

## 2. Identificación de los elementos de un programa informático en Java.

Desarrollo de Aplicaciones Multiplataforma. Programación.



# Contenido

## **1. Escritura de programas Java.**

1.1 Codificación del texto.

1.2 Notas previas.

## **2. Primer programa en Java.**

2.1 Proceso de compilación y ejecución desde la línea de comandos.

## **3. Identificadores.**

3.1 Reglas.

3.2 Convenios.

3.3 Palabras reservas.

## **4. Variables. Declaración, inicialización y utilización.**

4.1 Asignación.

4.2 Ámbito de las variables.

4.3 Constantes.

# 1. Escritura de programas Java.

## 1.1 Codificación del texto.

- El código fuente Java se escribe en documentos de texto con extensión .java.
- Por lo tanto, para escribirlo basta cualquier editor de texto como el bloc de notas.
- Al ser un lenguaje multiplataforma y especialmente pensado para su integración en redes, la codificación de texto utiliza el estándar Unicode, lo que implica que los programadores hispanohablantes podemos utilizar sin problemas símbolos de nuestra lengua como la ñe o las vocales con tildes o diéresis a la hora de poner nombre a nuestras variables.
- La codificación Unicode usa normalmente 16 bits (2 bytes por carácter) e incluye la mayoría de los códigos alfabéticos del mundo.

# 1. Escritura de programas Java.

## 1.2 Notas previas.

- Los archivos con código fuente en Java deben guardarse con la extensión .java y algunos detalles importantes a la hora de codificar son los siguientes:
  - En java (como en C) hay diferencia entre mayúsculas y minúsculas.
  - Cada instrucción en código debe terminar con.
  - Una instrucción puede abarcar más de una línea. Además se pueden dejar espacios y tabuladores a la izquierda e incluso en el interior de la instrucción para separar elementos de la misma.
  - Los comentarios, si son de una línea, deben comenzar con // y si ocupan más de una línea deben comenzar con /\* y terminar con \*/

# 1. Escritura de programas Java.

## 1.2 Notas previas.

*/\* Comentario*

*de varias líneas \*/*

*//Comentario de una línea*

*/\*\*Comentario de más de una línea que además se almacena en javadoc.*

*\*Javadoc extrae los comentarios del código y genera un archivo html a partir de este tipo de comentarios \*/*

- A veces se marcan bloques de código, es decir código agrupado. Cada bloque comienza con { y termina con }

*{*

*...código dentro del bloque*

*}*

*código fuera del bloque*

## 2. Primer programa en Java.

```
public class PrimerPrograma
{
    public static void main(String[] args) {
        System.out.println ("¡Mi primer programa!");
    }
}
```

## 2. Primer programa en Java.

Este código sirve para escribir ¡Mi primer programa! en la pantalla.

Para empezar a entender el código:

- La primera línea (`public class PrimerPrograma`) declara el nombre de la clase del código. (Más adelante se explicará que significa el concepto de clase). Un programa en java puede estar formado por un conjunto de clases. Cada una de estas clases está formada por datos y funciones (métodos). Entre todas estas clases que forman un programa, siempre habrá una clase que tenga una función con la siguiente cabecera: `public static void main (Strings args)`.
- La línea `public static void main(String args[])`, sirve para indicar el inicio del método main. Es la función por donde el programa empezará la ejecución. Este método contiene las instrucciones que se ejecutarán cuando el programa arranque. Es decir lo que está tras las llaves del main, es el programa en sí.
- La instrucción `System.out.println` sirve para escribir en pantalla. Como lo que escribimos es un texto, se encierra entre comillas.

## 2. Primer programa en Java.

- El archivo es aconsejable llamarlo PrimerPrograma.java ya que el nombre de la clase y el del archivo pueden coincidir. Y también, es más que aconsejable que el nombre de la clase comience con una letra mayúscula y le sigan letras en minúsculas. Si consta de varias palabras no pueden utilizarse espacios en blanco, por lo que se suelen juntar las palabras poniendo cada inicial de la palabra en mayúsculas.
- Éste tipo de reglas no obligatorias sino aconsejables (como por ejemplo el hecho de que las instrucciones interiores a un bloque dejen espacio a su izquierda) producen un código más legible y sobre todo hace que todos los programadores del planeta adoptemos la misma forma de escribir, simplificando el entendimiento del código.
- Para poder ejecutar el programa en java no hace falta usar un entorno de desarrollo, con tener instalado el JDK es suficiente.



