- la conjunción lógica o Y-lógico (AND),
- la disyunción lógica u O-lógico (OR) y
- la negación lógica o NO-lógico (NOT).

La **tabla de verdad** de estos operadores, suponiendo a y b variables de tipo lógico, es la siguiente:

a	NOT a
V	F
F	V

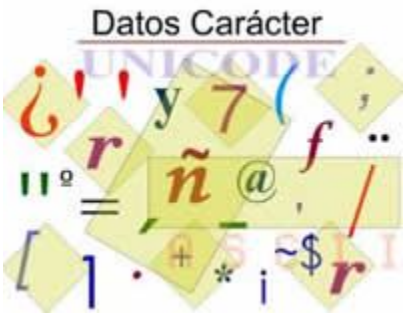
a	b	a AND b	a OR b
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

Existen otros operadores lógicos, pero sólo los tres indicados anteriormente son imprescindibles, por lo que son los que están disponibles en todos los lenguajes.

En Java, el tipo lógico recibe el nombre de ***boolean***, y los valores posibles son: ***true*** (verdadero) y ***false*** (falso).

- El operador **! es la negación lógica**. (NOT)
- El operador **|| es la disyunción lógica**. (OR)
- El operador **&& es la conjunción lógica**. (AND)

3.3.6 Carácter



Los datos de tipo carácter representan elementos individuales del conjunto de caracteres usado como código de entrada-salida. El código más usual es el **ASCII**, que usa 8 bits (1 byte) para representar cada carácter, por lo que permite usar hasta 256 caracteres distintos, lo cual es bastante limitado a la hora de representar todos los posibles símbolos de todos los idiomas y alfabetos existentes. Aunque es el más extendido, no es el único posible. Por ejemplo, Java ha incorporado como alfabeto el código **Unicode**, más reciente que el código ASCII, y que usa 16 bits para representar cada carácter, por lo que puede representar hasta 65.536 caracteres distintos, lo que hace posible representar todos los alfabetos de todas las lenguas, y todo tipo de símbolos gráficos, matemáticos, caracteres de control, etc. Además, el código Unicode es "compatible" con el código ASCII, ya que para los caracteres del código ASCII, Unicode asigna como código los mismos 8 bits, a los que les añade a la izquierda otros 8 bits todos a cero. La conversión de un carácter ASCII a Unicode es inmediata.

No suele haber operadores asociados al tipo carácter, salvo algunas funciones que obtienen el código de un carácter concreto, o que dado un número obtienen el carácter cuyo código es ese número.

En Java, el tipo carácter se llama ***char***, y el alfabeto o código usado es Unicode. Tiene además la peculiaridad de que es considerado por el compilador como un tipo de entero, por lo que puede operarse con caracteres como si fuesen números enteros, ya que realmente se opera con su código Unicode, que es un número.

3.4 Literales de los tipos primitivos

Un **literal**, **valor literal** o **constante literal** es un valor concreto para los tipos de datos primitivos del lenguaje, el tipo **String** o el tipo **null** (**null** no está diseñado como un número entero, carácter u otro tipo de dato específico. Debido a esto, es a veces deseable convertir explícitamente **null** en un tipo de dato específico). Los **literales booleanos** tienen dos únicos valores que puede aceptar el tipo: **true** y **false**. Por ejemplo, con la instrucción `boolean encontrado = true;` estamos declarando una variable de tipo booleana a la cual le asignamos el valor literal **true**.

Los **literales enteros** se pueden representar en tres notaciones:

- **Decimal**: por ejemplo 20. Es la forma más común.
- **Octal**: por ejemplo 024. Un número en octal siempre empieza por cero, seguido de dígitos octales (del **0** al **7**).
- **Hexadecimal**: por ejemplo 0x14. Un número en hexadecimal siempre empieza por **0x** seguido de dígitos hexadecimales (del **0** al **9**, de la '**a**' a la '**f**' o de la '**A**' a la '**F**').

A las constantes literales de tipo **long** se le debe añadir detrás una **l** ó **L**, por ejemplo 873L, si no se considera por defecto de tipo **int**. Se suele utilizar **L** para evitar la confusión de la ele minúscula con 1.

Los **literales reales** o en coma flotante se expresan con coma decimal o en notación científica, o sea, seguidos de un exponente **e** ó **E**. El valor puede finalizarse con una **f** o una **F** para indica el formato **float** o con una **d** o una **D** para indicar el formato **double** (por defecto es **double**). Por ejemplo, podemos representar un mismo literal real de las siguientes formas: **13.2**, **13.2D**, **1.32e1**, **0.132E2**. Otras constantes literales reales son por ejemplo: **.54**, **31.21E-5**, **2.f**, **6.022137e+23f**, **3.141e-9d**.

Un **literal carácter** puede escribirse como un carácter entre comillas simples como 'a', 'ñ', 'Z', 'p', etc. o por su código de la tabla Unicode, anteponiendo la secuencia de escape '****' si el valor lo ponemos en octal o '**\u**' si ponemos el valor en hexadecimal. Por ejemplo, si sabemos que tanto en ASCII como en Unicode, la letra A (mayúscula) es el símbolo número 65, y que 65 en octal es 101 y 41 en hexadecimal, podemos representar esta letra como '**\101**' en octal y '**\u0041**' en hexadecimal.

