INDICE

Creación de mi primer programa

Caso Práctico
Todos los lenguajes de programación están constituidos por elementos concretos que permiten realizar las operaciones básicas del lenguaje, como tipos de datos, operadores e instrucciones. Estos conceptos deben ser dominados por el programador con objeto de incorporarse con ciertas garantías a un equipo de programación. Debemos tener en cuenta que los programas trabajan con datos, los cuales almacenan en memoria y son posteriormente usados para la toma de decisiones en el programa.
En esta unidad se introducen los distintos tipos de datos que se pueden emplear en Java. En concreto, se verán los tipos primitivos en Java, así como las variables y las constantes. En posteriores capítulos veremos otros elementos básicos del lenguaje, incluyendo estructuras de datos más sofisticadas.
***María** y **Juan** han formado equipo para desarrollar una aplicación informática para una clínica veterinaria. **Ada** ha convocado una reunión con el cliente para concretar los requerimientos de la aplicación, pero lo que está claro es que debe ser multiplataforma. El lenguaje escogido ha sido Java.*
***María** tiene conocimientos de redes y de páginas web pero está floja en programación. **Juan** ha aprendido Java en su ciclo de DAI hace 4 años.*
*—Lo que hace falta es entender bien los conceptos de programación orientada a objetos —le comenta **Juan** a **María**. —Todo lo concerniente al mundo real puede ser modelado a través de objetos. Un objeto contiene datos y sobre él se hacen operaciones y gira toda la aplicación.*
***María** está entusiasmada con el proyecto, cree que es una buena oportunidad para aprender un lenguaje de la mano de **Juan** que se le da bastante bien todo el mundo de la programación.*

1 - Introducción

Cada vez que usamos un ordenador, estamos ejecutando varias aplicaciones que nos permiten realizar ciertas tareas. Por ejemplo, en nuestro día a día, usamos el correo electrónico para enviar y recibir correos, o el navegador para consultar páginas en Internet; ambas actividades son ejemplos de programas que se ejecutan en un ordenador.

Los programas de ordenador deben resolver un problema, para lo cual debemos utilizar de forma inteligente y lógica todos los elementos que nos ofrece el lenguaje. Por eso es importante elegir un lenguaje de programación con el que nos sintamos cómodos porque lo dominemos suficientemente y, por supuesto, porque sepamos que no va a ofrecer limitaciones a la hora de desarrollar aplicaciones para diferentes plataformas.

El lenguaje que vamos a utilizar en este módulo es Java, multiplataforma, robusto y fiable. Un lenguaje que reduce la complejidad y se considera dentro de los lenguajes modernos orientados a objetos. Esta unidad nos vamos a adentrar en su sintaxis, vamos a conocer los tipos de datos con los que trabaja, las operaciones que tienen definidas cada uno de ellos, utilizando ejemplos sencillos que nos muestren la utilidad de todo lo aprendido.

Para ello, vamos a tratar sobre cómo se almacenan y recuperan los datos de variables y cadenas en Java, y cómo se gestionan estos datos desde el punto de vista de la utilización de operadores. Trabajar con datos es fundamental en cualquier programa. Aunque ya hayas programado en este lenguaje, échale un vistazo al contenido de esta unidad, porque podrás repasar muchos conceptos.

PARA SABER MÁS

Ahora que vamos a empezar con la sintaxis de Java, quizás te interese tener a mano la documentación que ofrece la página web de Oracle sobre Java SE. 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.
Puedes consultar información de la versión 6 de Java SE, en el siguiente enlace donde puedes encontrar toda la documentación sobre esta tecnología:
Enlace a la web de Oracle donde podrás ver información sobre la plataforma Java SE.

http://download.oracle.com/javase/6/docs/index.html

Dentro de la documentación de Oracle sobre Java SE se encuentra el libro "The Java LanguageSpecification".
Este libro está escrito por los inventores del lenguaje, y constituye una referencia técnica casi obligada sobre el mismo. Como mucha de la documentación oficial de Java, se encuentra en inglés. El enlace directo es el siguiente:

http://java.sun.com/docs/books/jls/index.html

RECOMENDACIÓN

Acostúmbrate a leer y consultar la documentación sobre la versión de Java que estés utilizando en tus programas. Eso te ayudará a saber todas las posibilidades que tiene el lenguaje, y si en un momento dado estás utilizando bien una determinada característica.

2 – Las variables e identificadores

Caso práctico
María *y* ***Juan*** *han comprobado que una aplicación informática es un trabajo de equipo que debe estar perfectamente coordinado. El primer paso es la definición de los datos y las variables que se van a utilizar.*
Las decisiones que se tomen pueden afectar a todo el proyecto, en lo referente al rendimiento de la aplicación y ahorro de espacio en los sistemas de almacenamiento.
Después de la última reunión del equipo de proyecto ha quedado claro cuáles son las especificaciones de la aplicación a desarrollar. ***Juan*** *no quiere perder el tiempo y ha comenzado a preparar los datos que va a usar el programa. Le ha pedido a* ***María*** *que vean juntos qué variables y tipos de datos se van a utilizar,* ***Juan*** *piensa que le vendrá bien como primera tarea para familiarizarse con el entorno de programación y con el lenguaje en sí.*

Un programa maneja datos para hacer cálculos, presentarlos en informes por pantalla o impresora, solicitarlos al usuario, guardarlos en disco, etc. Para poder manejar esos datos, el programa los guarda en variables.

Una variable es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa. Las variables vienen determinadas por:

- **un nombre**, que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido.
- **un tipo de dato**, que especifica qué clase de información guarda la variable en esa zona de memoria
- **un rango de valores** que puede admitir dicha variable.

Al nombre que le damos a la variable se le llama **identificador**. Los identificadores permiten nombrar los elementos que se están manejando en un programa. Vamos a ver con más detalle ciertos aspectos sobre los identificadores que debemos tener en cuenta.

CITAS PARA PENSAR
Las grandes ideas requieren un gran lenguaje. Aristófanes.

PARA SABER MÁS
Bruce Eckel es el autor de los libros sobre Java y C++, dirigidos a programadores que desean aprender sobre estos lenguajes y sobre la programación orientada a objetos. Este escritor ha tenido la buena costumbre de editar sus libros para que puedan descargarse gratis. Así, podemos acceder de forma totalmente gratuita a la tercera edición de su libro "Thinking in Java" en el siguiente enlace (en inglés):
http://www.mindviewinc.com/Books/downloads.html

A partir de ahora es conveniente que utilices algún manual de apoyo para iniciarte a la programación en Java.
Te proponemos el de la serie de Libros "Aprenda Informática como si estuviera en primero", de la Escuela Superior de Ingenieros Industriales de San Sebastián (Universidad de Navarra):
http://mat21.etsii.upm.es/ayudainf/aprendainf/Java/Java2.pdf

2.1 – Identificadores

Un **identificador** en Java es una secuencia ilimitada sin espacios de letras y dígitos Unicode, de forma que **el primer símbolo de la secuencia debe ser una letra, un símbolo de subrayado (_) o el símbolo dólar ($)**. Por ejemplo, son válidos los siguientes identificadores:

$x5$ $\alpha\tau\eta$ NUM_MAX $numCuenta$

En la definición anterior decimos que un identificador es una secuencia ilimitada de caracteres Unicode. Pero… ¿qué es Unicode? Unicode es un código de caracteres o sistema de codificación, un alfabeto que recoge los caracteres de prácticamente todos los idiomas importantes del mundo. Las líneas de código en los programas se escriben usando ese conjunto de caracteres Unicode.