## 2. Primer programa en Java.

### 2.1 Proceso de compilación y ejecución desde la línea de comandos.

- La compilación del código java se realiza mediante el programa javac incluido en el software de desarrollo de Java. La forma de compilar es (desde la línea de comandos):

```
javac archivo.java
```

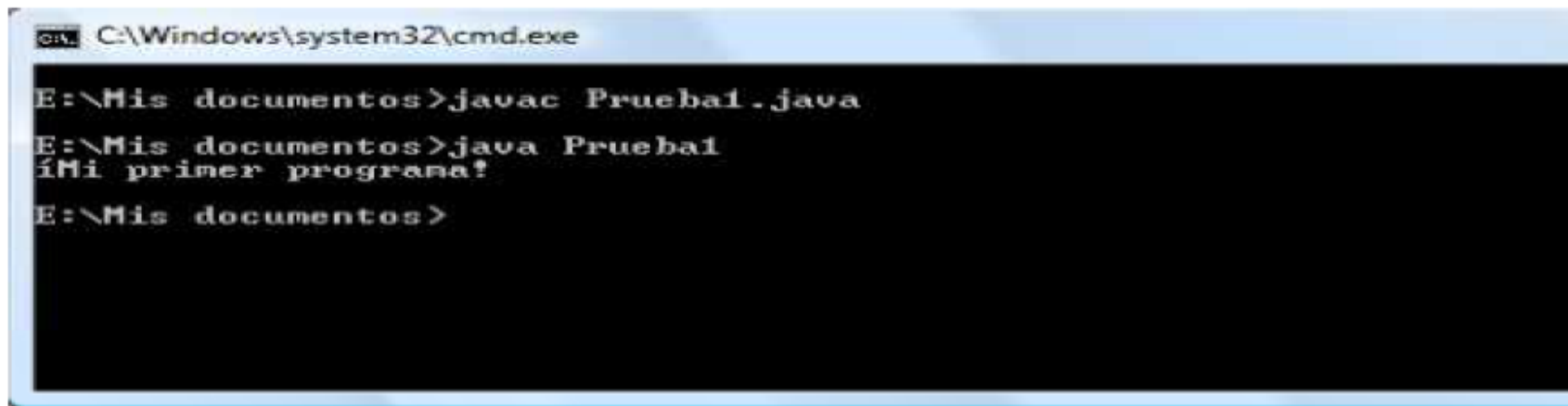
- El resultado de esto es un archivo con el mismo nombre que el archivo java pero con la extensión class. Esto ya es el archivo con el código en forma de bytecode. Es decir, con el código precompilado. Si el programa es ejecutable (sólo lo es si contiene el método main), el código se puede interpretar usando el programa java del kit de desarrollo (que se encuentra en el mismo sitio que javac). Sintaxis:

```
java archivo
```

## 2. Primer programa en Java.

### 2.1 Proceso de compilación y ejecución desde la línea de comandos.

- Estos comandos hay que escribirlos desde la línea de comandos en la carpeta en la que se encuentre el programa. Pero antes hay que asegurarse de que los programas del kit de desarrollo son accesibles desde cualquier carpeta del sistema. Para ello hay que haber configurado adecuadamente las variables del sistema (Path y Classpath).



```
C:\Windows\system32\cmd.exe

E:\Mis documentos>javac Prueba1.java
E:\Mis documentos>java Prueba1
¡Mi primer programa!
E:\Mis documentos>
```

### 3. Identificadores.

Se llama identificadores a los nombres que da el programador a: variables, funciones, clases,...

REGLAS:

- El primer carácter debe ser una letra, \_ ó \$.
- El lenguaje java distingue entre mayúsculas y minúsculas.
- Permite que el nombre del identificador esté acentuado y la letra ñ
- Se permite el uso de \_ y números, pero nunca al principio.
- No se permiten espacios en blanco como parte del identificador.
- No se permiten símbolos que no se consideren letras: +, -, %, /, ..
- Existen una serie de palabras reservadas.

### 3. Identificadores.

#### CONVENIOS:

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.	numPersonas, suma, suma2, edadAdulto
Nombre d CONSTANTE	En letras Mayúsculas separando las palabras con el guion bajo	PERIM_MAX, PI, CANT2, OCULTO
Nombre de una Clase	Comienza por letra Mayúscula.	MiModulo, Cadenas, MiPrimerPrograma
Nombre de función o método	Comienza con letra minúscula, y si tienen más de una palabra se separan por guion bajo.	modifica_Valores, obtiene_Valores

### 3. Identificadores.

#### PALABRAS RESERVADAS

Son identificadores usados por el lenguaje, por ejemplo:

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

## 4. Variables. Declaración, inicialización y utilización.

- Las variables son contenedores que sirven para almacenar los datos que utiliza un programa. Dicho más sencillamente, son nombres que asociamos a determinados datos. La realidad es que cada variable ocupa un espacio en la memoria RAM del ordenador para almacenar el dato al que se refiere. Es decir cuando utilizamos el nombre de la variable realmente estamos haciendo referencia a un dato que está en memoria. Antes de poder utilizar una variable, ésta se debe declarar. Lo cual se debe hacer de esta forma:

*tipo nombreVariable;*

- Donde tipo es el tipo de datos que almacenará la variable (texto, números enteros,...) y nombreVariable es el nombre con el que se conocerá la variable.

