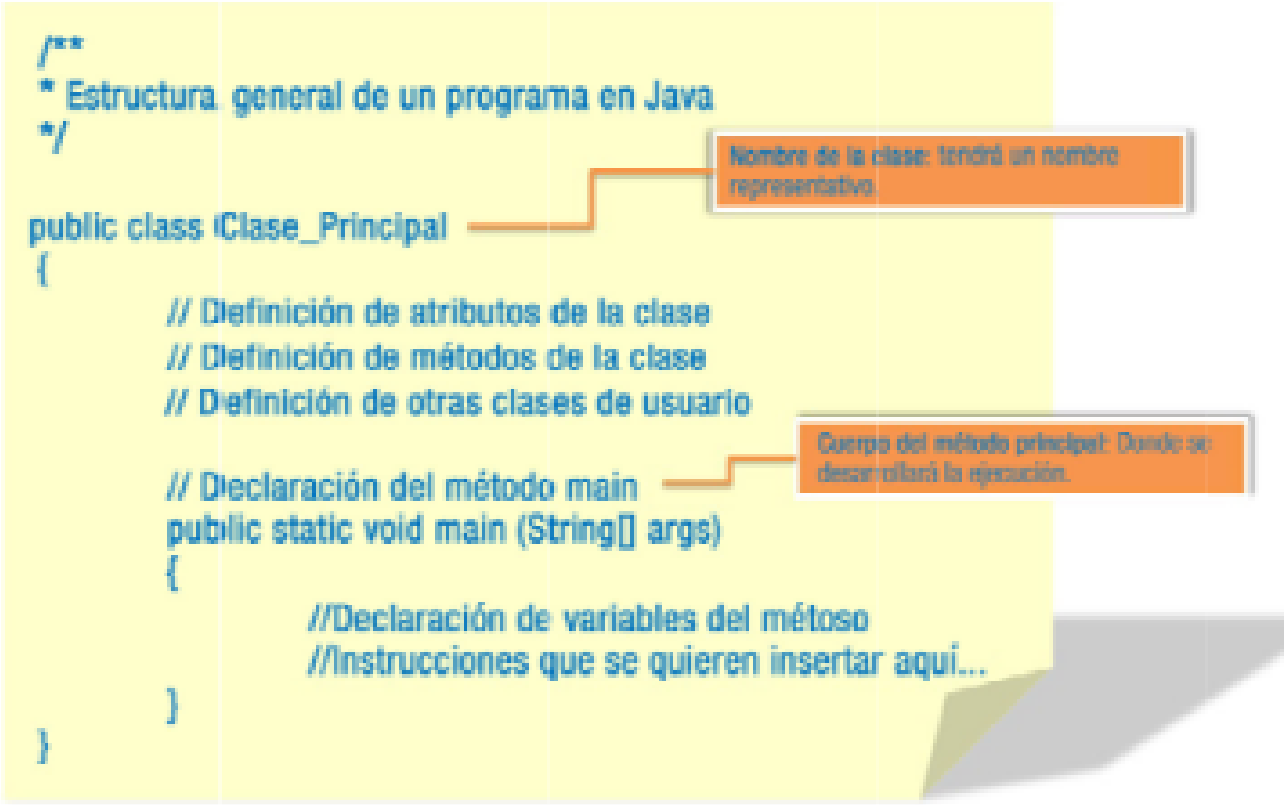
Sumario

UT 03: Elementos de un programa informático....2
1 Estructura de un programa en Java....2
2 Identificadores....3
3 Palabras reservadas....5
4 Variables. Declaración. Inicialización y utilización....5
4.1 Tipos de variables....5
4.2 Constantes....6
4.3 Declaración e inicialización....7
5 Tipos de datos....8
5.1 Tipos de datos primitivos....8
5.2 Tipos referenciados....10
5.3 Tipos enumerados....11
6 Literales....12
7 Operadores y expresiones. Precedencia de operadores....13
7.1 Operadores aritméticos....14
7.2 Operadores de asignación....15
7.3 Operador condicional....16
7.4 Operadores de relación....16
7.5 Operadores lógicos....18
7.6 Operadores de bits....19
7.7 Trabajo con cadenas....19
7.8 Precedencia de operadores....20
8 Conversiones de tipo. Implícitas y explícitas (casting)....22
9 Comentarios....25