Esto quiere decir que en Java se pueden utilizar varios alfabetos como el Griego, Árabe o Japonés. De esta forma, los programas están más adaptados a los lenguajes e idiomas locales, por lo que son más significativos y fáciles de entender tanto para los programadores que escriben el código, como para los que posteriormente lo tienen que interpretar, para introducir alguna nueva funcionalidad o modificación en la aplicación.

El estándar Unicode originalmente utilizaba 16 bits, pudiendo representar hasta 65.536 caracteres distintos, que es el resultado de elevar dos a la potencia dieciséis. Actualmente Unicode puede utilizar más o menos bits, dependiendo del formato que se utilice: **UTF-8**, **UTF-16** ó **UTF32**. **A cada carácter le corresponde unívocamente un número entero perteneciente al intervalo de 0 a 2 elevado a n, siendo n el número de bits utilizados para representar los caracteres.** Por ejemplo, la letra ñ es el entero164. 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.

RECOMENDACIÓN

Una buena práctica de programación es seleccionar nombres adecuados para las variables, eso ayuda a que el programa se autodocumente, y evita un número excesivo de comentarios para aclarar el código.

PARA SABER MÁS

Enlace para acceder a la documentación sobre las distintas versiones de Unicode en la página web oficial del estándar:

http://www.unicode.org/versions/

2.2Normas 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 el valor booleano** (`true` o `false`) **ni el valor nulo** (`null`).
- 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 cliente

sería recomendable que la misma se llamara algo así como FicheroClientes o ManejadorCliente, y no algo poco descriptivo como Cl33.

Además de estas restricciones, en la siguiente tabla puedes ver otras convenciones, que no siendo obligatorias, síson 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 con letra minúscula	`modifica_Valor, obtiene_Valor`

AUTOEVALUACIÓN:
Un identificador es una secuencia de uno o más símbolos Unicode que cumple las siguientes condiciones. Señala la afirmación correcta.

- ☐ Todos los identificadores han de comenzar con una letra, el carácter subrayado (_) o el carácter dólar ($).
- ☐ No puede incluir el carácter espacio en blanco.
- ☐ Puede tener cualquier longitud, no hay tamaño máximo.
- ◉ **Todas las anteriores son correctas.**

Además, se desaconseja el uso del símbolo dólar, y el guión bajo prácticamente sólo se utiliza para separar palabras en variables de tipo Constante.

2.3 – Palabras reservadas

Las palabras reservadas, a veces también llamadas palabras clave o keywords, son secuencias de caracteres formadas con letras ASCII cuyo uso se reserva al lenguaje y, por tanto, **no pueden utilizarse para crear identificadores.**

Las palabras reservadas en Java son:

Abstract	continue	for	new	switch
assert	default	goto	package s	ynchronized
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

Hay palabras reservadas que no se utilizan en la actualidad, como es el caso de **`const`** y **`goto`**, que apenas se utilizan en la actual implementación del lenguaje Java. Por otro lado, puede haber otro tipo de palabras o texto en el lenguaje que aunque no sean palabras reservadas tampoco se pueden utilizar para crear identificadores. Es el caso de true y false que, aunque puedan parecer palabras reservadas, porque no se pueden utilizar para ningún otro uso en un programa, técnicamente son **literales booleanos**. Igualmente, **`null`** es considerado un literal, no una palabra reservada.

DESTACADO

Cuando tras haber consultado la documentación de Java aún no tengas seguridad de cómo funciona alguna de sus características, pruébala en tu ordenador, y analiza cada mensaje de error que te dé el compilador para corregirlo. Busca en foros de Internet errores similares para ayudarte de la experiencia de otros usuarios y usuarias.

Normalmente, los editores y entornos integrados de desarrollo utilizan colores para diferenciar las palabras reservadas del resto del código, los comentarios, las constantes y literales, etc. De esta forma se facilita la lectura del programa y la detección de errores de sintaxis. Dependiendo de la configuración del entorno se utilizarán unos colores u otros, es posible *en negro*
Puede que te interese cambiar los colores que utiliza Netbeans para la sintaxis de tus programas, por ejemplo si quieres que los comentarios aparezcan en verde en lugar de en gris, o indicar que tienes la autoría de los mismos, en lugar de que te aparezca el nombre de usuario del sistema operativo.

2.4 - Tipos de variables. Constantes I

En un programa nos podemos encontrar distintos tipos de variables. Las diferencias entre una variable y otra dependerán de varios factores, por ejemplo, el tipo de datos que representan, si su valor cambia o no a lo largo de todo el programa, o cuál es el papel que llevan a cabo en el programa. De esta forma, el lenguaje de programación Java define los siguientes tipos de variables:

a) **Variables de tipos primitivos y variables referencia,** según el tipo de información que contengan. En función de a qué grupo pertenezca la variable, tipos primitivos o tipos referenciados, podrá tomar unos valores u otros, y se podrán definir sobre ella unas operaciones u otras.
b) **Variables y constantes**, dependiendo de si su valor cambia o no durante la ejecución del programa. La definición de cada tipo sería:
   - Variables. Sirven para almacenar los datos durante la ejecución del programa, pueden estar formadas por cualquier tipo de dato primitivo o referencia. Su valor puede cambiar varias veces a lo largo de todo el programa.
   - Constantes o variables finales. Son aquellas variables cuyo valor no cambia a lo largo de todo el programa.
c) **Variables miembro y variables locales**, en función del lugar donde aparezcan en el programa. La definición concreta sería:
   - Variables miembro. Son las variables que se crean dentro de una clase (*Componente software reutilizable expresado en términos de atributos y comportamientos o métodos. Los programadores pueden usar sus propias clases o las incluidas en el lenguaje*), fuera de cualquier método (*Elementos de una clase u objeto compuestos por una serie de sentencias que sirven para describir las acciones a realizar con esa clase u objeto*). Pueden ser de tipos primitivos o referencias, variables o constantes. En un lenguaje puramente orientado a objetos como es Java, todo se basa en la utilización de objetos (*Los objetos se crean a partir*

de clases, y representan casos individuales de una clase), los cuales se crean usando clases. En la siguiente unidad veremos los distintos tipos de variables miembro que se pueden usar.