## 4. Variables. Declaración, inicialización y utilización.

- Ejemplos:

*int días; // días es un número entero, sin decimales*

*boolean decisión; //decisión sólo puede ser verdadera o falsa*

- También se puede hacer que la variable tome un valor inicial al declarar:

*int días=365;*

- Y se puede declarar más de una variable a la vez del mismo tipo en la misma línea si las separamos con comas

*int días=365, año=23, semanas;*

- Al declarar una variable se puede incluso utilizar una expresión:

*int a=13, b=18;*

*int c=a+b; //es válido, c vale 31*

## 4. Variables. Declaración, inicialización y utilización.

- Java es un lenguaje muy estricto al utilizar tipos de datos.
- Variables de tipos distintos son incompatibles. Algunos autores hablan de lenguaje fuertemente tipado o incluso lenguaje muy tipificado.
- Se debe a una traducción muy directa del inglés strongly typed referida a los lenguajes que, como Java, son muy rígidos en el uso de tipos.
- Parte de la seguridad y robustez de las que hace gala Java se deben a esta característica.



## 4. Variables. Declaración, inicialización y utilización.

### 4.1 Asignación.

- En Java para asignar valores a una variable, basta con utilizar el signo =. Ya se ha visto en el apartado anterior que al declarar se puede asignar un valor:

```
int x=7;
```

- Pero la asignación se puede utilizar en cualquier momento (tras haber declarado la variable):

```
int x;
```

```
x=7;
```

```
x=x*2;
```

- Como se ve en la última línea anterior, la expresión para dar el valor a la variable puede ser tan compleja como queramos.

## 4. Variables. Declaración, inicialización y utilización.

### 4.2 Ámbito de las variables.

- Toda variable tiene un ámbito. Esto es la parte del código en la que una variable se puede utilizar. De hecho las variables tienen un ciclo de vida:
  - (1) En la declaración se reserva el espacio necesario para que se puedan comenzar a utilizar (digamos que se avisa de su futura existencia).
  - (2) Se la asigna su primer valor (la variable nace).
  - (3) Se la utiliza en diversas sentencias.
  - (4) Cuando finaliza el bloque en el que fue declarada, la variable muere. Es decir, se libera el espacio que ocupa esa variable en memoria. No se la podrá volver a utilizar.
- Una vez que la variable ha sido eliminada, no se puede utilizar. Dicho de otro modo, no se puede utilizar una variable más allá del bloque en el que ha sido definida.

## 4. Variables. Declaración, inicialización y utilización.

### 4.2 Ámbito de las variables.

- Ejemplo:

```
{ int x=9;
```

```
}
```

```
int y=x; //error, ya no existe x
```

- Según dónde esté declarada una variable, puede ser:
  - Variable miembro de una clase.
  - Variables locales.
- Las variables miembro (de toda la clase), al ser declaradas toman valores por defecto; Las variables numéricas toman el valor 0, los caracteres null(\0), las booleanas false, y las referenciadas a null.

```
int x; //x ya vale cero
```

- Las variables locales en cambio no se inicializan solas.

## 4. Variables. Declaración, inicialización y utilización.

### 4.3 Constantes.

- Son espacios de memoria identificados con un nombre cuyo valor permanece invariable a lo largo del programa.
- En Java, para poder conseguir esto, se utiliza la palabra final en el momento de la declaración de la variable, consiguiendo con ello que el valor que contiene no se pueda modificar durante toda la ejecución del programa.
- En el momento de la declaración exige asignarle un valor, por ejemplo:

*final float PI=3.14159;*

*PI=4; //Error, no podemos cambiar el valor de PI*

## 5. Tipos de datos.

- Un tipo de dato proporciona una descripción al compilador de:
  - Cuánta memoria se debe asignar.
  - Qué tipo de dato se puede almacenar (números, letras,..)
  - Qué operaciones se pueden realizar sobre esos datos.
- 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.

## 5. Tipos de datos.

### 5.1 Tipos de datos primitivos.

Tipo de variable		Bytes que ocupa	Rango de valores
<b>boolean</b>		1	true, false
ENTERO	<b>byte</b>	1	-128 a 127
	<b>short</b>	2	-32.768 a 32.767
	<b>int</b>	4	-2.147.483.648 a 2.147.483.649
	<b>long</b>	8	$-9 \cdot 10^{18}$ a $9 \cdot 10^{18}$
DECIMAL	<b>float</b>	4	$-3,4 \cdot 10^{38}$ a $3,4 \cdot 10^{38}$
	<b>double</b>	8	$-1,79 \cdot 10^{308}$ a $1,79 \cdot 10^{308}$
<b>char</b>		2	Caracteres (en Unicode)

# 5. Tipos de datos.

## 5.1 Tipos de datos primitivos.

### Numéricos enteros

- Se usan para representar números enteros con signo.
- Hay cuatro tipos: byte, short, int y long.
- Los literales enteros se almacenan por defecto en el tipo int, (4 bytes con signo), o si se trabaja con números muy grandes, con el tipo long, (8 bytes con signo), añadiendo una L ó l al final del número.