UT 03: Elementos de un programa informático

1 Estructura de un programa en Java

El siguiente gráfico representa la estructura de un programa escrito en Java:



Analicemos sus componentes:

- **public class Clase_Principal:** Todos los programas han de incluir una clase como esta. En ella se incluyen los demás elementos del programa, como el método o función "main()". Esta clase puede contener a su vez otra as clases del usuario, pero solo una puede ser public. El nombre del fichero .Java que contiene el código fuente de nuestro programa, coincidirá con el nombre de la clase que estamos describiendo en estas líneas. Hay que tener en cuenta que Java distingue entre mayúsculas y minúsculas a la hora de nombrar el fichero y la clase principal.
- **public static void main (String[] args):** Es el método que representa al programa

principal. En él se podrán incluir las instrucciones oportunas para la ejecución del programa. Desde él se podrá hacer uso del resto de clases creadas. Todos los programas Java tienen un método main.

- **Comentarios:** Los comentarios se suelen incluir en el código fuente para realizar aclaraciones, anotaciones o cualquier otra indicación que el programador estime oportuna. Estos comentarios pueden introducirse de dos formas, con // y con /*
  - Con */ establecemos el inicio de un comentario que acabará con /*.
  - Con // indicamos que el comentario es una línea completa.
- **Bloques de código:** son conjuntos de instrucciones que se marcan mediante la apertura y cierre de llaves { } . El código así marcado es considerado interno al bloque.
- **Punto y coma:** cada línea de código ha de terminar con punto y coma (;). En caso de no hacerlo, tendremos errores sintácticos.

2 Identificadores

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, es decir, la "etiqueta" 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.
- El rango de valores que puede admitir dicha variable.

El nombre dado a la variable se 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.

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 ($).

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: UTF8, UTF16 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 entero 164. 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.

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.

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 el valor booleano (true o false) ni el valor nulo (null).
- Los identificadores deben ser lo más descriptivos posibles. Es mejor usar palabras completas que abreviaturas crípticas. El código será más legible y comprensible, y además será autodocumentado. Por ejemplo, si neceitamos una variable que almacene datos de un cliente es recomendable llamarla FicheroClientes y no Cl33.

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`

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. Palabras reservadas en Java:

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 const y goto. También 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.

4 Variables. Declaración. Inicialización y utilización.

4.1 Tipos de variables

En un programa podemos encontrar distintos tipos de variables. Las diferencias entre una variable y otra dependen 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. 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.
    - 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. Igual que las variables miembro, las variables locales también pueden ser de tipos primitivos o referencias.

4.2 Constantes

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
```

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. Puede almacenar valores del tipo primitivo int. Su valor 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.

4.3 Declaración e inicialización

Las variables se pueden declarar en cualquier bloque de código, dentro de llaves, mediante 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;
```

Declaramos numAlumnos como una variable de tipo int, y otras dos variables radio e importe de tipo double. Aunque no es obligatorio, se inicializan con un valor.

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, por lo que el compilador le asigna un valor por defecto, que depende del tipo de variable:

Las variables miembro sí se inicializan automáticamente cuando no les damos un valor. Cuando son de tipo numérico se inicializan por defecto a 0, pero si son de tipo carácter se inicializan al carácter null (\0). A las de tipo boolean se les asigna el valor por defecto false, y si son de 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; si no 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.

5 Tipos de datos

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.