- Variables locales. Son las variables que se crean y usan dentro de un método o, en general, dentro de cualquier bloque de código. La variable deja de existir cuando la ejecución del bloque de código o el método finaliza. Al igual que las variables miembro, las variables locales también pueden ser de tipos primitivos o referencias.

AUTOEVALUACIÓN:

Relaciona los tipos de variables con la característica correspondiente, escribiendo el número asociado a la característica en el hueco correspondiente.

Ejercicio de relacionar		
Las variables...	**Relación**	**Tienen la característica de que …**
Locales.	2	1. Una vez inicializadas su valor nunca cambia.
Miembro.	3	2. Van dentro de un método.
Constantes.	1	3. Van dentro de una clase.

2.5Tipos de variables. Constantes II

El siguiente ejemplo muestra el código para la creación de varios tipos de variables. Como ya veremos en apartados posteriores, las variables necesitan declararse, a veces dando un valor y otras con un valor por defecto.

```
/**
 * Aplicación ejemplo de tipos de variables
 *
 * @author FMA
 */
public class ejemplovariables {
    final double PI =3.1415926536;  // PI es una constante
    int x;                          // x es una variable miembro
                                    // de clase ejemplovariables

    int obtenerX(int x) {               // x es un parámetro
            int valorantiguo = this.x;  // valorantiguo es una variabe local
            return valorantiguo;
    }

    // el método main comienza la ejecución de la aplicación
    public static void main(String[] args) {
        // aquí iría el código de nuestra aplicación


    } // fin del método main

} // fin de la clase ejemplovariables
```

Este programa crea una clase que contiene las siguientes variables:

- **Variable constante llamada `PI`**: esta variable por haberla declarado como constante no podrá cambiar su valor a lo largo de todo el programa.
- **Variable miembro llamada `x`:** Esta variable pertenece a la clase **`ejemplovariables`**. La variable **`x`** puede almacenar valores del tipo primitivo **`int`**. El valor de esta variable podrá ser modificado en el programa, normalmente por medio de algún otro método que se cree en la clase.
- **Variable local llamada `valorantiguo`:** Esta variable es local porque está creada dentro del método **`obtenerX()`**. Sólo se podrá acceder a ella dentro del método donde está creada, ya que fuera de él no existe.

En apartados posteriores veremos cómo darle más funcionalidad a nuestros programas, mostrar algún resultado por pantalla, hacer operaciones, etc. Por el momento, si ejecutas el ejemplo anterior simplemente mostrará el mensaje "**GENERACIÓN CORRECTA**", indicando que todo ha ido bien y el programa ha finalizado.

Creando nuestro primer programa

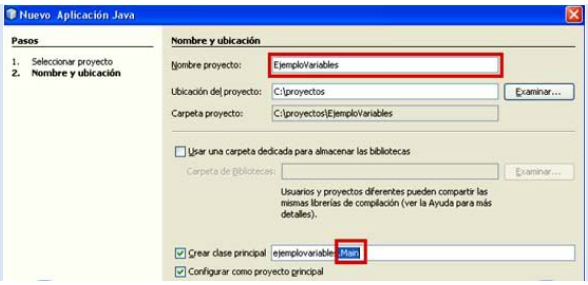
1

2

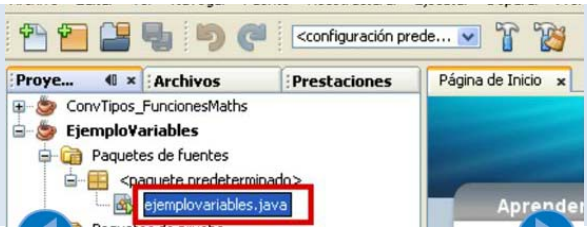
3

Escribimos el nombre del proyecto *EjemploVariables*
Si queremos cambiar la ubicación de los archivos del proyecto seleccionamos *Examinar.*
Como no vamos a utilizar paquetes en nuestra aplicación, eliminamos el texto **.Main** en el cuadro de texto *Crear clase principal*

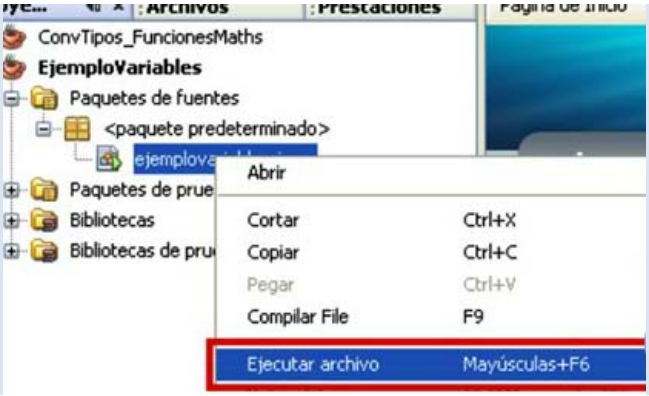
4



5



6


ConvTipos_FuncionesMaths
EjemploVariables
Paquetes de fuentes
<paquete predeterminado>
Paquetes de prue
Bibliotecas
Bibliotecas de pru
Abrir
Cortar Ctrl+X
Copiar Ctrl+C
Pegar Ctrl+V
Compilar File F9
Ejecutar archivo Mayúsculas+F6

3. Los tipos de datos

Caso práctico
María *ya ha hecho sus pinitos como programadora. Ahora mismo está ayudando a Juan con las variables y le ha surgido un problema.*
—El lenguaje se está comportando de una forma extraña, quiero llamar a una variable final y no me deja— Le comenta a ***Juan****.*
—Claro, eso es porque final es una palabra reservada y ya hemos visto que no la puedes utilizar para nombrar variables— le responde ***Juan****.*
—¡Vaya!— exclama ***María****, —¡es verdad!, ¿y qué otros requisitos debo tener en cuenta a la hora de declarar las variables?*
—Pues lo importante es saber qué tipo de información hay que guardar, para poder asignarles el tipo de dato adecuado. ¿Tienes un momento y te lo cuento?— le dice ***Juan***

En los lenguajes fuertemente tipados, a todo dato (*constante, variable o expresión*) le corresponde un tipo que es conocido antes de que se ejecute el programa.

El tipo limita el valor de la variable o expresión, las operaciones que se pueden hacer sobre esos valores, y el significado de esas operaciones. Esto es así porque **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.**

Debido a que el tipo de dato de una variable se conoce durante la revisión que hace el compilador para detectar errores, o sea en tiempo de compilación, esta labor es mucho más fácil, ya que no hay que esperar a que se ejecute el programa para saber qué valores va a contener esa variable. Esto se consigue con un control muy exhaustivo de los tipos de datos que se utilizan, lo cual tiene sus ventajas e inconvenientes, por ejemplo cuando se intenta asignar un valor de un tipo, a una variable de otro tipo. Sin embargo, en Java, puede haber conversión entre ciertos tipos de datos, como veremos posteriormente.