Existen unos caracteres especiales que se representan utilizando secuencias de escape:

Secuencia de escape	Significado	Secuencia de escape	Significado
\b	Retroceso	\r	Retorno de carro
\t	Tabulador	\"	Carácter comillas dobles
\n	Salto de línea	\'	Carácter comillas simples
\f	Salto de página	\\	Barra diagonal

Los **literales de cadenas de caracteres** se indican entre comillas dobles. Al construir una cadena de caracteres se puede incluir cualquier carácter Unicode excepto un carácter de retorno de carro, por ejemplo en la siguiente instrucción utilizamos la secuencia de escape **\"** para escribir dobles comillas dentro del mensaje:

```
String texto = "Juan dijo: \"Hoy hace un día fantástico…\"";
```

3.5 Tipos referenciados

A partir de los ocho tipos datos primitivos, se pueden construir otros tipos de datos. Estos tipos de datos se llaman **tipos referenciados** o **referencias**, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
int[] arrayDeEnteros;
Cuenta cuentaCliente;
```

En la primera instrucción declaramos una lista de números del mismo tipo, en este caso, enteros. En la segunda instrucción estamos declarando la variable u objeto cuentaCliente como una referencia de tipo Cuenta.

Cuando un conjunto de datos que utiliza una aplicación tiene características similares se suelen agrupar en estructuras para facilitar el acceso a los mismos, son los llamados datos estructurados.

Son datos estructurados los **arrays, listas, árboles**, etc. Pueden estar en la memoria del programa en ejecución, guardados en el disco como ficheros, o almacenados en una base de datos.

Además de los ocho tipos de datos primitivos que ya hemos visto, Java proporciona un tratamiento especial a los textos o cadenas de caracteres mediante el tipo de dato String. Java crea automáticamente un nuevo objeto de tipo String cuando se encuentra una cadena de caracteres encerrada entre comillas dobles. En realidad se trata de objetos, y por tanto son tipos referenciados, pero se pueden utilizar de forma sencilla como si fueran variables de tipos primitivos:

```
String mensaje;
mensaje = "El primer programa";
```

Los tipos de datos referenciados los veremos más adelante con mayor profundidad.

4 Datos Estructurados

Hay datos de la vida real que no se pueden representar y procesar usando únicamente los tipos de datos básicos. Un empleado de la empresa es algo más que una serie de datos individuales, podemos verlo como una estructura que aglutina todos esos datos de tipo básico. Otro **ejemplo** podría ser una serie de valores que represente, con fines estadísticos, la altura en cm de una muestra de 5.000 personas, sobre los que tenemos que calcular variables estadísticas (media, moda, mediana, varianza, desviación típica, etc.) Usar 5.000 variables de tipo entero sería inmanejable porque habría que escribir la suma de las 5.000 variables en una expresión, cada una con su nombre. Es por eso que necesitamos **estructuras de datos**, que nos faciliten la tarea de **gestionar** gran cantidad de datos, o que sencillamente nos permitan representar los datos reales de una forma más fiel.

Además de los tipos simples, o básicos o primitivos, **la mayoría de los lenguajes permiten usar una serie de estructuras de datos algo más complejas, que se construyen a partir de otros tipos, que pueden ser los propios tipos básicos o primitivos o ser a su vez tipos estructurados, o tipos definidos por el usuario**. En este apartado se han incluido los tipos estructurados que suelen proporcionar ya definidos los lenguajes, aunque el usuario puede definir también sus propias estructuras de datos a través de la definición de objetos, que son estructuras y podrían incluirse aquí.

4.1 Cadenas de caracteres (en Java son objetos de la clase String)

En general, **una cadena de caracteres es una sucesión de caracteres (letras, números y demás caracteres del alfabeto del lenguaje)**. Se utilizan normalmente como un tipo de dato predefinido, para palabras, frases o cualquier otra sucesión de caracteres. Un **ejemplo** de valor posible para una cadena de caracteres es:

- el nombre de una persona,
- una frase de un libro,
- la contraseña de un usuario que accede a un curso a través de internet, o
- el número de teléfono, que aunque no contenga más que dígitos entre sus caracteres, podemos almacenarlo como de tipo cadena de caracteres, ya que no vamos a realizar ningún tipo de operación matemática con él.

Es muy habitual que se delimiten mediante comillas (") al principio y al final de la cadena. Como un **ejemplo** concreto podemos ver la cadena de caracteres:

```
"Esto es una cadena de caracteres"
```

Como ya hemos visto en el apartado de los literales de tipos primitivos, para poder mostrar, por ejemplo, una comilla (") dentro de la cadena y no tener problemas con las comillas que la delimitan, se usan **secuencias de escape**. Esto se aplica a otros caracteres reservados o no imprimibles como el retorno de carro o salto de línea. No obstante, las expresiones para producir estas secuencias de escape dependen del lenguaje de programación que se esté usando.

Una forma común, en muchos lenguajes y en Java en concreto, de "escapar" un carácter es anteponiéndole un «\» (sin comillas). Por ejemplo:

```
"la cadena \"ejemplo\" contiene comillas"
```

realmente será interpretada por el lenguajes como la cadena:

```
la cadena "ejemplo" contiene comillas
```

Algunas operaciones comunes con cadenas son:

- **Concatenar dos cadenas de caracteres para formar una nueva**.

  En Java, la concatenación de cadenas se consigue con el operador +. Ejemplo con Java:

  `"El empleado encargado de la sección es "+"José García"` genera la cadena:

  `El empleado encargado de la sección es José García`

- **Obtener una subcadena a partir de otra**. Ejemplo con Java:

  `"El empleado encargado".substring(3,11)` devuelve la subcadena `empleado`

- **Obtener el carácter de una posición de la cadena**. Ejemplo con Java:

  `"Empleado".charAt(2)` devuelve el carácter de la posición 3, que es `'p'`(La primera posición es la cero).

- **Buscar una subcadena o un carácter dentro de la cadena, y saber en qué posición se encuentra**.

  Ej.: `"Empleado".indexOf('p')` devolvería un `2`.

- **Calcular la longitud de una cadena**

  `"Empleado".length()` devuelve el valor 8.

- **Comprobar si empieza o termina con una secuencia de caracteres**

  `"Empleado".startsWith("Em")` devuelve `true` (verdadero).

  `"Empleado".endsWith("a")` devuelve `false` (falso)

- **Reemplazar parte de una cadena por otra**

  `"El empleado encargado".replaceAll("e","X")` cambia todas las `'e'` por `'X'`, quedando la cadena como `El XmplXado Xncargado.`

  Fíjate que sin embargo la letra 'E' no cambia, ya que para el lenguaje se trata de un carácter totalmente independiente a la letra 'e'.

En Java, las cadenas de caracteres se forman como objetos de las clases ***String***, o ***StringBuffer***, que ya vienen incluidas como parte del kit de desarrollo Java (JDK), pero que pueden considerarse como "extensiones" al núcleo del lenguaje.

4.2 Vectores y Matrices (Arrays)

Un *array* es un conjunto de variables o registros del mismo tipo que puede estar almacenado en memoria principal o en memoria auxiliar.

- Los arrays de 1 dimensión se denominan ***vectores***,
- los de 2 o más dimensiones se denominan ***matrices***.

En definitiva **un array es un conjunto de posiciones consecutivas de memoria, en la que se guardan datos del mismo tipo, y que recibe un único nombre, el nombre del array:**

- **Cada posición guarda un dato individual,**
- **tiene un número asociado (un índice) que indica el lugar que ocupa dentro del array**
- **Para acceder directamente a un dato individual (para darle valor, modificarlo o consultarlo...) lo hacemos mediante el nombre del array seguido del índice.**

El nombre del **array** localiza la dirección de comienzo en memoria de la estructura, mientras que el índice representa el desplazamiento respecto a esa dirección de comienzo para llegar al elemento deseado.

Al declarar un array, deberemos decir cuántas **dimensiones** tiene, y cuantos **elementos** en cada dimensión. Algunos lenguajes obligan a hacer esto durante la compilación del programa (en tiempo de compilación), y otros permiten hacerlo durante la ejecución del programa (en tiempo de ejecución). En lo que suelen coincidir es en no limitar el número de dimensiones posibles. A nosotros nos cuesta imaginar más de tres dimensiones, porque lo asociamos al espacio, pero conceptualmente, no hay por qué limitarlo.

Por **ejemplo**, podemos representar las distintas mesas o puestos de trabajo de una empresa, de forma que:

1. La **primera** dimensión represente el edificio concreto de la empresa.
2. Dentro de cada edificio hay varias plantas, representadas por la **segunda** dimensión.
3. Dentro de cada planta, hay varios pasillos, representados por la **tercera** dimensión.
4. Dentro de cada pasillo, hay distintos despachos, representados por la **cuarta** dimensión, y
5. dentro de cada despacho hay varias mesas, representadas por la **quinta** dimensión.

Cuando hablamos de matrices, vectores y arrays, tendemos a limitar el concepto de array al de tabla, pero en informática el concepto de array o tabla es mucho más amplio.

En Java, para **definir un array**, se pone el tipo de los elementos individuales, seguido de un corchete por cada dimensión que queramos darle, y del nombre del array. Posteriormente habrá que dimensionarlo, indicando cuantos elementos va a tener, y posteriormente se usará para almacenar valores con algún propósito. Operaciones típicas con arrays son ***recorrerlo*** procesando todos sus elementos (llenarlo, etc.), ***consultar*** el valor de un elemento, ***modificar*** un elemento (borrarlo, darle valor, actualizarlo)

Ejemplo en Java: (sólo para que vaya sonando)