### EJEMPLO: Variables1

```
public class Variables1 {  
  
    public static void main(String[] args) {  
        byte varByte = 1;  
        short varShort = 3276;  
        int varInt = 100000;  
        long varLong = 1000000000L;  
  
        System.out.println (" Mi dato1="+varByte);  
        System.out.println (" Mi dato2="+varShort);  
        System.out.println (" Mi dato3="+varInt);  
        System.out.println (" Mi dato4="+varLong);  
    }  
}
```

# 5. Tipos de datos.

## 5.1 Tipos de datos primitivos.

### Numéricos enteros

- No se acepta en general asignar variables de distinto tipo. Sí se pueden asignar valores de variables enteras a variables enteras de un tipo superior (por ejemplo asignar un valor int a una variable long). Pero al revés no se puede:

```
int i=12;
```

```
byte b=i;//error de compilación, posible pérdida de precisión
```

- La solución es hacer un cast. Esta operación permite convertir valores de un tipo a otro. Se usa así:

```
int i=12;
```

```
byte b=(byte) i; //El (cast) evita el error
```

- Hay que tener en cuenta en estos castings que si el valor asignado sobrepasa el rango del elemento, el valor convertido no tendrá ningún sentido ya que no puede almacenar todos los bits necesarios para representar ese número:

```
int i=1200;
```

```
byte b=(byte) i; //El valor de b no tiene sentido
```



## 5. Tipos de datos.

### 5.1 Tipos de datos primitivos.

#### Numéricos con decimales

- Los decimales se almacenan en los tipos float y double. Se les llama de coma flotante por como son almacenados por el ordenador. Los decimales no son almacenados de forma exacta por eso siempre hay un posible error. En los decimales de coma flotante se habla, por tanto de precisión. Es mucho más preciso el tipo double que el tipo float.
- Para asignar valores literales a una variable de coma flotante, hay que tener en cuenta que el separador decimal es el punto y no la coma. Es decir para asignar el valor 2,75 a la variable x se haría: `x=2.75`;
- A un valor literal (como 1.5 por ejemplo), se le puede indicar con una f al final del número que es float (1.5f por ejemplo) o una d para indicar que es double. Si no se indica nada, un número literal siempre se entiende que es double, por lo que al usar tipos float hay que convertir los literales. Los valores decimales se pueden representar en notación decimal: 1.345E+3 significaría  $1,345 \cdot 10^3$  o lo que es lo mismo 1345.

# 5. Tipos de datos.

## 5.1 Tipos de datos primitivos.

### Numéricos con decimales

- Lógicamente no podemos asignar valores decimales a tipos de datos enteros:

```
int x=9.5; //error
```

- Sí podremos mediante un cast:

```
int x=(int) 9.5; pero perderemos los decimales (en el ejemplo, x vale 9).
```

- El caso contrario sin embargo sí se puede hacer:

```
int x=9;
```

```
double y=x; //correcto
```

- La razón es que los tipos de coma flotante son más grandes que los enteros, por lo que no hay problema de pérdida de valores.

## 5. Tipos de datos.

### 5.1 Tipos de datos primitivos.

Numéricos con decimales

EJEMPLO: Variables2

Añade, declara e inicializa una variable tipo double.

```
public class Variables2 {  
    public static void main(String[] args) {  
        float varFloat;  
        varFloat=3.46f;  
        System.out.println (" El valor de la variable es="+varFloat);  
    }  
}
```

# 5. Tipos de datos.

## 5.1 Tipos de datos primitivos.

### Booleana

- Los valores booleanos (o lógicos) sirven para indicar si algo es verdadero (true) o falso (false).

```
boolean b=true;
```

- A diferencia del lenguaje C, no se pueden en Java asignar números a una variable booleana (en C, el valor false se asocia al número 0, y cualquier valor distinto de cero se asocia a true).
- Tampoco tiene sentido intentar asignar valores de otros tipos de datos a variables booleanas mediante casting:

```
boolean b=(boolean) 9; //no tiene sentido
```

## 5. Tipos de datos.

### 5.1 Tipos de datos primitivos.

#### Booleana

EJEMPLO: Variables3.

```
public class Variables3 {  
    public static void main(String[] args) {  
        boolean varTipoBool;  
        varTipoBool=true;  
        System.out.println (" El valor de la variable es="+varTipoBool);  
    }  
}
```

# 5. Tipos de datos.

## 5.1 Tipos de datos primitivos.

### Caracteres

- Los valores de tipo carácter sirven para almacenar símbolos de escritura (en Java se puede almacenar cualquier código Unicode). Los valores Unicode son los que Java utiliza para los caracteres.