5.1 Tipos de datos primitivos

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
```

Como peculiaridad, 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, al elegir el tipo de datos tendremos en cuenta cómo es la información que hay que guardar: de tipo texto, numérico, ... así como el rango de valores que puede alcanzar. A veces, aunque queramos representar un número sin decimales, tendremos que utilizar datos de tipo real.

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

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. Para representar 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.

El tipo de dato real permite representar cualquier número con decimales. Tal como 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, es decir, se traslada 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 x 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. 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.

5.2 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 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, que incluyen 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";
```

```
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
```

Para mostrar por pantalla un mensaje utilizamos System.out, conocido como la salida estándar del programa (o consola).

5.3 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.

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.

```
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
```

EJEMPLO:

```
public enum Demarcacion{PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO}

// Instancia de un enum de la clase Demarcación
Demarcacion delantero = Demarcacion.DELANTERO;

//Devuelve un String con el nombre de la constante (DELANTERO)
delantero.name();

//Devuelve un String con el nombre de la constante (DELANTERO)
delantero.toString();

//Devuelve un entero con la posición del enum según está declarada (3)
delantero.ordinal();

//Compara el enum con el parámetro según el orden declarados lo enum
delantero.compareTo(Enum otro);

//Devuelve un array que contiene todos los enum
Demarcacion.values();
```

EJEMPLO:

```
public enum Equipo
{
        BARÇA("FC Barcelona",1), REAL_MADRID("Real Madrid",2),
        SEVILLA("Sevilla FC",4), VILLAREAL("Villareal",7);

        private String nombreClub;
        private int puestoLiga;

        private Equipo (String nombreClub, int puestoLiga){
                this.nombreClub = nombreClub;
```

```
                this.puestoLiga = puestoLiga;
        }

        public String getNombreClub() {
                return nombreClub;
        }

        public int getPuestoLiga() {
                return puestoLiga;
        }

}

// Instanciamos el enumerado
Equipo villareal = Equipo.VILLAREAL;

// Devuelve un String con el nombre de la constante
System.out.println("villareal.name()= "+villareal.name());

// Devuelve el contenido de los atributos
System.out.println("villareal.getNombreClub()= "+villareal.getNombreClub());
System.out.println("villareal.getPuestoLiga()= "+villareal.getPuestoLiga());
```

RESULTADOS:

```
villareal.name()= VILLAREAL
villareal.getNombreClub()= Villareal
villareal.getPuestoLiga()= 7
```

6 Literales

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

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

7 Operadores y expresiones. Precedencia de operadores

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 en unidades posteriores.

7.1 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

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 se presenta 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	

7.2 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.

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

```
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
```

7.3 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 en las siguientes unidades (Estructuras de control).

7.4 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
```

Para comparar cadenas (String) no se emplea ==. Este operador relacional compara si dos String son de la misma instancia. Ejemplo:

```
String cadena1 = new String("Hola");
String cadena2 = new String("Hola");
if (cadena1 == cadena2)
{
   ...
}
```

da false, las cadenas no son iguales, ya que son dos instancias -dos news- distintos.

Sin embargo:

```
String cadena1 = "Hola";
String cadena2 = "Hola";
if (cadena1 == cadena2)
{
   ...
}
```

daría true, ya que el compilador ve que tiene dos veces la misma cadena y crea una única instancia para ella -sólo hace internamente un new-.

Forma de comparar String: Con el método equals().

```
String cadena1 = new String("Hola");
String cadena2 = new String("Hola");
if (cadena1.equals(cadena2))
{
   ...
}
```

esto daría true siempre que ambas cadenas tengan el mismo texto dentro

También es posible:

```
String cadena2 = new String("Hola");
if ("Hola".equals(cadena2))
{
   ...
}
```

es decir, podemos usar directamente el método equals() sobre una cadena de texto literal, sin necesidad de tener variable. Esto es útil para evitar comparar primero si la cadena es null antes de llamar a equals(). Es decir, evita hacer esto

```
String cadena2 = new String("Hola");
if (cadena2!=null)
   if (cadena2.equals("Hola"))
   {
      ...
   }
```

Podemos comparar las cadenas sin importar las mayúsculas o minúsculas:

```
String a = "hola";
String b = "HOLA";

