1.4 Tipos de datos primitivos

1. Introducción

Se llaman **tipos primitivos** a los tipos de datos originales de un lenguaje de programación, esto es, aquellos que nos proporciona el lenguaje. Java posee los siguientes:

Tipo de variable	Bytes que ocupa	Rango de valores
boolean	1	true, false
char	2	Caracteres en Unicode
byte	1	-128 a 127
short	2	-32.768 a 32.767
int	4	-2.147.483.648 a 2.147.483.647
long	8	$-9 \cdot 10^{18}$ a $9 \cdot 10^{18}$
float	4	$-3{,}4 \cdot 10^{38}$ a $3{,}4 \cdot 10^{38}$
double	8	$-1{,}79 \cdot 10^{308}$ a $1{,}79 \cdot 10^{308}$

2. Enteros

Los tipos **byte**, **short**, **int** y **long** sirven para almacenar datos enteros. Los enteros son números sin decimales.

Un **literal** es un elemento de programa que representa directamente un valor:

```java
int number=16; //16 es un literal
```

Los literales se pueden expresar de varias maneras:

- En decimal, es como se representan por defecto: `16`
- En binario, anteponiendo 0b: `0b10000`
- En octal, anteponiendo 0: `020`
- En hexadecimal, anteponiendo 0x: `0x10`

Por defecto, un literal entero es de tipo int. Si se le coloca detrás la letra **L**, entonces el literal será de tipo long.

```
int number=16; //16 es un literal entero
long number_long=16L; //16L es un literal de tipo long
```

No se acepta en general asignar variables de distinto tipo pero existen excepciones. Por ejemplo, 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 casting. Esta operación permite convertir valores de un tipo a otro. Se usa así:

```
int i=12;
byte b=(byte) i; //El casting 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
```

Si lo que asignamos a la variable es un literal, java hace una conversión implícita siempre y cuando el literal esté dentro del rango permitido para dicho tipo. Por ejemplo, el siguiente código no da error porque 127 está dentro del rango de los byte aunque el literal sea por defecto int:

```
byte b=127;
```

Sin embargo, el siguiente código sí da error porque 128 sobrepasa el rango de los tipos *byte*:

```
byte b=128; //error
```

En el siguiente ejemplo, utilizamos `System.out.println` para escribir en pantalla el valor de las variables:

```
package tema1_4_TiposDeDatosPrimitivos;

public class Integers {

    public static void main(String[] args) {

        int i;
        long l;
        byte b;
```

```
        short s;

        i = 16;// 16 decimal
        System.out.println(i);
        i = 020;// 20 octal=16 decimal
        System.out.println(i);
        i = 0x10;// 10 hexadecimal=16 decimal
        System.out.println(i);
        i = 0b10000;// 10000 binario=16 decimal
        System.out.println(i);

        l = 6985742369L; // Si se le quita la L da error
        System.out.println(l);
        b = 127; // No da error porque está dentro del rango de los byte aunque el literal
sea por defecto int
        System.out.println(b);
        s = 32767; // No da error porque está dentro del rango de los short aunque el
literal sea por defecto int
        System.out.println(s);

        i = 1200;
        System.out.println(i);
        b = (byte) i;
        System.out.println(b); // El valor de b no tiene sentido

        // A partir de java7, se pueden usar guiones para facilitar la lectura al
programador:
        System.out.println("Número: " + 1_000_000); // Salida en consola: Número: 1000000

    }

}
```

3. Números decimales

Los decimales se almacenan en los tipos **float** y **double**. Los decimales no son almacenados de forma exacta por eso siempre hay un posible error y se habla de precisión. Es mucho más preciso el tipo double que el tipo float.

Para asignar valores literales a una variable decimal, 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:

```
double d=3.49;//el literal 3.49 por defecto es double
float f=3.49f;//el literal 3.49 se tiene que convertir a float
```

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

```
int x=9.5; //error
int x=(int) 9.5;/*podemos mediante un casting pero perdemos los decimales,
                  es decir, x valdrá 9*/
```

El caso contrario sin embargo sí se puede hacer:

```java
int x=9;
double y=x; //correcto
```

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

4. Booleanos

Los valores booleanos o lógicos se almacenan en el tipo **boolean**. Sirven para indicar si algo es verdadero (**true**) o falso (**false**).

```java
boolean b=true;
boolean c=false;
```

Por otro lado, a diferencia del lenguaje C, no se puede 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:

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

5. Caracteres

Los valores de tipo carácter sirven para almacenar símbolos de escritura. En Java se puede almacenar cualquier código Unicode en el tipo **char**.

Los **literales** carácter van entre comillas simples, como por ejemplo: 'a'.