*char letra;*

*letra='C'; //Los caracteres van entre comillas*

*letra=67; //El código Unicode de la C es el 67. Esta línea*

*//hace lo mismo que la anterior*

# 5. Tipos de datos.

## 5.2 Conversión entre tipos (casting).

- Ya se ha comentado anteriormente la necesidad de uso del operador de casting para poder realizar asignaciones entre tipos distintos. Como resumen general del uso de casting véanse estos ejemplos:

```
int a;
```

```
byte b=12;
```

```
a=b;
```

- El código anterior es correcto porque un dato byte es más pequeño que uno int y Java le convertirá de forma implícita. Lo mismo pasa de int a double por ejemplo. Sin embargo en:

```
int a=1;
```

```
byte b;
```

```
b=a;
```

# 5. Tipos de datos.

## 5.2 Conversión entre tipos (casting).

- El compilador devolverá error aunque el número 1 sea válido para un dato byte. Para ello hay que hacer un casting o conversión explícita. Eso significa poner el tipo deseado entre paréntesis delante de la expresión.

```
int a=1;
```

```
byte b;
```

```
b= (byte) a; //correcto
```

- En el siguiente ejemplo:

```
byte n1=100, n2=100, n3;
```

```
n3= n1 * n2 /100;
```



## 5. Tipos de datos.

### 5.2 Conversión entre tipos (casting).

- Aunque el resultado es 100, y ese resultado es válido para un tipo byte; lo que ocurrirá en realidad es un error.
- Eso es debido a que la multiplicación  $100 * 100$  da como resultado 10000, es decir un número de tipo int.
- Aunque luego se divide entre 100, no se vuelve a convertir a byte; ya que ante cualquier operación el tipo resultante siempre se corresponde con el tipo más grande que intervenga en la operación.
- Lo correcto sería:  $n3 = (\text{byte})(n1 * n2 / 100);$
- Para la conversión automática o implícita hacemos la conversión de un tipo de más bits a uno de menos. Por ello se establece la norma de que "en las conversiones el tipo destino siempre debe ser igual o mayor que el tipo fuente".

## 5. Tipos de datos.

### 5.2 Conversión entre tipos (casting).

Tipo Origen	Tipo Destino
<i>byte</i>	<i>double, float, long, int, char, short</i>
<i>short</i>	<i>double, float, long, int</i>
<i>char</i>	<i>double, float, long, int</i>
<i>int</i>	<i>double, float, long</i>
<i>long</i>	<i>double, float</i>
<i>float</i>	<i>double</i>

## 5. Tipos de datos.

EJEMPLO: Ejemplo de conversión implícita:

```
public class ConverImplicita {  
  
    public static void main(String[] args) {  
        int int1=5,int2,int3;  
        short short1=10;  
        char charc='f';  
        float float1;  
        //conversiones implícitas  
        int2=charc;//conversión implícita de char a int  
        float1=int1; //conversión implícita de int a float  
        int3=short1;//conversión implícita de short a int  
        System.out.println("El valor de c(tipo char) es "+charc+" y ese valor en  
            un int ha quedado en "+int2);  
        System.out.println("El valor de k(tipo int) es "+int1+" y ese valor en un  
            float ha quedado en "+float1);  
        System.out.println("El valor de s(tipo short) es "+short1+" y ese valor  
            en un int ha quedado en "+int3);    }}  
}
```

# 5. Tipos de datos.

## 5.2 Conversión entre tipos (casting).

- Conversión explícita: Cuando hacemos una conversión de un tipo con menos bits a otro con más bits.
- Formato:

*variable\_donde\_se\_guarda = (tipo\_variable\_donde\_se\_guarda)*

*variable\_contiene\_dato\_convertir;*

- En Java es posible transformar el tipo de una variable u objeto en otro diferente al original con el que fue declarado. Este proceso se denomina "conversión", "moldeado", "tipado" o "casting". La conversión se lleva a cabo colocando el tipo destino entre paréntesis, a la izquierda del valor que queremos convertir.
- El tamaño de los tipos que queremos convertir es muy importante. No todos los tipos se convertirán de forma segura. Por ejemplo, al convertir un long en un int, el compilador corta los 32 bits superiores del long (de 64 bits), de forma que encajen en los 32 bits del int, con lo que, si contienen información útil, ésta se perderá.