// son iguales
if (a.equalsIgnoreCase(b)) {
   System.out.println("a y b son iguales");
}
```

7.5 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

```java
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
```

7.6 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

8 Los Bits

El método más sencillo de representación son los números naturales. Por ejemplo, si tengo el número 85 en decimal, solo tengo que llevarlo a binario y obtengo una serie de unos y ceros:

```
1010101 = 85 en binario
```

Cada dígito (un cero o un uno) de este número se llama bit. Java tiene una serie de operadores capaces de manipular estos dígitos, son los operadores de bits.

Operadores de bits

Operador	Utilización	Resultado
<<	A << B	Desplazamiento de A a la izquierda en B posiciones
>>	A >> B	Desplazamiento de A a la derecha en B posiciones, tiene en cuenta el signo.
>>>	A >>> B	Desplazamiento de A a la derecha en B posiciones, no tiene en cuenta el signo.
&	A & B	Operación AND a nivel de bits
|	A | B	Operación OR a nivel de bits
^	A ^ B	Operación XOR a nivel de bits
~	~A	Complemento de A a nivel de bits

Para operar a nivel de bit es necesario tomar toda la longitud predefinida para el tipo de dato. Estamos acostumbrados a desechar los ceros a la izquierda en nuestra representación de números. Pero aquí es importante. Si trabajamos una variable de tipo short con un valor de 3, está representada de la siguiente manera :

```
0000000000000011
```

Aquí los 16 bits de un short se tienen en cuenta.

9 Desplazamientos

Los operadores de desplazamiento, mueven los bits a la izquierda o a la derecha. El primer operando será la victima a sacudir. El segundo indicará cuantas posiciones.

9.1 Desplazamiento a la izquierda

Deseamos correr el número 33 dos posiciones a la izquierda. Entonces realizamos :

```
int j = 33;
```

```
int k = j << 2;
```

Este es el resultado:

```
00000000000000000000000000100001 : j = 33
00000000000000000000000010000100 : k = 33 << 2 ; k = 132
```

Cada "hueco" que queda a la derecha tras correr este número se rellena con ceros. Los bits a la izquierda se pierden, no es una operación de rotación. Si prestamos atención, observaremos que esta operación multiplicó a j por 2 tantas veces como posiciones se ha desplazado. En este caso se multiplicó por 4 (2 x 2). Hay que notar que el signo del número puede cambiar tras la operación (por ejemplo 1 << 31 = -2147483648).

9.2 Desplazamiento a la derecha con signo

Volvamos a colocar como estaban los bits del caso anterior. Queremos obtener nuevamente el número 33. Para esto desplazamos el número 132 dos posiciones a la derecha.

```
int k = 132;
int m = k >> 2;
```

Como resultado obtenemos el número original.

```
00000000000000000000000010000100 : k = 132
00000000000000000000000000100001 : m = 132 >> 2 ; m = 33
```

Podemos ver que el corrimiento a la derecha realiza una división de enteros. Divide por 2, tantas veces como posiciones desplazadas.

Veamos que ocurre si pretendemos realizar un desplazamiento a la derecha con un número negativo. Tengan en cuenta que la representación de números es de complemento a 2. Si tengo una variable de tipo int con el valor –1 , internamente está almacenada de la siguiente forma :

```
11111111111111111111111111111111 : -1 complemento a 2
```

Ahora realicemos un programa para ver que ocurre con el desplazamiento.