Ahora no es el momento de entrar en detalle sobre la conversión de tipos, pero sí debemos conocer con exactitud de qué tipos de datos dispone el lenguaje Java. Ya hemos visto que las variables, según la información que contienen, se pueden dividir en variables de tipos primitivos y variables referencia. Pero ¿qué es un tipo de dato primitivo? ¿Y un tipo referencia? Esto es lo que vamos a ver a continuación. Los tipos de datos en Java se dividen principalmente en dos categorías:

- **Tipos de datos sencillos o primitivos**. Representan valores simples que vienen predefinidos en el lenguaje; contienen valores únicos, como por ejemplo un carácter o un número.
- **Tipos de datos referencia.** Se definen con un nombre o referencia (puntero) que contiene la dirección en memoria de un valor o grupo de valores. Dentro de este tipo tenemos por ejemplo los vectores o arrays, que son una serie de elementos del mismo tipo, o las clases, que son los modelos o plantillas a partir de los cuales se crean los objetos.

En el siguiente apartado vamos a ver con detalle los diferentes tipos de datos que se engloban dentro de estas dos categorías.

AUTOEVALUACIÓN:
El tipado fuerte de datos supone que:

- A todo dato le corresponde un tipo que es conocido antes de que se ejecute el programa.
- El lenguaje hace un control muy exhaustivo de los tipos de datos.

☐ El compilador puede optimizar mejor el tratamiento de los tipos de datos.

☉ **Todas las anteriores son correctas.**

En un lenguaje fuertemente tipado se cumplen todas las condiciones anteriores.

3.1 Tipos de datos primitivos I

Los tipos primitivos son aquéllos datos sencillos que constituyen los tipos de información más habituales: números, caracteres y valores lógicos o booleanos. Al contrario que en otros lenguajes de programación orientados a objetos, **en Java no son objetos**.

Una de las mayores ventajas de tratar con tipos primitivos en lugar de con objetos, es que el compilador de Java puede optimizar mejor su uso. Otra importante característica, es que cada uno de los tipos primitivos tiene **idéntico** tamaño y comportamiento en todas las versiones de Java y para cualquier tipo de ordenador. Esto quiere decir que no debemos preocuparnos de cómo se representarán los datos en distintas plataformas, y asegura la **portabilidad** de los programas, a diferencia de lo que ocurre con otros lenguajes. Por ejemplo, el tipo **`int`** siempre se representará con 32 bits, con signo, y en el formato de representación complemento a 2, en cualquier plataforma que soporte Java.

TIPOS DE DATOS PRIMITIVOS

Tipo	Descripción	Bytes	Rango	Valor por default
byte	Entero muy corto	1	-128 a 127	0
short	Entero corto	2	-32,768 a 32,767	0
int	Entero	4	-2,147,483,648 a 2,147,483,647	0
long	Entero largo	8	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	0L
float	Numero con punto flotante de precisión individual con hasta 7 dígitos significativos	4	+/-1.4E-45 (+/-1.4 times 10^{-45}) a +/-3.4E38 (+/-3.4 times 10^{38})	0.0f
double	Numero con punto flotante de precisión doble con hasta 16 dígitos significativos	8	+/-4.9E-324 (+/-4.9 times 10^{-324}) a +/-1.7E308 (+/-1.7 times 10^{308})	0.0d
char	Carácter Unicode 2	\u0000 a \uFFFF	'\u0000'	
boolean	Valor Verdadero o Falso	1	true o false	false

```
public class Example {
   public static void main(String[] args) {
      //declarar variables
      byte mes = 12;
      int contador = 0;
      double pi = 3.14159265358979323846264338327795;
      float interes = 4.25e2F;
      char letra = 'Z';
      boolean encontrado = true;
      //imprimir valores
      System.out.println(mes); //imprimirá 12
      System.out.println(contador); //imprimirá 0
      System.out.println(pi); //imprimirá 3.141592653589793
      System.out.println(interes); //imprimirá 425.0
```

```
        System.out.println(letra); //imprimirá Z
        System.out.println(encontrado); //imprimirá true
    }
}
```

¿SABÍAS QUÉ?.
Java especifica el tamaño y formato de todos los tipos de datos primitivos con independencia de la plataforma o sistema operativo donde se ejecute el programa, de forma que el programador no tiene que preocuparse sobre las dependencias del sistema, y no es necesario escribir versiones distintas del programa para cada plataforma.

Hay una peculiaridad en los tipos de datos primitivos, y es que el tipo de dato **`char`** es considerado por el compilador como un tipo numérico, ya que los valores que guarda son el código Unicode correspondiente al carácter que representa, no el carácter en sí, por lo que puede operarse con caracteres como si se tratara de números enteros.

Por otra parte, a la hora de elegir el tipo de dato que vamos a utilizar ¿qué criterio seguiremos para elegir un tipo de dato u otro? Pues deberemos tener en cuenta cómo es la información que hay que guardar, si es de tipo texto, numérico, … y, además, qué rango de valores puede alcanzar. En este sentido, hay veces que aunque queramos representar un número sin decimales, tendremos que utilizar datos de tipo real.

Por ejemplo, el tipo de dato **`int`** no podría representar la población mundial del planeta, ya que el valor máximo que alcanza es de 2.147.483.647, siendo éste el número máximo de combinaciones posibles con 32 bits, teniendo en cuenta que la representación de los números enteros en Java utiliza complemento a 2. Si queremos representar el valor correspondiente a la población mundial del planeta, cerca de 7 mil millones de habitantes, tendríamos que utilizar al menos un tipo de dato **`long`**, o si no tenemos problemas de espacio un tipo **`float`**.

Sin embargo, los tipos reales tienen otro problema: la **precisión**. Vamos a ver más sobre ellos en el siguiente apartado.

Si quieres obtener información sobre cómo se lleva a cabo la representación interna de números enteros y sobre la aritmética binaria, puedes consultar el siguiente enlace:
http://platea.pntic.mec.es/~lgonzale/tic/binarios/aritmetica.html

3.2Tipos de datos primitivos II

El tipo de dato real permite representar cualquier número con decimales. Al igual que ocurre con los enteros, la mayoría de los lenguajes definen más de un tipo de dato real, en función del número de bits usado para representarlos. Cuanto mayor sea ese número:

- Más grande podrá ser el número real representado en valor absoluto
- Mayor será la precisión de la parte decimal

Entre cada dos números reales cualesquiera, siempre tendremos infinitos números reales, por lo que la mayoría de ellos los representaremos de forma aproximada. Por ejemplo, en la aritmética convencional, cuando dividimos 10 entre 3, el resultado es 3.3333333…, con la secuencia de 3

repitiéndose infinitamente. En el ordenador sólo podemos almacenar un número finito de bits, por lo que el almacenamiento de un número real será siempre una aproximación.

Los números reales se representan en coma flotante, que consiste en trasladar la coma decimal a la primera cifra significativa del valor, con objeto de poder representar el máximo de números posible.