## 5. Tipos de datos.

EJEMPLO: Ejemplo de conversión explícita:

```
public class ConverExplicita {  
    public static void main(String[] args) {  
        int int1,int2;  
        char chark;  
        byte bytes=70;  
        float floath=9.2f;  
        long longc=1000000000L;  
        //conversiones explícitas  
        int1=(int)longc;  
        chark=(char)bytes; /*cuando la variable destino es char y el origen  
es numérico, la conversión no puede ser automática*/  
        int2=(int)floath; /*trunca la parte decimal*/  
        System.out.println("El valor de longc(tipo long) es "+longc+ " y ese  
valor en un int ha quedado en "+int1);  
        System.out.println("El valor de bytes(tipo byte) es "+bytes+ " y ese  
valor en un char ha quedado en "+chark);  
        System.out.println("El valor de floath(tipo float) es "+floath+ " y  
ese valor en un int ha quedado en "+int2);  
    }  
}
```

## 5. Tipos de datos.

### 5.3 Tipos de datos referenciados.

- Un tipo de dato referenciado no guarda el valor real del dato, sino que almacena una dirección en memoria. En Java los objetos y los arrays entre otros.

*int[] arrayDeEnteros;*

*Cuenta cuentaCliente;*

## 5. Tipos de datos.

### 5.3 Tipos de datos referenciados.

#### Arrays (vectores)

- En Java los vectores se declaran utilizando corchetes ( [ y ] ), tras la declaración del tipo de datos que contendrá el vector. Por ejemplo, esta sería la declaración de un vector de números enteros (int):

```
int vectorNumeros[ ]; // Vector de números
```

- Se observa la ausencia de un número que indique cuántos elementos componen el vector, debido a que Java no deja indicar el tamaño de un vector vacío cuando le declara. La asignación de memoria al vector se realiza de forma explícita en algún momento del programa.

## 5. Tipos de datos.

### 5.3 Tipos de datos referenciados.

#### Arrays (vectores)

- Para ello o se utiliza el operador new:

```
int vectorNumeros = new int[ 5 ]; // Vector de 5 números
```

- O se asigna una lista de elementos al vector:

```
int vectorIni = { 2, 5, 8}; // == int vectorIni[3]=new int[3];
```

- Se puede observar que los corchetes son opcionales en este tipo de declaración de vector, tanto después del tipo de variable como después del identificador. Si se utiliza la forma de new se establecerá el valor 0 a cada uno de los elementos del vector.



# 5. Tipos de datos.

## 5.3 Tipos de datos referenciados.

### Strings (cadenas)

- En Java se tratan como una clase especial llamada String. Las cadenas se gestionan internamente por medio de una instancia de la clase String. Una instancia de la clase String es un objeto que ha sido creado siguiendo la descripción de la clase.

### *Cadenas constantes*

- Representan múltiples caracteres y aparecen dentro de un par de comillas dobles.
- Se implementan en Java con la clase String.
- Cuando Java encuentra una constante de cadena, crea un caso de la clase String y define su estado, con los caracteres que aparecen dentro de las comillas dobles.
- Vemos un ejemplo de cadena declarada con la clase String de Java:

*String capitalUSA = "Washington D.C.";*

*String nombreBonito = "Amelia";*

## 5. Tipos de datos.

### 5.4 Tipos de datos 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.

## 5. Tipos de datos.

### 5.4 Tipos de datos enumerados.

- Los tipos enumerados son un vector de números naturales  $\{0, \dots, n\}$  lo que ocurre es que, para facilitar la comprensión del código, cambias la forma de representar los números  $\{0, 1, 2, \dots, n\}$  por otra más representativa {Lunes, Martes, ..., Domingo}.
- Pero no debes olvidar que los elementos no son String ni nada por el estilo son números.
- Piensa por ejemplo los números romanos, donde no debes ver V como una letra sino como el número 5. Aquí pasa lo mismo el elemento Lunes de un tipo Enumerado no es la cadena de caracteres "Lunes", es el número 0.
- Puedes devolver el valor numérico de un elemento con el método ordinal(), por ejemplo, si la variable diaSemana tiene el valor lunes, diaSemana.ordinal() devolverá 0. diaSemana.ordinal()+1 devolverá 1 etc.

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

#### Operadores aritméticos básicos.

- En Java disponemos de los operadores aritméticos habituales en lenguajes de programación como son suma, resta, multiplicación, división y operador que devuelve el resto de una división entre enteros (en otros lenguajes denominado operador mod o módulo de una división).

OPERADOR	DESCRIPCIÓN
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de una división entre enteros (en otros lenguajes mod)

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Hay que tener en cuenta que el resultado de estos operadores varía notablemente si usamos enteros o si usamos números de coma flotante. Por ejemplo:

*double resultado1, d1=14, d2=5;*

*int resultado2, i1=14, i2=5;*

*resultado1= d1 / d2;*

*resultado2= i1 / i2;*

- resultado1 valdrá -- mientras que resultado2 valdrá --.

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Es más incluso:

```
double resultado;
```

```
int i1=7,i2=2;
```

```
resultado=i1/i2; //Resultado valdrá --
```

```
resultado=(double)i1/i2; //Resultado valdrá --
```

- Ejemplo:

```
int resultado, i1=14, i2=5;
```

```
resultado = i1 % i2; //El resultado será --
```

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

Operadores relacionales y operadores lógicos.