```
public class CorreNeg {
   public static void main(String args[]){
          int x = -1;
          int y = x >> 2;
          System.out.println("El resultado es: " + String.valueOf(y));
   }
}
```

La salida del programa nos indica que:

```
El resultado es: -1
```

Quedó exactamente igual. Prueben de correr el número tantas posiciones como tengan

ganas y obtendrán el mismo resultado. Esto ocurre porque en el desplazamiento, los "huecos" que quedan a la izquierda se rellenan con el bit uno (1), quedando inalterable.

Este operador desplaza el conjunto de bit a la derecha y agrega a la izquierda los bits que faltan según el bit de signo, o sea el más significativo. Si se encuentra con un número positivo, el bit de signo vale 0, entonces agrega ceros, en cambio si son negativos el bit de signo vale 1, entonces agrega unos. Este proceso, denominado extensión de signo mantiene el signo del número como si se tratara de una división. Por esto se lo conoce como desplazamiento con signo.

9.3 Desplazamiento a la derecha sin signo

Modifiquemos ligeramente el programa anterior agregándole al operador un símbolo >. Nos queda de esta manera :

```
int x = -1;
int y = x >>> 2;
```

Si ejecutamos el programa nos dice lo siguiente :

```
El resultado es: 1073741823
```

Veamos de donde salió este número raro. Si lo llevamos a binario tenemos :

```
00111111111111111111111111111111 : 1073741823 en binario
```

Ahora nos damos cuenta que se han agregado dos ceros a la izquierda. Este operador desplaza a la derecha, pero no tiene en cuenta el signo. Siempre agrega bit con el valor cero, por lo que se llama desplazamiento sin signo. Este operador suele ser más adecuado que el >> cuando queremos manipular los bits mismos, no su representación numérica.

10 Operadores lógicos de bits

Estos operadores extienden las operaciones booleanas a los enteros. Para comprender como trabajan debemos descomponer los enteros en un conjunto de bits. El operador aplicará una operación lógica bit por bit, tomando el valor de uno como verdadero y el valor de cero como falso. De un operando toma un bit y aplica la operación al bit que tiene la misma posición del segundo operando. Como resultado obtenemos otro entero.

10.1 Operador AND de Bits

Si ambos bits comparados son 1, establece el resultado en 1. De lo contrario da como resultado 0.

```
int k = 132;   // k: 00000000000000000000000010000100
int l = 144;   // l: 00000000000000000000000010010000
int m = k & l; // m: 00000000000000000000000010000000

El resultado da 128
```

10.2 Operador OR de Bits

Si por lo menos uno de los dos bits comparados es 1, establece el resultado en 1. De lo contrario da como resultado 0.

```
int k = 132;   // k: 00000000000000000000000010000100
int l = 144;   // l: 00000000000000000000000010010000
int m = k | l; // m: 00000000000000000000000010010100

El resultado da 148
```

10.3 Operador XOR de Bits

Si uno de los bits comparados es 0 y el otro 1, el resultado es 1. Si ambos bits comparados son iguales, el resultado es 0.

```
int k = 132;   // k: 00000000000000000000000010000100
int l = 144;   // l: 00000000000000000000000010010000
int m = k ^ l; // m: 00000000000000000000000000010100

El resultado da 20
```

10.4 Operador NOT de Bits

Sólo invierte los bits, es decir, convierte los ceros en unos y viceversa. Observemos que es el único de esta familia que tiene un solo operando.

```
int k = 132;   // k: 00000000000000000000000010000100
int m = ~k;    // m: 11111111111111111111111101111011

El resultado da -133
```

Como los enteros negativos en Java se representan con el método del complemento a dos, podemos realizar un sencillo experimento de cambiarle el signo a un número. Para realizarlo debemos aplicar a un entero el operador NOT y sumarle uno.

```
int x = 123;
int y = ~x;
int z = y + 1;

El resultado da -123,
```

10.5 Trabajo con cadenas

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.

```
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
```

10.6 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

11 Conversiones de tipo. Implícitas y explícitas (casting).

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 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
```

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	-

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.

Reglas de promoción

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.

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);
y=Math.ceil(num);
z=Math.floor(num);
// x = 4
// y = 4
// z = 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';
c = (char) 65;
c = (char) ((int) 'A' + 1);
//num = 65
// c = 'A'
// c = 'B'
```

Conversiones de tipo con cadenas de caracteres

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 leemos de teclado un número 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);
```

12 Comentarios

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 */. Igual que con loscomentarios 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
*/
```