```
...
int [ ][ ] arrayBidimensionalDeEnteros;
arrayBidimensionalDeEnteros = new int[3][4];
arrayBidimensionalDeEnteros[0][0]=23;
...
arrayBidimensionalDeEnteros[2][3]=376;
```

4.3 Registros

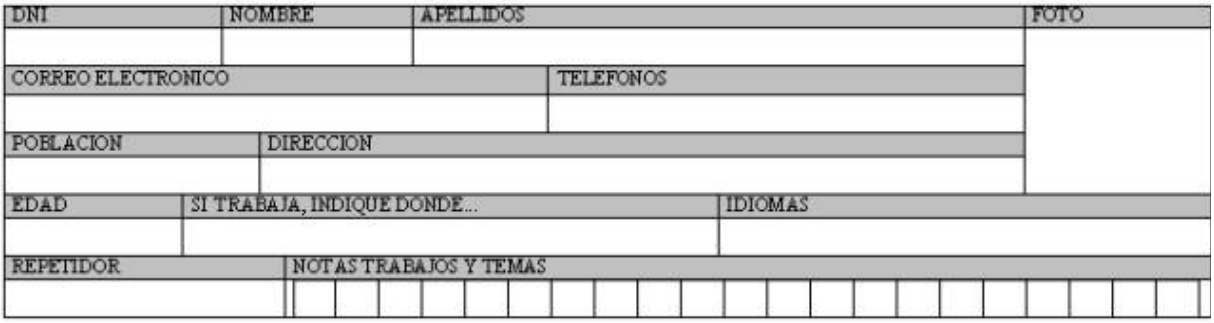
Un registro es una estructura de datos formada por yuxtaposición de elementos de distinto tipo que contiene información relativa a un mismo ente u objeto. A cada uno de los elementos que componen el

registro se les llama ***campos.*** Los campos aparecen en un orden determinado y se identifican por un nombre. **Para definir el registro, hay que darle un nombre y un tipo a cada campo**. **El tipo de cada campo puede ser una estructura de datos a su vez.**

Por **ejemplo**, para representar la información de un alumno en un módulo profesional, que requiere asignarle una nota para el trabajo final de cada una de las 21 unidades de que consta, se puede usar un registro ***alumno*** que contenga los siguientes campos: ***apellidos*** (cadena de caracteres), ***nombre*** (cadena de caracteres), ***edad*** (entero), ***sexo*** (carácter), ***repetidor*** (lógico) ***notasTrabajosTemas*** (vector de números reales, de 21 elementos)

Posteriormente podré crear variables de tipo a `lumno` que contendrán los datos de un alumno concreto.

- **alumno**



No existen operaciones fijas a realizar con un registro, pero las habituales son:

- consultar sus valores,
- modificarlos y actualizarlos,
- crearlos y
- borrarlos.

En Java no existen los registros como tales, pero se puede crear este tipo de estructuras como objetos de una **clase**. En la clase definiré la estructura del objeto, indicando los campos de que consta, y el tipo de cada uno.

4.4 Archivos o Ficheros

Un archivo es un conjunto de información sobre un mismo tema, tratada como unidad de almacenamiento y organizada de forma estructurada para la búsqueda y recuperación de un dato individual. El registro es la estructura que necesitamos usar para poder guardar los datos e informaciones en soportes de almacenamiento masivo o permanente, tales como discos duros, disquetes, CD's, memorias Flash, cintas magnéticas, etc. y poder recuperarlos cada vez que quiera usarlos en una aplicación sin tener que volver a introducirlos de nuevo. Si sólo se guardan en la memoria principal del ordenador, se perderán cuando éste se apague.

Habitualmente los **ficheros o archivos** están compuestos por una colección de registros homogéneos. Así, por **ejemplo** lo lógico es que los datos de los alumnos del módulo profesional se guardaran en memoria permanente, en el disco duro de algún ordenador. Para ello lo normal es que se cree un fichero de alumnos que sería una colección de registros alumno, como los definidos en el apartado anterior.

Las operaciones típicas a realizar con un fichero son:

- Creación del fichero.
- Inserción de un registro
- Eliminación o borrado de un registro
- Consulta de uno, varios o todos los registros.
- Modificación o actualización de un registro.

- Borrado del fichero.
- Búsqueda y localización de un registro concreto (normalmente habrá que hacerlo antes de cualquier operación de recuperación o actualización de un registro.)
- Duplicado o copia de seguridad del fichero.
- Ordenación del fichero.
- Mezcla de los datos de varios ficheros para formar uno nuevo.

Muchas de las tareas de manipulación de los ficheros las realizará el sistema operativo y los programas que hagamos, normalmente realizarán peticiones al sistema operativo para gestionar los archivos. Eso permite que nuestro programa no tenga que entrar en los detalles concretos de los soportes de almacenamiento del ordenador en el que se ejecuta. De eso se encargará el Sistema Operativo por lo que el programa funcionará con cualquier dispositivo y en cualquier ordenador.

4.5 Listas

Son estructuras de datos dinámicas que se gestionan en la memoria RAM utilizando punteros.

Una lista enlazada está compuesta por:

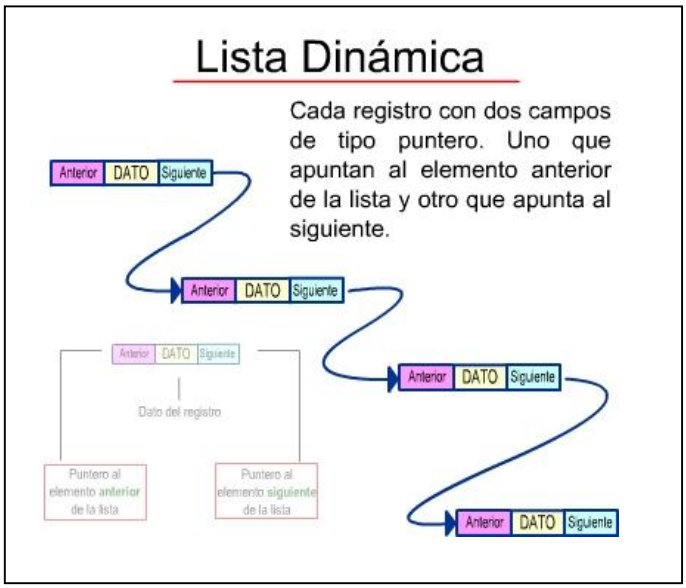
- **Distintos elementos o nodos**, cada uno de los cuales será un registro del mismo tipo, dispuestos de forma secuencial.
- **En cada nodo se incluye un campo que sea de tipo "puntero al siguiente nodo de la lista**"
- **Un puntero de inicio que siempre apunte al primer elemento de la lista**.
- **El campo "puntero al siguiente nodo de la lista" del último nodo debe apuntar a un valor especial**, que normalmente recibe el nombre de ***Null***, y que indica que no hay ningún otro elemento detrás.

Por **ejemplo**, en el caso de que queramos hacer una lista de empleados, necesitaremos punteros de tipo empleado. Un puntero apuntará siempre al primer empleado de la lista.

Cada nodo de la lista será un registro empleado, al que además de la información del empleado habremos añadido un puntero de tipo empleado, al que llamamos "puntero al siguiente empleado".

El puntero al siguiente empleado apuntará la siguiente empleado (contendrá la dirección de memoria donde está el registro del siguiente empleado).

El último empleado de la lista tendrá un valor Null, para indicar que no hay más registros de empleado en la lista.



4.6 Árboles

Al igual que las listas, **un árbol es una estructura de datos dinámica, que se construye usando punteros**. La diferencia es que se trata de una **estructura jerárquica**.

En un árbol tendremos:

- **Un nodo llamado raíz del árbol** por el que se accede a la estructura.
- **Cada nodo puede tener varios nodos hijos**, cada uno de los cuales se puede considerar como la raíz de un nuevo subárbol, y que constituye una rama del árbol general.

- **Cada nodo deberá contener campos de tipo "puntero a nodo hijo"** para cada uno de los posibles hijos.
- **Para un nodo concreto, sólo existe un nodo padre**, por lo que un nodo no puede pertenecer más que a una rama del árbol.
- **Cuando una rama se termina, los campos que apuntan a los hijos, deben contener el valor Null**.
- **Es necesario disponer de un puntero que apunte siempre al nodo raíz del árbol**, para poder acceder a la estructura, ya que si no sería imposible usarla, aunque estuviera en memoria ocupando espacio.

5 Datos Definidos por el usuario

