# 1. Los tipos de datos.

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

El tipo limita el valor de la variable o expresión, las operaciones que se pueden hacer sobre esos valores, y el significado de esas operaciones. Esto es así porque un **tipo de dato** no es más que una **especificación de los valores que son válidos** para esa variable **, y de las operaciones que se pueden realizar con ellos.**

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

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

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

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

## 1.1. Tipos de datos primitivos I.

Los tipos primitivos son aquellos 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.

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

**Debes conocer**
En el siguiente enlace se muestran los tipos de datos primitivos en Java con el rango de valores que pueden tomar, el tamaño que ocupan en memoria y sus valores por defecto.

Tipos de Datos Primitivos en Java en ManualWeb

**Ver secuencias de escape:** 2.B.1 Secuencias de escape en cadenas o Strings

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

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

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

**Para saber más**

Si quieres obtener información sobre cómo se lleva a cabo la representación interna de números enteros y sobre la aritmética binaria, puedes consultar el siguiente enlace:

Enlace a una página de consulta sobre binario.

## 1.2. Tipos de datos primitivos II.

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

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

Entre cada dos números reales cualesquiera, siempre tendremos infinitos números reales, por lo que **la mayoría de ellos los representaremos de forma aproximada.** Por ejemplo, en la aritmética convencional, cuando dividimos 10 entre 3, el resultado es 3.3333333…, con la secuencia de 3 repitiéndose infinitamente. En el ordenador sólo podemos almacenar un número finito de bits, por lo que el almacenamiento de un número real será siempre una aproximación.
Los números reales se representan en coma flotante, que consiste en trasladar la coma decimal a la primera cifra significativa del valor, con objeto de poder representar el máximo de números posible.

Un número se expresa como:

$$Valor = mantisa * 2^{exponente}$$

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

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

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

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

## 1.3. Declaración e inicialización.

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

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

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

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

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

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

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

- Las **variables miembro** sí se inicializan automáticamente, si no les damos un valor. Cuando son de tipo numérico, se inicializan por defecto a 0, si don de tipo carácter, se inicializan al carácter `null` (\0), si son de tipo boolean se les asigna el valor por defecto `false`, y si son tipo referenciado se inicializan a `null`.

- Las **variables locales** no se inicializan automáticamente. Debemos asignarles nosotros un valor antes de ser usadas, ya que si el compilador detecta que la variable se usa antes de que se le asigne un valor, produce un error. Por ejemplo en este caso:

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

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

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

En el ejemplo anterior *la instrucción* `if` hace que si se cumple la condición que hay entre paréntesis (cualquiera que indiquemos), entonces el programa asignará el valor 5 a la variable `p`; sino se cumple la condición, `p` quedará sin inicializar. Pero si `p` no se ha inicializado, no tendría valor para asignárselo a `q`. Por ello, el compilador detecta ese posible problema y produce un error del tipo **"La variable podría no haber sido inicializada"**, independientemente de si se cumple o no la condición del `if`.

## 1.4. Tipos referenciados.

A partir de los ocho tipos datos primitivos, se pueden construir otros tipos de datos. Estos tipos de datos se llaman **tipos referenciados** o **referencias**, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador. También se les conoce como tipos de datos **estructurados,** se denominan así porque en su mayor parte están destinados a **contener múltiples valores de tipos más simples**, primitivos.

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

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

Como comentábamos al principio del apartado de Tipos de datos, cualquier aplicación de hoy en día necesita no perder de vista una cierta cantidad de datos. Cuando el conjunto de datos utilizado tiene características similares se suelen agrupar en estructuras para facilitar el acceso a los mismos, son los llamados **datos estructurados**. Son datos estructurados los **arrays, listas, árboles**, etc. Pueden estar en la memoria del programa en ejecución, guardados en el disco como ficheros, o almacenados en una base de datos. Dedicaremos, más adelante en el curso, varios temas a los datos estructurados.

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

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

Hemos visto qué son las variables, cómo se declaran y los tipos de datos que pueden adoptar. Anteriormente hemos visto un ejemplo de creación de variables, en esta ocasión vamos a crear más variables, pero de distintos tipos primitivos y los vamos a mostrar por pantalla. Los tipos referenciados los veremos en la siguiente unidad.

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

```
/* ejemplotipos.java
   Programa que crea variables de distintos tipos primitivos
   y los muestra por pantalla
*/
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
```

### 1.5. Tipos enumerados.

Los **tipos de datos enumerados** son una forma de declarar una variable con un *conjunto restringido de valores*. Por ejemplo, *los días de la semana*, *las estaciones del año, los meses, etc*. Es como si definiéramos nuestro propio tipo de datos.
La forma de declararlos es con la palabra reservada `enum`, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A los valores que se colocan dentro de las llaves se les considera como constantes, van separados por comas y deben ser valores únicos.

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

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

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.

```
/*
  tiposenumerados.java
  Ejemplo de Tipos enumerados
  @author FMA
*/
public class tiposenumerados {
    public enum Dias {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
    public static void main(String[] args) {
        // codigo de la aplicacion
        Dias diaactual = Dias.Martes;
        Dias diasiguiente = Dias.Miercoles;
        System.out.print("Hoy es: ");
        System.out.println(diaactual);
        System.out.println("Mañana\nes\n"+diasiguiente);
    } // fin main
} // fin tiposenumerados
```

En este ejemplo hemos utilizado el método `System.out.print`. Como podrás comprobar si lo ejecutas, la instrucción número 18 escribe el texto que tiene entre comillas pero no salta a la siguiente línea, por lo que el la instrucción número 19 escribe justo a continuación.

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

## 2. Literales de los tipos primitivos. 2.B.2 Literales primitivos

Un **literal**, **valor literal** o **constante literal** es un valor concreto para los tipos de datos primitivos del lenguaje, el tipo String o el tipo null.
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:

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

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

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

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

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

## 3. Trabajo con cadenas. 2.B.3 Operaciones con cadenas de caracteres

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

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

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

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

A continuación varios ejemplos de las distintas operaciones que podemos realizar concadenas de caracteres o String en Java: 2.B.4 formateo de salida con especificadores de formato

```
// @author FMA

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
```