Un número se expresa como: $Valor = mantisa \times 2^{exponente}$

En concreto, sólo se almacena la mantisa y el exponente al que va elevada la base. Los bits empleados por la mantisa representan la precisión del número real, es decir, el número de cifras decimales significativas que puede tener el número real, mientras que los bits del exponente expresan la diferencia entre el mayor y el menor número representable, lo que viene a ser el intervalo de representación.

En Java las variables de tipo **`float`** se emplean para representar los números en coma flotante de simple precisión de 32 bits, de los cuales 24 son para la mantisa y 8 para el exponente. La mantisa es un valor entre -1.0y 1.0 y el exponente representa la potencia de 2 necesaria para obtener el valor que se quiere representar. Por su parte, las variables tipo **`double`** representan los números en coma flotante de doble precisión de 64 bits, de los cuales 53 son para la mantisa y 11 para el exponente.

La mayoría de los programadores en Java emplean el tipo `double` cuando trabajan con datos de tipo real, es una forma de asegurarse de que los errores cometidos por las sucesivas aproximaciones sean menores. De hecho, Java considera los valores en coma flotante como de tipo `double` por defecto.

Con el objetivo de conseguir la máxima portabilidad de los programas, Java utiliza el estándar internacional IEEE 754 para la representación interna de los números en coma flotante, que es una forma de asegurarse de que el resultado de los cálculos sea el mismo para diferentes plataformas.

PARA SABER MÁS

La siguiente página es la web oficial sobre el estándar internacional IEEE 754-2008 para representación de números en coma flotante (en inglés):

http://grouper.ieee.org/groups/754/

AUTOEVALUACIÓN:

Relaciona los tipos primitivos con los bits y rango de valores correspondientes, escribiendo el número asociado en el hueco correspondiente.

Ejercicio de relacionar

Tipo	Relación	Característica
short	3	Coma flotante de 64 bits, usando la representación IEE754-2008
byte	5	Entero de 32 bits, rango de valores de -2.147.483.648 (-2^{31}) a 2.147.483.647 ($+2^{31}-1$)
double	1	Entero de 16 bits, rango de valores de -32.768 (-2^{15}) a +32.767 ($+2^{15}-1$)
long	6	Coma flotante de 32 bits, usando la representación IEEE 745-2008
int	2	Entero de 8 bits, rango de valores de -128 (-2^{7}) a +127 ($+2^{7}-1$)
float	4	Entero de 64 bits, rango de valores de -9.223.372.036.854.775.808 (-263) a 9.223.372.036.854.775.807 (+263-1)

Además de los anteriores, también son tipos primitivos el tipo de dato boolean, con valores true y false, y el tipo de datos char, que almacena el código Unicode de un carácter.

3.3 Declaración e inicialización

Llegados a este punto cabe preguntarnos ¿cómo se crean las variables en un programa? ¿Qué debo hacer antes de usar una variable en mi programa? Pues bien, como podrás imaginar, debemos crear las variables antes de poder utilizarlas en nuestros programas, indicando qué nombre va a tener y qué tipo de información va a almacenar, en definitiva, debemos **declarar la variable**.

Las variables se pueden declarar en cualquier bloque de código, dentro de llaves. Y lo hacemos indicando su identificador y el tipo de dato, separadas por comas si vamos a declarar varias a la vez, por ejemplo:

```
int numAlumnos = 15;
double radio = 3.14, importe = 102.95;
```

De esta forma, estamos declarando **numAlumnos** como una variable de tipo **int**, y otras dos variables **radio** e **importe** de tipo **double**. Aunque no es obligatorio, hemos aprovechado la declaración de las variables para inicializarlas a los valores 15, 3.14 y 102.95 respectivamente.

Si la variable va a permanecer inalterable a lo largo del programa, la declararemos como **constante**, utilizando la palabra reservada **final** de la siguiente forma:

```
final double PI = 3.1415926536;
```

En ocasiones puede que al declarar una variable no le demos valor, ¿qué crees que ocurre en estos casos? Pues que el compilador le asigna un valor por defecto, aunque depende del tipo de variable que se trate:

- Las **variables miembro** sí se inicializan automáticamente, si no les damos un valor. Cuando son de tipo numérico, se inicializan por defecto a 0, si don de tipo carácter, se inicializan al carácter **null** (\0), si son de tipo boolean se les asigna el valor por defecto **false**, y si son tipo referenciado se inicializan a **null**.
- Las **variables locales** no se inicializan automáticamente. Debemos asignarles nosotros un valor antes de ser usadas, ya que si el compilador detecta que la variable se usa antes de que se le asigne un valor, produce un error. Por ejemplo en este caso:

```
int p;
int q = p; // error
```

Y también en este otro, ya que se intenta usar una variable local que podría no haberse inicializado:

```
int p;
if (. . . )
p = 5 ;
int q = p; // error
```

En el ejemplo anterior la instrucción if hace que si se cumple la condición que hay entre paréntesis (cualquiera que indiquemos), entonces el programa asignará el valor 5 a la variable p; sino se cumple la condición, p quedará sin inicializar. Pero si p no se ha inicializado, no tendría valor para asignárselo a q.

Por ello, el compilador detecta ese posible problema y produce un error del tipo **"La variable podría no haber sido inicializada"**, independientemente de si se cumple o no la condición del **if**.

AUTOEVALUACIÓN:
De las siguientes, señala cuál es la afirmación correcta:

- La declaración de una variable consiste básicamente en indicar el tipo que va a tener seguido del nombre y su valor.
- Java no tiene restricción de tipos.
- Todos los tipos tienen las mismas operaciones a realizar con ellos: suma, resta, multiplicación, etc.
- **Todas las anteriores son incorrectas.**

Nada de lo afirmado sobre declaración de variables, tipado de datos y operaciones es correcto.

3.4 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.

Como comentábamos al principio del apartado de Tipos de datos, cualquier aplicación de hoy en día necesita no perder de vista una cierta cantidad de datos. Cuando el conjunto de datos utilizado 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 descrito, 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";
```

Hemos visto qué son las variables, cómo se declaran y los tipos de datos que pueden adoptar. Anteriormente hemos visto un ejemplo de creación de variables, en esta ocasión vamos a crear más

```
public class ejemplotipos {

    // el método main inicia la ejecución de la aplicación
    public static void main(String[] args) {
        // Código de la aplicación
        int i = 10;
        double d = 3.14;
        char c1 = 'a';
        char c2 = 65;
        boolean encontrado = true;
        String msj = "Bienvenido a Java";

        System.out.println("La variable i es de tipo entero y su valor es: " + i);
        System.out.println("La variable f es de tipo double y su valor es: "+d);
        System.out.println("La variable c1 es de tipo carácter y su valor es: "+c1);
        System.out.println("La variable c2 es de tipo carácter y su valor es: "+c2);
        System.out.println("La variable encontrado es de tipo booleano y su valor es: "+encontrado);
        System.out.println("La variable msj es de tipo String y su valor es: " + msj);
    } // fin del método main