Algunos lenguajes permiten **definir** datos al **usuario** mediante distintos mecanismos. De ellos, los más útiles son las **clases** que nos permiten crear **objetos**, en aquellos lenguajes que permiten programación orientada a objetos, ya que sí permiten definir auténticas estructuras de datos, tan complejas como queramos, en cuanto a valores posibles como a operaciones permitidas. Los tipos enumerados son menos útiles, y hay muchos lenguajes que ni siquiera disponen de mecanismos para definirlos o usarlos.

5.1 Enumerados

Los **tipos de datos enumerados** son una forma de declarar una variable con un conjunto restringido de valores. Por ejemplo, los días de la semana, las estaciones del año, los meses, etc.

La forma de declararlos es con la palabra reservada **enum**, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A los valores que se colocan dentro de las llaves se les considera como constantes, van separados por comas y deben ser valores únicos.

La lista de valores se coloca entre llaves, porque un tipo de datos **enum** no es otra cosa que una especie de clase en Java, y todas las clases llevan su contenido entre llaves.

Al considerar Java este tipo de datos como si de una clase se tratara, no sólo podemos definir los valores de un tipo enumerado, sino que también podemos definir operaciones a realizar con él y otro tipo de elementos, lo que hace que este tipo de dato sea más versátil y potente que en otros lenguajes de programación.

5.2 Objetos

Decir que algo es un objeto viene a ser como decir que has usado un cacharro. Es una definición bastante ambigua, en principio. Pero eso es lo que deliberadamente queremos en el caso de los objetos. ¿Qué es un objeto? ¿Qué nos aporta? ¿Para qué sirve?

Los **objetos** tratan de permitir una libertad total al definir un tipo de dato:

- Nos permiten definir cualquier estructura de datos,
- cualquier conjunto de valores posibles,
- cualquier conjunto de operaciones que se pueden realizar con esos datos
- cualquier grado de libertad en la definición que permita adaptar los programas a las cosas del mundo real.

En la vida todo son **objetos**, y lo que tratamos es de que el programador pueda hacer un modelo de tipo de datos a la medida del objeto de la vida real que quiere representar.

Realmente se trata de un tipo estructurado y definido por el usuario. Está disponible para los lenguajes orientados a objetos, que son la mayoría de los que se usan actualmente.

Nos proporcionan un mecanismo realmente potente y flexible para definir cualquier tipo de datos que deseemos, o cualquier estructura de datos que podamos imaginar.

A través de la definición de una clase, se define una estructura de datos, tanto en lo relativo a qué valores va a tomar como en lo relativo a qué operaciones se van a poder aplicar a esos valores.

En la clase se puede definir la estructura del objeto de forma similar a como se definía un registro, **indicando qué campos (variables miembro de objeto) va a tener, y de qué tipo van a ser, junto a otras características que van a limitar quien va a tener acceso a cada campo, y qué tipo de acceso.** Cada variable que forme parte del objeto puede ser de cualquier tipo estructurado, incluyendo cualquier otra clase ya definida, o incluso la propia clase (similar a los punteros de las listas enlazadas)

Por otro lado, en la clase podemos definir todo un conjunto de métodos (o procedimientos) en los que se define qué uso se va a poder hacer de esos datos, junto a restricciones de acceso y seguridad.

Y toda esa estructura se puede crear como una auténtica caja negra, de la que conozco qué le puedo dar y qué puedo esperar, pero nada acerca de cómo se realizan los procesos internamente. **A esto se le llama ocultación de la información**.

5.3 Características que definen los objetos a través de un ejemplo

Los objetos serán variables del tipo que define la clase, aunque en la terminología de la programación orientada a objetos se les suele llamar instancias de la clase.

Por ejemplo, yo puedo definir la clase **`Alumno`**, en la que indico:

- **los campos que va a tener un objeto de tipo Alumno**:
    - **apellidos** (cadena de caracteres)
    - **nombre** (cadena de caracteres)
    - **edad** (entero)
    - **sexo** (carácter)
    - **repetidor** (lógico)
    - **notasTrabajosTemas** (vector de números reales, de 21 elementos)

Con este apartado estoy definiendo **la estructura de los datos de este tipo**.

- **Las operaciones a realizar con un alumno:**
    - **Crearlo**
    - **Ponerle nombre**

- **Ponerle apellidos**
- **Rellenar cualquier otro campo**
- **Actualizar los valores**
- **Imprimirlo**
- **ponerle nota en el trabajo de un tema**
- **calcularle la nota media,** si ya tiene nota en todos los trabajos.

Con este último apartado estoy definiendo también **el comportamiento de los objetos que sean del tipo `Alumno`**.

Cada variable que declare como de tipo **`Alumno`** va a ser una nueva instancia de la clase, o lo que es lo mismo, un nuevo objeto de esa clase.

Además esto aporta otra serie de beneficios, como la facilidad para reutilizar código, que se verán con más detalle en las unidades correspondientes a programación orientada a objetos.

6 Operadores y expresiones

Los **operadores** llevan a cabo operaciones sobre un conjunto de datos u operandos, representados por literales y/o identificadores. Los operadores pueden ser unarios, binarios o terciarios, según el número de operandos que utilicen sean uno, dos o tres, respectivamente. Los operadores actúan sobre los tipos de datos primitivos y devuelven también un tipo de dato primitivo.

Los operadores se combinan con los literales y/o identificadores para formar expresiones. Una **expresión** es una combinación de operadores y operandos que se evalúa produciendo un único resultado de un tipo determinado.

El resultado de una expresión puede ser usado como parte de otra expresión o en una sentencia o instrucción. Las expresiones, combinadas con algunas palabras reservadas o por sí mismas, forman las llamadas **sentencias** o **instrucciones**.

Por ejemplo, pensemos en la siguiente expresión Java:

```
i + 1
```

Con esta expresión estamos utilizando un operador aritmético para sumarle una cantidad a la variable **`i`**, pero es necesario indicar al programa qué hacer con el resultado de dicha expresión:

```
suma = i + 1;
```

Que lo almacene en una variable llamada **`suma`**, por ejemplo. En este caso ya tendríamos una acción completa, es decir, una sentencia o instrucción.

Más ejemplos de sentencias o instrucciones los tenemos en las declaraciones de variables, vistas en apartados anteriores, o en las estructuras básicas del lenguaje como sentencias condicionales o bucles, que veremos más adelante.

Las expresiones de asignación, al poder ser usadas como parte de otras asignaciones u operaciones, son consideradas tanto expresiones en sí mismas como sentencias.

6.1 Operadores aritméticos

Los operadores aritméticos son aquellos operados que combinados con los operandos forman expresiones matemáticas o aritméticas.

Operador	Operación Java	Expresión Java	Resultado
-	Operador unario de cambio de signo	-10	-10
+	Adición	1.2 + 9.3	10.5
-	Sustracción	312.5 – 12.3	300.2
*	Multiplicación	1.7 * 1.2	1.02
/	División (entera o real)	0.5 / 0.2	2.5
%	Resto de la división entera	25 % 3	1

El resultado de este tipo de expresiones depende de los operandos que utilicen:

Tipo de los operandos	Resultado
Un operando de tipo long y ninguno real (float o double)	long
Ningún operando de tipo long ni real (float o double)	int
Al menos un operando de tipo double	double
Al menos un operando de tipo float y ninguno double	float

Otro tipo de operadores aritméticos son los operadores unarios incrementales y decrementales.

Producen un resultado del mismo tipo que el operando, y podemos utilizarlos con notación prefija, si el operador aparece antes que el operando, o notación postfija, si el operador aparece después del operando. En la tabla puedes ver un ejemplo de utilización de cada operador incremental.

Tipo operador	Expresión Java	
++ (incremental)	Prefija:	Postfija:
	x=3;	x=3;
	y=++x;	y=x++;
	// x vale 4 e y vale 4	// x vale 4 e y vale 3
--(decremental)	5-- // el resultado es 4	

6.2 Operadores de asignación

El principal operador de esta categoría es el operador asignación **=**, que permite al programa darle un valor a una variable, y que ya hemos utilizado en varias ocasiones en esta unidad. Además de este operador, Java proporciona otros operadores de asignación combinados con los operadores aritméticos, que permiten abreviar o reducir ciertas expresiones.

Por ejemplo, el operador **+=** suma el valor de la expresión a la derecha del operador con el valor de la variable que hay a la izquierda del operador, y almacena el resultado en dicha variable. En la siguiente tabla se muestran todos los operadores de asignación compuestos que podemos utilizar en Java

Operador	Ejemplo en Java	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

6.3 Operador condicional

El operador condicional **?:** sirve para evaluar una condición y devolver un resultado en función de si es verdadera o falsa dicha condición. Es el único operador ternario de Java, y como tal, necesita tres operandos para formar una expresión.

El primer operando se sitúa a la izquierda del símbolo de interrogación, y siempre será una expresión booleana, también llamada condición. El siguiente operando se sitúa a la derecha del símbolo de interrogación y antes de los dos puntos, y es el valor que devolverá el operador condicional si la condición es verdadera. El último operando, que aparece después de los dos puntos, es la expresión cuyo resultado se devolverá si la condición evaluada es falsa.

```
condición ? exp1 : exp2
```

Por ejemplo, en la expresión:

```
(x>y)? x : y;
```

Se evalúa la condición de si **x** es mayor que **y**, en caso afirmativo se devuelve el valor de la variable **x**, y en caso contrario se devuelve el valor de **y**.

El operador condicional se puede sustituir por la sentencia **if...then...else** que veremos más adelante.

6.4 Operadores de relación

Los operadores relacionales se utilizan para comparar datos de tipo primitivo (numérico, carácter y booleano). El resultado se utilizará en otras expresiones o sentencias, que ejecutarán una acción u otra en función de si se cumple o no la relación.

Operador	Ejemplo en Java	Significado
==	op1 == op2	op1 igual a op2
!=	op1 != op2	op1 distinto de op2
>	op1 > op2	op1 mayor que op2
<	op1 < op2	op1 menor que op2
>=	op1 >= op2	op1 mayor o igual que op2
<=	op1 <= op2	op1 menor o igual que op2

Estas expresiones en Java dan siempre como resultado un valor booleano **true** o **false**. En la tabla aparecen los operadores relacionales en Java.

6.5 Operadores lógicos

Los operadores lógicos realizan operaciones sobre valores booleanos, o resultados de expresiones relacionales, dando como resultado un valor booleano.

Los operadores lógicos los podemos ver en la tabla que se muestra a continuación. Existen ciertos casos en los que el segundo operando de una expresión lógica no se evalúa para ahorrar tiempo de ejecución, porque con el primero ya es suficiente para saber cuál va a ser el resultado de la expresión.

Por ejemplo, en la expresión **a && b** si **a** es falso, no se sigue comprobando la expresión, puesto que ya se sabe que la condición de que ambos sean verdadero no se va a cumplir. En estos casos es más conveniente colocar el operando más propenso a ser falso en el lado de la izquierda. Igual ocurre con el operador **||**, en cuyo caso es más favorable colocar el operando más propenso a ser verdadero en el lado izquierdo.

Operador	Ejemplo en Java	Significado
!	!op	Devuelve **true** si el operando es **false** y viceversa.
&	op1 & op2	Devuelve **true** si op1 y op2 son **true**
|	op1 | op2	Devuelve **true** si op1 u op2 son **true**
^	op1 ^ op2	Devuelve **true** si sólo uno de los operandos es **true**
&&	op1 && op2	Igual que &, pero si op1 es **false** ya no se evalúa op2
||	op1 || op2	Igual que |, pero si op1 es **true** ya no se evalúa op2

6.6 Operadores de bits

Además hay operadores de bits que nosotros no vamos a utilizar de momento.

6.7 Precedencia de operadores

El orden de precedencia de los operadores determina la secuencia en que deben realizarse las operaciones cuando en una expresión intervienen operadores de distinto tipo.

Las reglas de precedencia de operadores que utiliza Java coinciden con las reglas de las expresiones del álgebra convencional. Por ejemplo:

- La multiplicación, división y resto de una operación se evalúan primero.
  Si dentro de la misma expresión tengo varias operaciones de este tipo, empezaré evaluándolas de izquierda a derecha.
- La suma y la resta se aplican después que las anteriores. De la misma forma, si dentro de la misma expresión tengo varias sumas y restas empezaré evaluándolas de izquierda a derecha.

A la hora de evaluar una expresión es necesario tener en cuenta la **asociatividad** de los operadores. La asociatividad indica qué operador se evalúa antes, en condiciones de igualdad de precedencia. Los operadores de **asignación**, el operador condicional **(?:)**, los operadores incrementales (++, --) y el **casting** son asociativos por la derecha. El resto de operadores son asociativos por la izquierda, es decir, que se empiezan a calcular en el mismo orden en el que están escritos: de izquierda a derecha.

Por ejemplo, en la expresión siguiente:

```
10 / 2 * 5
```

Realmente la operación que se realiza es **(10 / 2) * 5**, porque ambos operadores, división y multiplicación, tienen igual precedencia y por tanto se evalúa primero el que antes nos encontramos por la izquierda, que es la división. El resultado de la expresión es **25**. Si fueran asociativos por la derecha, puedes comprobar que el resultado sería diferente, primero multiplicaríamos **2 * 5** y luego dividiríamos entre **10**, por lo que el resultado sería **1**. En esta otra expresión:

```
x = y = z = 1
```

Realmente la operación que se realiza es **x = (y = (z = 1))**. Primero asignamos el valor de **1** a la variable **z**, luego a la variable **y**, para terminar asignando el resultado de esta última asignación a **x**. Si el operador asignación fuera asociativo por la izquierda esta operación no se podría realizar, ya que intentaríamos asignar a **x** el valor de **y**, pero **y** aún no habría sido inicializada.

En la tabla se detalla el orden de precedencia y la asociatividad de los operadores que hemos comentado en este apartado. Los operadores se muestran de mayor a menor precedencia.

Operador	Tipo	Asociatividad
++ --	Unario, notación postfija	Derecha
++ -- + - (cast) ! ~	Unario, notación prefija	Derecha
* / %	Aritméticos	Izquierda
+ -	Aritméticos	Izquierda
<< >> >>>	Bits	Izquierda
< <= > >=	Relacionales	Izquierda
== !=	Relacionales	Izquierda
&	Lógico, Bits	Izquierda
^	Lógico, Bits	Izquierda
|	Lógico, Bits	Izquierda
&&	Lógico	Izquierda
||	Lógico	Izquierda
?:	Operador condicional	Derecha
= += -= *= /= %=	Asignación	Derecha

7 Conversión de tipo

Imagina que queremos dividir un número entre otro ¿tendrá decimales el resultado de esa división? Podemos pensar que siempre que el dividendo no sea divisible entre el divisor, tendremos un resultado con decimales, pero no es así. Si el dividendo y el divisor son variables de tipo entero, el resultado será entero y no tendrá decimales. Para que el resultado tenga decimales necesitaremos hacer una **conversión de tipo**.

Las conversiones de tipo se realizan para hacer que el resultado de una expresión sea del tipo que nosotros deseamos, en el ejemplo anterior para hacer que el resultado de la división sea de tipo real y tenga decimales. Existen dos tipos de conversiones:

- **Conversiones automáticas**. Cuando a una variable de un tipo se le asigna un valor de otro tipo numérico con menos bits para su representación, se realiza una conversión automática. En ese caso, el valor se dice que es promocionado al tipo más grande (el de la variable), para poder hacer la asignación. También se realizan conversiones automáticas en las operaciones aritméticas, cuando estamos utilizando valores de distinto tipo, el valor más pequeño se promociona al valor más grande, ya que el tipo mayor siempre podrá representar cualquier valor del tipo menor (por ejemplo, de int a long o de float a double).
- **Conversiones explícitas**. Cuando hacemos una conversión de un tipo con más bits a un tipo con menos bits. En estos casos debemos indicar que queremos hacer la conversión de manera expresa, ya que se puede producir una pérdida de datos y hemos de ser conscientes de ello. Este tipo de conversiones se realiza con el **operador cast**. El operador cast es un operador unario que se forma colocando delante del valor a convertir el tipo de dato entre paréntesis. Tiene la misma precedencia que el resto de operadores unarios y se asocia de izquierda a derecha.

Debemos tener en cuenta que **un valor numérico nunca puede ser asignado a una variable de un tipo menor en rango, si no es con una conversión explícita**. Por ejemplo:

```
int a;
byte b;
a = 12; // no se realiza conversión alguna
b = 12; // se permite porque 12 está dentro del rango permitido de valores para b
b = a; // error, no permitido (incluso aunque 12 podría almacenarse en un byte)
byte b = (byte) a; // Correcto, forzamos conversión explícita
```

En el ejemplo anterior vemos un caso típico de error de tipos, ya que estamos intentando asignarle a **b** el valor de **a**, siendo **b** de un tipo más pequeño. Lo correcto es promocionar **a** al tipo de datos **byte**, y entonces asignarle su valor a la variable **b**.

7.1 Conversión de tipos de datos en Java

Tabla de Conversión de Tipos de Datos Primitivos

		Tipo destino							
		boolean	char	byte	short	int	long	float	double
Tipo origen	boolean	-	N	N	N	N	N	N	N
	char	N	-	C	C	CI	CI	CI	CI
	byte	N	C	-	CI	CI	CI	CI	CI
	short	N	C	C	-	CI	CI	CI	CI
	int	N	C	C	C	-	CI	CI*	CI
	long	N	C	C	C	C	-	CI*	CI*
	float	N	C	C	C	C	C	-	CI
	double	N	C	C	C	C	C	C	-

Explicación de los símbolos utilizados:

- **N:** Conversión no permitida (un **boolean** no se puede convertir a ningún otro tipo y viceversa).
- **CI:** Conversión implícita o automática. Un asterisco indica que puede haber posible pérdida de datos.
- **C:** Casting de tipos o conversión explícita.

El asterisco indica que puede haber una posible pérdida de datos, por ejemplo al convertir un número de tipo **int** que usa los 32 bits posibles de la representación, a un tipo **float**, que también usa 32 bits para la representación, pero 8 de los cuales son para el exponente.

En cualquier caso, las conversiones de números en coma flotante a números enteros siempre necesitarán un **Casting**, y deberemos tener mucho cuidado debido a la pérdida de precisión que ello supone.

7.1.1 Reglas de Promoción de Tipos de Datos

Cuando en una expresión hay datos o variables de distinto tipo, el compilador realiza la promoción de unos tipos en otros, para obtener como resultado el tipo final de la expresión.

Esta promoción de tipos se hace siguiendo unas reglas básicas que sirven de base para realizar esta promoción de tipos, y resumidamente son las siguientes:

- Si uno de los operandos es de tipo **double**, el otro es convertido a **double**.
- En cualquier otro caso:
    - Si el uno de los operandos es **float**, el otro se convierte a **float**
    - Si uno de los operandos es **long**, el otro se convierte a **long**
    - Si no se cumple ninguna de las condiciones anteriores, entonces ambos operandos son convertidos al tipo **int**.

7.1.2 Conversión de números en Coma flotante (float, double) a enteros (int)

Cuando convertimos números en coma flotante a números enteros, la parte decimal se trunca (redondeo a cero). Si queremos hacer otro tipo de redondeo, podemos utilizar, entre otras, las siguientes funciones:

- **Math.round(num):** Redondeo al siguiente número entero.
- **Math.ceil(num):** Mínimo entero que sea mayor o igual a num.
- **Math.floor(num):** Entero mayor, que sea inferior o igual a num.

Ejemplo:

```
double num=3.5;
x=Math.round(num); // x = 4
y=Math.ceil(num); // y = 4
z=Math.floor(num); // z = 3
```

Probad el siguiente ejemplo:

```
public class calculo {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		// aritmética flotante con enteros
		// para obtener decimales de una operación
		// hay que forzar a que el tipo sea (float)
		int a=7,b=3;
		System.out.println("el calculo es:"+(float)a/b);
	}
```

Si pongo el casting explícito (float) da 2.33333 y si no lo pongo da 2.

7.1.3 Conversiones entre caracteres (char) y enteros (int)

Como un tipo **char** lo que guarda en realidad es el código **Unicode** de un carácter, los caracteres pueden ser considerados como números enteros sin signo.

Ejemplo:

```
int num;
char c;
num = (int) 'A'; //num = 65
c = (char) 65; // c = 'A'
c = (char) ((int) 'A' + 1); // c = 'B'
```

7.1.4 Conversiones de tipo con cadenas de caracteres (String)

Para convertir cadenas de texto a otros tipos de datos se utilizan las siguientes funciones:

```
num=Byte.parseByte(cad);
num=Short.parseShort(cad);
num=Integer.parseInt(cad);
num=Long.parseLong(cad);
num=Float.parseFloat(cad);
num=Double.parseDouble(cad);
```

Por ejemplo, si hemos leído de teclado un número que está almacenado en una variable de tipo **String** llamada **cadena**, y lo queremos convertir al tipo de datos **byte**, haríamos lo siguiente:

```
byte n=Byte.parseByte(cadena);
```

8 Comentarios

Los comentarios son muy importantes a la hora de describir qué hace un determinado programa. Todos los lenguajes de programación disponen de alguna forma de introducir comentarios en el código. En el caso de Java, nos podemos encontrar los siguientes tipos de comentarios:

- **Comentarios de una sola línea**. Utilizaremos el delimitador // para introducir comentarios de sólo una línea.

```
// comentario de una sola línea
```

- **Comentarios de múltiples líneas**. Para introducir este tipo de comentarios, utilizaremos una barra inclinada y un asterisco (**/***), al principio del párrafo y un asterisco seguido de una barra inclinada (***/**) al final del mismo.

```
/* Esto es un comentario
de varias líneas */
```

- **Comentarios Javadoc**. Utilizaremos los delimitadores /** y */. Al igual que con los comentarios tradicionales, el texto entre estos delimitadores será ignorado por el compilador. Este tipo de comentarios se emplean para generar documentación automática del programa. A través del programa javadoc, incluido en JavaSE, se recogen todos estos comentarios y se llevan a un documento en formato .html.

```
/** Comentario de documentación.
Javadoc extrae los comentarios del código y
genera un archivo html a partir de este tipo de comentarios
*/
```

9 Entrada y salida en un programa (teclado/pantalla)

```
import java.util.*;
public class entradas {
	public static void main(String[] parametro)
	{
		Scanner teclado = new Scanner(System.in);
		// definimos  un objeto teclado para lectura de datos
		int entero;
		float decimales;
		String mitexto,mitexto2;
		// vamos a leer diferentes tipos de datos
		System.out.println("dame un entero;");
		entero=teclado.nextInt();
		System.out.println("dame un numero con decimales");
		decimales=teclado.nextFloat();
		mitexto=teclado.nextLine();
		/* si se leen texto después de otros datos hay que quitar
		el /n mediante nextLine*/
		System.out.println("dame un texto");
		mitexto=teclado.nextLine();
		System.out.println("el entero fue:"+entero);
		System.out.println("el decimal fue:"+decimales);
		System.out.println("el texto fue:"+mitexto);
	}
}
```

System.out.print
System.out.println

Son usados para salida: uno imprime y el cursor se queda en la línea y el otro imprime y salta a la línea siguiente.

Teclado es el objeto de tipo **Scanner** a través del cual leemos los distintos tipos de datos, si hay error en la entrada el programa se aborta, pero esto lo solucionaremos con el bloque **try..catch** y otras clases de Java que permiten filtrar los datos introducidos como texto que veremos más adelante, en principio supondremos que las entradas dadas serán las correctas.