- Los Operadores de relación sirven para comparar valores y siempre devuelven valores booleanos.

	operador	significado
De RELACIÓN	<	Menor
	>	Mayor
	>=	Mayor o igual
	<=	Menor o igual
	==	Igual
	!=	Distinto

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Ejemplo: Ejemplo operadores de relación.

```
public class OpRelacion {  
    public static void main(String[] args) {  
        int x=100;  
        boolean r1,r2;  
        r1=x>10;  
        r2=x<10;  
        System.out.println(r1); /*---- */  
        System.out.println(r2); /*----*/  
    }  
}
```



## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Java soporta cinco operadores lógicos, mostrados en la siguiente tabla:

Operador	Uso	Devuelve verdadero si...
<code>&amp;&amp;</code>	<code>op1 &amp;&amp; op2</code>	op1 y op2 son ambos verdaderos, condicionalmente evalúa op2 (si op1 es falso ya no se evalúa)
<code>&amp;</code>	<code>op1 &amp; op2</code>	op1 y op2 son ambos verdaderos, siempre evalúa op1 y op2
<code>//</code>	<code>op1 // op2</code>	op1 o op2 son verdaderos, condicionalmente evalúa op2 (si op1 es verdadero ya no se evalúa)
<code>/</code>	<code>op1 / op2</code>	op1 o op2 son verdaderos, siempre evalúa op1 y op2
<code>^</code>	<code>op1 ^ op2</code>	si sólo uno de los operadores es verdadero
<code>!</code>	<code>! op</code>	op es falso

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Ejemplo: uso de operadores lógicos.

```
public class OpLogicos {  
    public static void main(String[] args) {  
        boolean result1,result2;  
        float dato1=15,dato2=18,dato3=20;  
        result1=dato1==dato2;  
        result2=(dato1<dato2)&&(dato2<=dato3);  
        System.out.println("El valor de la variable resul1 es"+result1);/*---- */  
        System.out.println("El valor de la variable resul2 es"+result2);/*---- */  
    }  
}
```

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Los operadores lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. El operador (NOT) sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;
```

```
int edad = 21;
```

```
mayorDeEdad = edad >= 18; //mayorDeEdad será ---
```

```
menorDeEdad = !mayorDeEdad; //menorDeEdad será ---
```

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- El operador && (AND) sirve para evaluar dos expresiones de modo que, si ambas son ciertas, el resultado será true sino el resultado será false. Ejemplo:

```
boolean carnetConducir=true;
```

```
int edad=20;
```

```
boolean puedeConducir= (edad>=18) && carnetConducir;
```

```
//Si la edad es de al menos 18 años y carnetConducir es true,
```

```
// puedeConducir es true
```

- El operador || (OR) sirve también para evaluar dos expresiones. El resultado será true si al menos una de las expresiones es true. Ejemplo:

```
boolean nieva =true, llueve=false, graniza=false;
```

```
boolean malTiempo= nieva || llueve || graniza;
```

- ¿Hace mal tiempo?

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

#### Operadores de asignación.

- Permiten asignar valores a una variable. El fundamental es "=".

```
int contador = 0;
```

- Inicia la variable contador con un valor 0. Sin embargo, se pueden usar expresiones más complejas como:

```
x += 3;
```

- En el ejemplo anterior lo que se hace es sumar 3 a la x (es lo mismo  $x+=3$ , que  $x=x+3$ ). Eso se puede hacer también con todos estos operadores:

```
+= -= *= /= &= |= ^= %= >>= <<=
```

- Supongamos que necesitamos sumar un número a una variable y almacenar el resultado en la misma variable, como a continuación:

```
i = i + 2;
```

- Se puede abreviar esta sentencia con el operador de atajo +=, de la siguiente manera:

```
i += 2;
```

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- La siguiente tabla muestra los operadores de atajo de asignación y sus equivalentes largos:

Operador	Uso	Equivalente a
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>
<code>&amp;=</code>	<code>op1 &amp;= op2</code>	<code>op1 = op1 &amp; op2</code>

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- También se pueden concatenar asignaciones (aunque no es muy recomendable):

*x1 = x2 = x3 = 5; //todas valen 5*

- Otros operadores de asignación son "++" (incremento) y "--" (decremento). Ejemplo:

*x++; //esto es x=x+1;*

*x--; //esto es x=x-1;*

- Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo x++ o ++x. La diferencia estriba en el modo en el que se comporta la asignación. Ejemplo:

*int x=5, y=5, z;*

*z=x++; //z vale 5, x vale 6*

*z=++y; //z vale 6, y vale 6*

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Ejemplo: Indica qué es lo que visualiza estos programas sin pasarlo a ordenador.

```
public class Concat1 {  
    public static void main(String[] args)  
    {  
        int T=4;  
        Scanner entrada=new Scanner (System.in);  
        System.out.println("\n\n\tEl valor que contiene la variable T es " +T );  
        System.out.println("\n\n\tEl valor que contiene la variable T es " + ++T );  
    }  
}
```



## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Ejemplo: Indica qué es lo que visualiza estos programas sin pasarlo a ordenador.