} // fin de la clase ejemplotipos
```

variables, pero de distintos tipos primitivos y los vamos a mostrar por pantalla. Los tipos referenciados los veremos en la siguiente unidad.

Para mostrar por pantalla un mensaje utilizamos **`System.out`**, conocido como la salida estándar del programa. Este método lo que hace es escribir un conjunto de caracteres a través de la línea de comandos. En Netbeans esta línea de comandos aparece en la parte inferior de la pantalla. Podemos utilizar **`System.out.print`** o **`System.out.println`**. En el segundo caso lo que hace el método es que justo después de escribir el mensaje, sitúa el cursor al principio de la línea siguiente. El texto en color gris que aparece entre caracteres `//` son comentarios que permiten documentar el código, pero no son tenidos en cuenta por el compilador y, por tanto, **no afectan a la ejecución del programa**.

3.5 Tipos 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. Es como si definiéramos nuestro propio tipo de datos.

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.

CITAS PARA PENSAR

Oigo y olvido. Veo y recuerdo. Hago y comprendo. Proverbio chino.

En el siguiente ejemplo puedes comprobar el uso que se hace de los tipos de datos enumerados. Tenemos una variable **`Dias`** que almacena los días de la semana. Para acceder a cada elemento del tipo enumerado se utiliza el nombre de la variable seguido de un punto y el valor en la lista. Más tarde veremos que podemos añadir métodos y campos o variables en la declaración del tipo enumerado, ya que como hemos comentado un tipo enumerado en Java tiene el mismo tratamiento que las clases.

En este ejemplo hemos utilizado el método **`System.out.print`**. Como podrás comprobar si lo ejecutas, la instrucción número 18 escribe el texto que tiene entre comillas pero no salta a la siguiente línea, por lo que la instrucción

```
10  public class tiposenumerados {
11      public enum Dias {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
12
13      public static void main(String[] args) {
14          // codigo de la aplicacion
15          Dias diaactual = Dias.Martes;
16          Dias diasiguiente = Dias.Miercoles;
17
18          System.out.print("Hoy es: ");
19          System.out.println(diaactual);
20          System.out.println("Mañana\nes\n"+diasiguiente);
21
22      } // fin main
23
24  } // fin tiposenumerados
```

número 19 escribe justo a continuación.

Sin embargo, también podemos escribir varias líneas usando una única sentencia. Así lo hacemos en la instrucción número 20, la cual imprime como resultado tres líneas de texto. Para ello hemos utilizado un carácter especial, llamado **carácter escape** (\). Este carácter sirve para darle ciertas órdenes al compilador, en lugar de que salga impreso en pantalla. Después del carácter de escape viene otro carácter que indica la orden a realizar, juntos reciben el nombre de **secuencia de escape**. La secuencia de escape **\n** recibe el nombre de **carácter de nueva línea**. Cada vez que el compilador se encuentra en un texto ese carácter, el resultado es que mueve el cursor al principio de la línea siguiente. En el próximo apartado vamos a ver algunas de las secuencias de escape más utilizadas.

4. Literales de los tipos primitivos

CASO
***Ada** se encuentra con **María** y **Juan**.*
*—¿Cómo van esos avances con Java?— **Juan** sabe lo que significa eso, **Ada** se interesa por el trabajo que están llevando a cabo. Ya tienen claro qué tipos de datos utilizar, pero necesitan cerciorarse de los valores que pueden almacenar esos tipos de datos, es decir, qué literales pueden contener, para estar seguros que han hecho la elección adecuada.*
*—Muy bien— contesta **Juan**. —Si quieres te hacemos una demostración para que veas la estructura del programa.*
*A **Ada** le satisface la eficacia con que trabajan **María** y **Juan**, apenas ha comenzado con el proyecto y pronto podrá ver resultados inmediatos.*

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`. 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`').

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

Normalmente, los objetos en Java deben ser creados con la orden new. Sin embargo, los literales ***`String`*** *no lo necesitan ya que son objetos que se crean implícitamente por Java.*

Los **literales de cadenas de caracteres** se indican entre comillas dobles. En el ejemplo anterior "El primer programa" es un literal de tipo cadena de caracteres. 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…\"";
```

En el ejemplo anterior de tipos enumerados ya estábamos utilizando secuencias de escape, para introducir un salto de línea en una cadena de caracteres, utilizando el carácter especial **`\n`**.

5. Operadores y expresiones.

CASO
María *y* ***Juan*** *tienen definidas las variables y tipos de datos a utilizar en la aplicación. Es el momento de ponerse a realizar los cálculos que permitan manipular esos datos, sumar, restar, multiplicar, dividir y mucho más. En definitiva se trata de llevar los conocimientos matemáticos al terreno de la programación, ver cómo se pueden hacer operaciones aritméticas, lógicas o de comparación en el lenguaje Java.*
También necesitarán algún operador que permita evaluar una condición y decidir las acciones a realizar en cada caso. Es importante conocer bien cómo el lenguaje evalúa esas expresiones, en definitiva, cuál es la precedencia que tiene cada operador.

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:

expresion

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

sentencia = instruccion

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 en unidades posteriores.

int x = 1; // asignamos el valor 1 a la variable x

Como curiosidad comentar que las expresiones de asignación, al poder ser usadas como parte de otras asignaciones u operaciones, son consideradas tanto expresiones en sí mismas como sentencias.

CITAS PARA PENSAR

Lo que no hemos realizado no es más que lo que todavía no hemos intentado hacer. Alexis de Tocqueville.

5.1 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	

```
10  public class operadoresaritmeticos {
11      public static void main(String[] args) {
12          short x = 7;
13          int y = 5;
14          float f1 = 13.5f;
15          float f2 = 8f;
16          System.out.println("El valor de x es " + x +" y el valor de y es " +y);
17          System.out.println("El resultado de x + y es " + (x + y));
18          System.out.println("El resultado de x - y es " + (x - y));
19          System.out.printf("%s\n%s%s\n","División entera:","x / y = ",(x/y));
20          System.out.println("Resto de la división entera: x % y = " + (x % y));
21          System.out.printf("El valor de f1 es %f y el de f2 es %f\n",f1,f2);
22          System.out.println("El resultado de f1 / f2 es " + (f1 / f2));
23      } // fin de main
24  } // fin de la clase operadoresaritmeticos
```

En el ejemplo vemos un programa básico que utiliza operadores aritméticos. Observa que usamos **`System.out.printf`** para mostrar por pantalla un texto formateado. El texto entre dobles comillas son los argumentos del método **`printf`** y si usamos más de uno, se separan con comas. Primero indicamos cómo queremos que salga el texto, y después el texto que queremos mostrar. Fíjate que con el primer **`%s`** nos estamos refiriendo a una variable de tipo **`String`**, o sea, a la primera cadena de texto, con el siguiente **`%s`** a la segunda cadena y con el último **`%s`** a la tercera. Con **`%f`** nos referimos a un argumento de tipo **`float`**, etc.