En programación, secuencias de escape es el conjunto de caracteres que en el código es interpretado con algún fin. En Java, la barra invertida `\` se denomina **carácter de escape**, el cual indica que el carácter puesto a continuación será convertido en carácter especial o, si ya es especial, dejará de ser especial. Por ejemplo, el carácter `n` no es especial pero con la `\` delante se convierte en especial ya que `\n` se interpreta como un salto de línea. La `\` es un carácter especial pero con otra `\` delante deja de ser especial y simplemente es una barra invertida.

En la siguiente tabla tenemos algunas secuencias de escape en Java:

Carácter	Significado
\t	Tabulador
\n	Salto de línea
\"	Dobles comillas
\'	Comillas simples
\\	Barra invertida

Carácter	Significado
\udddd	Representa el carácter Unicode cuyo código es representado por dddd en hexadecimal

Como vimos en el tema 1.1 Introducción, la descripción completa del estándar Unicode está disponible en la página web https://unicode.org/. En dicha página, encontramos las tablas de caracteres en hexadecimal. Para saber el código de los caracteres en decimal, podemos acceder al siguiente enlace: https://unicode-table.com/es/. Los caracteres imprimibles son del 32 al 126 y del 161 al 255.

Para insertar en el código caracteres no disponibles en el teclado, se hace de manera diferente según el Sistema Operativo:

- Linux: *Ctrl+Shift* y luego se pulsa *u*(para indicar que es Unicode) y el código Unicode en hexadecimal en el teclado numérico desactivado.
- Windows: *Alt* y el código Unicode en hexadecimal.

```
package tema1_4_TiposDeDatosPrimitivos;

public class Characters {

    public static void main(String[] args) {

        char character;

        character = 'C'; // Los literales carácter van entre comillas simples
        System.out.println(character);
        character = 67; // El código Unicode de la C es el 67. Esta línea hace lo mismo
que letra='C'
        System.out.println(character);
        character = '\u0043'; // El código Unicode de la C en hexadecimal es el 0043. Esta
línea hace lo mismo que letra='C'
        System.out.println(character);
        character = '\n'; // Carácter especial Nueva línea
        System.out.println(character);
        character = '\''; // Carácter especial Comillas simples
        System.out.println(character);
        character = '\"'; // Carácter especial Dobles comillas
        System.out.println(character);
        character = '\\'; // Carácter especial Barra inclinada
        System.out.println(character);
        character = 9752; // Código decimal del carácter trébol
        System.out.println(character);
        character = '\u2618';// Código hexadecimal del carácter trébol
        System.out.println(character);
        character = '☘';// Carácter trébol
        System.out.println(character);

    }

}
```

Si necesitamos almacenar más de un carácter, entonces debemos usar otro tipo de datos que nos permite manejar cadenas de caracteres: **String**.

En Java, las cadenas no se modelan como un dato de tipo primitivo, sino a través de la clase String. El texto es uno de los tipos de datos más importantes y por ello Java lo trata de manera especial. Para Java, las cadenas de texto son objetos especiales. Los textos deben manejarse creando objetos de tipo String.

Los **literales** cadena se escriben entre comillas dobles: `"Esto es un literal cadena"`.

Ejemplo:

```
String s="Estamos aprendiendo a programar";
```

En java existe también la **cadena vacía o nula ("")**, es decir, una cadena sin ningún carácter.
Ejemplo: `String s="";` A la variable *s* se le está asignando la cadena vacía o nula.

6. Conversión entre tipos (Casting)

Ya se ha comentado anteriormente la necesidad del 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;  //error
```

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

```
int a=1;
byte b;
b= (byte) a; //correcto
```

7. Errores

Otra de las ventajas grandes de Eclipse es que los errores aparecen en el código a medida que escribimos, al estilo de los procesadores de texto modernos como Word por ejemplo. Eso nos permite corregir errores de forma más eficiente.

El compilador muestra dos tipos de anomalías: errores y advertencias (warning). Un error es una condición que impide la obtención del programa final. Hasta que no se solventen todos los errores, el compilador no puede terminar su trabajo. Las advertencias son mensajes que muestra el compilador sobre situaciones especiales en las que se ha detectado una anomalía pero que, asumiendo ciertas condiciones, la traducción del programa continúa. Por tanto, la compilación de un programa no termina mientras tenga errores pero, sin embargo, se puede obtener un programa traducido aunque el compilador muestre advertencias.

En el eclipse, los errores en el código aparecen subrayados en rojo y en amarillo las advertencias.

Si un proyecto tiene algún archivo de clase con errores, el propio proyecto en el explorador de paquetes aparece marcado con una x roja. De ese modo sabremos que algún error existe en él. Si en lugar de error tenemos una advertencia, entonces aparece un triángulo amarillo. En el código aparecen esos mismos símbolos acompañados de una bombilla a la izquierda del número de línea.

Ejemplo de advertencia:

```
public static void main(String[] args) {
    int x = 3;//The value of the local variable x is not used
    x = 7;
}
```

Ejemplo de error:

```
public static void main(String[] args) {
     int x = 3;
     y = 7; //y cannot be resolved to a variable
}
```