```
public class Concat2 {  
    public static void main(String[] args)  
    {  
        int T=4;  
        Scanner entrada=new Scanner (System.in);  
        System.out.println("\n\n\tEl valor que contiene la variable T es " +T++ );  
        System.out.println("\n\n\tEl valor que contiene la variable T es " + ++T );  
    }  
}
```

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

- Ejemplo: Indica qué es lo que visualiza estos programas sin pasarlo a ordenador.

```
public class Concat3 {  
    public static void main(String[] args)  
    {  
        int T=4;  
        Scanner entrada=new Scanner (System.in);  
        System.out.println("\n\n\tEl valor que contiene la variable T es " +T );  
        System.out.println("\n\n\tEl valor que contiene la variable T es " + T++ );  
        System.out.println("\n\n\tEl valor que contiene la variable T es " + T );  
    }  
}
```

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.1 Operadores.

#### Operador ?

- Este operador (conocido como if de una línea) permite ejecutar una instrucción u otra según el valor de la expresión. Sintaxis:

*expresionlogica?valorSiVerdadero:valorSiFalso;*

- Ejemplo:

*paga=(edad>18)?6000:3000;*

- En este caso si la variable edad es mayor de 18, la paga será de 6000, sino será de 3000. Se evalúa una condición y según es cierta o no se devuelve un valor u otro. Nótese que esta función ha de devolver un valor y no una expresión correcta. Es decir, no funcionaría:

*(edad>18)? paga=6000: paga=3000; //ERROR!!!!*

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.2 Precedencia.

- A veces hay expresiones con operadores que resultan confusas. Por ejemplo, en:

*resultado = 8 + 4 / 2;*

- Es difícil saber el resultado. ¿Cuál es? ¿seis o diez? La respuesta es 10 y la razón es que el operador de división siempre precede en el orden de ejecución al de la suma.
- Es decir, siempre se ejecuta antes la división que la suma. Siempre se pueden usar paréntesis para forzar el orden deseado:

*resultado = (8 + 4) / 2;*

- Cuando en una sentencia aparecen varios operadores el compilador deberá de elegir en qué orden aplica los operadores. A esto se le llama precedencia.

## 6. Operadores y expresiones. Precedencia de operadores.

Tipo de operadores	Operadores de este tipo
Operadores posfijos	[ ] . (parametros) expr++ expr--
Operadores unarios	++expr --expr +expr -expr ~ !
Creación o conversión	new (tipo) expr
Multiplicación	* / %
Suma / Resta	+ -
Desplazamiento	<<
Comparación	< <= instanceof > >=
Igualdad/Desigualdad	== !=
AND a nivel de bit	&
OR a nivel de bit	^
XOR a nivel de bit	
AND lógico	&&
OR lógico	
Condicional	? :
Asignación	= += -= *= /= %= &= ^=  = <<= ==

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.2 Precedencia.

- Los operadores con mayor precedencia son evaluados antes que los operadores con una precedencia relativa menor. Cuando en una sentencia aparecen operadores con la misma precedencia:
  - Los operadores de asignación son evaluados de derecha a izquierda.
  - Los operadores binarios, (menos los de asignación) son evaluados de izquierda a derecha.
- 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

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.2 Precedencia.

- Se puede indicar explícitamente al compilador de Java cómo se desea que se evalúe la expresión con paréntesis balanceados ( ). Para hacer que el código sea más fácil de leer y mantener, es preferible ser explícito e indicar con paréntesis que operadores deben ser evaluados primero.
- Por ejemplo, la siguiente expresión produce un resultado diferente dependiendo de si se realiza la suma o división en primer lugar:  $x + y / 100$
- Si no se indica explícitamente al compilador el orden en que se quiere que se realicen las operaciones, entonces el compilador decide basándose en la precedencia asignada a los operadores.
- Como el operador de división tiene mayor precedencia que el operador de suma el compilador evaluará  $y/100$  primero. Así:  $x + y / 100$  es equivalente a:  $x + (y / 100)$

## 6. Operadores y expresiones. Precedencia de operadores.

### 6.2 Precedencia.

- En la tabla anterior los operadores con mayor precedencia están en la parte superior, los de menor precedencia en la parte inferior.
- De izquierda a derecha la precedencia es la misma. Es decir, tiene la misma precedencia el operador de la suma que el de la resta. Esto último provoca conflictos, por ejemplo, en:

*resultado = 9 / 3 \* 3;*

- El resultado podría ser uno ó nueve. En este caso el resultado es nueve, porque la división y el producto tienen la misma precedencia; por ello el compilador de Java realiza primero la operación que este más a la izquierda, que en este caso es la división.
- Una vez más los paréntesis podrían evitar estos conflictos.