5.2 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

Un ejemplo de operadores de asignación combinados lo tenemos a continuación:

```
public class operadoresasignacion {

    // clase principal main que inicia la aplicación
    public static void main(String[] args) {
        int x;
        int y;
        x = 5;  // operador asignación
        y = 3;  // operador asignación

        //operadores de asignación combinados
        System.out.printf("El valor de x es %d y el valor de y es %d\n", x,y);
        x += y;
        // podemos utilizar indistintamente printf o println
        System.out.println(" Suma combinada: x += y " + " ..... x vale " + x);
        x = 5;
        x -= y;
        System.out.println(" Resta combinada: x -= y " + " ..... x vale " + x);
        x = 5;
        x *= y;
        System.out.println(" Producto combinado: x *= y " + " ..... x vale " + x);
        x = 5;
        x /= y;
        System.out.println(" Division combinada: x /= y " + " ..... x vale " + x);
        x = 5;
        x %= y;
        System.out.println(" Resto combinada: x %= y " + " ..... x vale " + x);
    } // fin main
} // fin operadoresasignacion
```

Para saber más

En el siguiente enlace tienes información interesante sobre cómo se pueden utilizar los caracteres especiales incluidos en la orden printf (en inglés):

http://www.java2s.com/Tutorial/Java/0120__Development/UsingJavasprintfMethod.htm

5.3 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
true false

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 en la siguiente unidad de Estructuras de control.

CITAS PARA PENSAR.

La buena escritura debe ser concisa. Una oración no debe contener palabras innecesarias, un párrafo no debe contener oraciones innecesarias.

William Strunk, Jr.

5.4 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.

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

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

Hasta ahora hemos visto ejemplos que creaban variables y se inicializaban con algún valor. Pero ¿y si lo que queremos es que el usuario introduzca un valor al programa?
Entonces debemos agregarle interactividad a nuestro programa, por ejemplo utilizando la clase **`Scanner`**.

Aunque no hemos visto todavía qué son las clases y los objetos, de momento vamos a pensar que la clase **`Scanner`** nos va a permitir leer los datos que se escriben por teclado, y que para usarla es necesario importar el paquete de clases que la contiene. El código del ejemplo lo tienes a continuación. El programa se quedará esperando a que el usuario escriba algo en el teclado y pulse la tecla intro. En ese momento se convierte lo leído a un valor del tipo int y lo guarda en la variable indicada. Además de los operadores relacionales, en este ejemplo utilizamos también el operador condicional, que compara si los números son iguales. Si lo son, devuelve la cadena iguales y sino, la cadena distintos.

```
public class ejemplorelacionales {
   // método principal que inicia la aplicación
   public static void main( String args[] )
   {
      // clase Scanner para petición de datos
      Scanner teclado = new Scanner( System.in );
      int x, y;
      String cadena;
      boolean resultado;
      System.out.print( "Introducir primer número: " );
      x = teclado.nextInt(); // pedimos el primer número al usuario
      System.out.print( "Introducir segundo número: " );
      y = teclado.nextInt(); // pedimos el segundo número al usuario
      // realizamos las comparaciones
      cadena=(x==y)?"iguales":"distintos";
      System.out.printf("Los números %d y %d son %s\n",x,y,cadena);
      resultado=(x!=y);
      System.out.println("x != y  // es " + resultado);
      resultado=(x < y );
      System.out.println("x < y  // es " + resultado);
      resultado=(x > y );
      System.out.println("x > y  // es " + resultado);
      resultado=(x <= y );
      System.out.println("x <= y  // es " + resultado);
      resultado=(x >= y );
      System.out.println("x >= y  // es " + resultado);
   } // fin método main
} // fin clase ejemplorelacionales
```

AUTOEVALUACIÓN:

Señala cuáles son los operadores relacionales en Java.

- ☐ = =, ! =, >, <, > =, < =.
- ☐ ==, =!, >, <, =>, =<.
- ☐ =, !=, >, <, >=, <=.
- ☑ ==, !=, >, <, >=, <=.

Tienes claro cuáles son los operadores relacionales en Java.

5.5 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

En el siguiente código puedes ver un ejemplo de utilización de operadores lógicos:

```
public class operadoreslogicos {
  public static void main(String[] args) {
    System.out.println("OPERADORES LÓGICOS");

    System.out.println("Negacion:\n ! false es : " + (! false));
    System.out.println(" ! true es : " + (! true));

    System.out.println("Operador AND (&):\n false & false es : " + (false & false));
    System.out.println(" false & true es : " + (false & true));
    System.out.println(" true & false es : " + (true & false));
    System.out.println(" true & true es : " + (true & true));

    System.out.println("Operador OR (|):\n false | false es : " + (false | false));
    System.out.println(" false | true es : " + (false | true));
    System.out.println(" true | false es : " + (true | false));
    System.out.println(" true | true es : " + (true | true));

    System.out.println("Operador OR Exclusivo (^):\n false ^ false es : " + (false ^ false));
    System.out.println(" false ^ true es : " + (false ^ true));
    System.out.println(" true ^ false es : " + (true ^ false));
    System.out.println(" true ^ true es : " + (true ^ true));

    System.out.println("Operador &&:\n false && false es : " + (false && false));
    System.out.println(" false && true es : " + (false && true));
    System.out.println(" true && false es : " + (true && false));
    System.out.println(" true && true es : " + (true && true));

    System.out.println("Operador ||:\n false || false es : " + (false || false));
    System.out.println(" false || true es : " + (false || true));
    System.out.println(" true || false es : " + (true || false));
    System.out.println(" true || true es : " + (true || true));
  } // fin main
} // fin operadoreslogicos
```

5.6 Operadores de bits

Los operadores a nivel de bits se caracterizan porque realizan operaciones sobre números enteros (o **char**) en su representación binaria, es decir, sobre cada dígito binario.

En la tabla tienes los operadores a nivel de bits que utiliza Java.

Operador	Ejemplo en Java	Significado
~	~op	Realiza el complemento binario de op (invierte el valor de cada bit)
&	op1 & op2	Realiza la operación AND binaria sobre op1 y op2
|	op1 | op2	Realiza la operación OR binaria sobre op1 y op2
^	op1 ^ op2	Realiza la operación OR-exclusivo (XOR) binaria sobre op1 y op2
<<	op1 << op2	Desplaza op2 veces hacia la izquierda los bits de op1
>>	op1 >> op2	Desplaza op2 veces hacia la derecha los bits de op1
>>>	op1 >>> op2	Desplaza op2 (en positivo) veces hacia la derecha los bits de op1

PARA SABER MÁS

Los operadores de bits raramente los vas a utilizar en tus aplicaciones de gestión. No obstante, si sientes curiosidad sobre su funcionamiento, puedes ver el siguiente enlace dedicado a este tipo de operadores:

http://www.zator.com/Cpp/E4_9_3.htm

5.7 Trabajo con cadenas

Ya hemos visto en el apartado de literales que el objeto **`String`** se corresponde con una secuencia de caracteres entrecomillados, como por ejemplo "hola". Este literal se puede utilizar en Java como si de un tipo de datos primitivo se tratase, y, como caso especial, no necesita la orden **`new`** para ser creado.

No se trata aquí de que nos adentremos en lo que es una clase u objeto, puesto que lo veremos en unidades posteriores, y trabajaremos mucho sobre ello. Aquí sólo vamos a utilizar determinadas operaciones que podemos realizar con el objeto **`String`**, y lo verás mucho más claro con ejemplos descriptivos.

Para aplicar una operación a una variable de tipo **`String`**, escribiremos su nombre seguido de la operación, separados por un punto. Entre las principales operaciones que podemos utilizar para trabajar con cadenas de caracteres están las siguientes:

- **Creación**. Como hemos visto en el apartado de literales, podemos crear una variable de tipo **`String`** simplemente asignándole una cadena de caracteres encerrada entre comillas dobles.
- **Obtención de longitud**. Si necesitamos saber la longitud de un **`String`**, utilizaremos el método **`length()`**.
- **Concatenación**. Se utiliza el operador **`+`** o el método **`concat()`** para concatenar cadenas de caracteres.
- **Comparación**. El método **`equals()`** nos devuelve un valor booleano que indica si las cadenas comparadas son o no iguales. El método **`equalsIgnoreCase()`** hace lo propio, ignorando las mayúsculas de las cadenas a considerar.
- **Obtención de subcadenas**. Podemos obtener cadenas derivadas de una cadena original con el método **`substring()`**, al cual le debemos indicar el inicio y el fin de la subcadena a obtener.
- **Cambio a mayúsculas/minúsculas**. Los métodos **`toUpperCase()`** y **`toLowerCase()`** devuelven una nueva variable que transforma en mayúsculas o minúsculas, respectivamente, la variable inicial.
- **Valueof**. Utilizaremos este método para convertir un tipo de dato primitivo (**`int`**, **`long`**, **`float`**, etc.) a una variable de tipo **`String`**.

A continuación varios ejemplos de las distintas operaciones que podemos realizar con cadenas de caracteres o **`String`** en Java:

```
public class ejemplocadenas {
    public static void main(String[] args)
    {
          String cad1 = "CICLO DAM";
          String cad2 = "ciclo dam";

          System.out.printf( "La cadena cad1 es: %s y cad2 es: %s", cad1,cad2 );

          System.out.printf( "\nLongitud de cad1: %d", cad1.length() );

          // concatenación de cadenas (concat o bien operador +)
          System.out.printf( "\nConcatenación: %s", cad1.concat(cad2) );

          //comparación de cadenas
          System.out.printf("\ncad1.equals(cad2) es %b", cad1.equals(cad2) );
          System.out.printf("\ncad1.equalsIgnoreCase(cad2) es %b", cad1.equalsIgnoreCase(cad2) );
          System.out.printf("\ncad1.compareTo(cad2) es %d", cad1.compareTo(cad2) );

          //obtención de subcadenas
          System.out.printf("\ncad1.substring(0,5) es %s", cad1.substring(0,5) );

          //pasar a minúsculas
          System.out.printf("\ncad1.toLowerCase() es %s", cad1.toLowerCase() );

          System.out.println();
    } // fin main

} // fin ejemplocadenas
```

5.8 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 )`** (1) **`* 5`** (2), 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

REFLEXIONA.

¿Crees que es una buena práctica de programación utilizar paréntesis en expresiones aritméticas complejas, aún cuando no sean necesarios?

El uso de paréntesis, incluso cuando no son necesarios, puede hacer más fácil de leer las expresiones aritméticas complejas.

6. Conversión de tipo.

CASO
María *ha avanzado mucho en sus conocimientos sobre Java y ha contado con mucha ayuda por parte de* ***Juan.*** *Ahora mismo tiene un problema con el código, y le comenta —Estoy atrancada en el código. Tengo una variable de tipo byte y quiero asignarle un valor de tipo int, pero el compilador me da un error de posible pérdida de precisión ¿tú sabes qué significa eso?. —Claro —le contesta Juan, — es un problema de conversión de tipos, para asignarle el valor a la variable de tipo byte debes hacer un casting. —¡Ah! —dice María, —¿y cómo se hace eso?*

Imagina que queremos dividir un número entre otro ¿tendrá decimales el resultado de esa división? Podemos pensar que siempre que el denominador no sea divisible entre el divisor, tendremos un resultado con decimales, pero no es así. Si el denominador 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, por ende, tenga decimales. Existen dos tipos de conversiones:

Las operaciones aritmeticas con bytes se hacen de forma binaria

```
byte numero1b = 100;
byte numero2b = 125;
byte sumab = (byte) (numero1b + numero2b);
System.out.println("\u001B[35mSuma : " + sumab);
Suma : -31
```

- **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**.

6.1 - 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.

6.1.1 - 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 en base a las cuales se realiza 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**.

6.1.2 - 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.

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

6.1.3 - 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'
```

6.1.4 - 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);
```

7. Comentarios.

CASO
***Juan** ha podido comprobar los avances que ha hecho **María** con la programación. Ya domina todos los aspectos básicos sobre sintaxis, estructura de un programa, variables y tipos de datos. **Ada** le acaba de comunicar que van a sumarse al proyecto dos personas más, **Ana** y **Carlos** que están haciendo las prácticas del ciclo de Desarrollo de Aplicaciones Multiplataforma en la empresa. —Al principio de cada programa indicaremos una breve descripción y el autor. En operaciones complicadas podríamos añadir un comentario les ayudará a entender mejor qué es lo que hace —indica Juan. —De acuerdo —comenta **María**, —y podemos ir metiendo los comentarios de la herramienta esa que me comentaste, **Javadoc**, para que se cree una documentación aún más completa. — ¡Ajá! —asiente **Juan**, —pues ¡manos a la obra!*

Los comentarios son muy importantes a la hora de describir qué hace un determinado programa. A lo largo de la unidad los hemos utilizado para documentar los ejemplos y mejorar la comprensión del código. Para lograr ese objetivo, es normal que cada programa comience con unas líneas de comentario que indiquen, al menos, una breve descripción del programa, el autor del mismo y la última fecha en que se ha modificado.

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
*/
```

RECOMENDACIÓN
Una buena práctica de programación es añadir en la última llave que delimita cada bloque de código, un comentario indicando a qué clase o método pertenece esa llave.

PARA SABER MÁS

Si quieres ir familiarizándote con la información que hay en la web de Oracle, en el siguiente enlace puedes encontrar más información sobre la herramienta Javadoc incluida en el Kit de Desarrollo de Java SE (en inglés):

http://download.oracle.com/javase/6/docs/technotes/guides/javadoc/index.html

https://docs.oracle.com/javase/10/javadoc/toc.htm