# CONTENIDO

[Particularidades del lenguaje 2](#_Toc450549934)

[Tipos de datos en java. Tipos primitivos y Objeto 2](#_Toc450549935)

[Tipo de datos primitivos 2](#_Toc450549936)

[Clases de Envoltorio o Wrapper 3](#_Toc450549937)

[Valores por defecto de una variable 4](#_Toc450549938)

[Conversiones de tipo 4](#_Toc450549939)

[Conversiones implícitas 4](#_Toc450549940)

[Conversiones explícitas 5](#_Toc450549941)

[Constantes 6](#_Toc450549942)

[Métodos y campos estáticos 6](#_Toc450549943)

[Arrays 7](#_Toc450549944)

[Tipos Enumerados 8](#_Toc450549945)

# Particularidades del lenguaje

Antes de avanzar con los siguientes aspectos de la Programación Orientada a Objetos, nos gustaría compartir algunas características propias del lenguaje, que si bien no están asociadas necesariamente a conceptos propios de **OOP** (Oriented Object Programming) nos van a ayudar en la resolución de diferentes problemas.

## Tipos de datos en java. Tipos primitivos y Objeto

Los primeros lenguajes de programación no usaban objetos, solo variables. Una variable es un espacio de la memoria a la que asignamos un contenido, que puede ser un valor numérico (sólo números, con su valor de cálculo) o de tipo carácter o cadena de caracteres (valor alfanumérico que constará sólo de texto o de texto mezclado con números).

Como ejemplo podemos definir una variable a que contenga 32 y esto lo escribimos como a = 32. Posteriormente podemos cambiar el valor de a y hacer que a almacene el número 78 (a = 78). O hacer “a” equivalente al valor de otra variable “b” así: a = b.

### Tipo de datos primitivos

Toda información que se maneja en Java puede representarse por un objeto o por un tipo de dato básico o de tipo primitivo. Java soporta ocho tipos de datos primitivos, que son, a saber:

|  |  |
| --- | --- |
| **Tipo básico** | **Tamaño** |
| ***byte*** | 8 bits |
| ***short*** | 16 bits |
| ***int*** | 32 bits |
| ***long*** | 64 bits |
| ***char*** | 16 bits |
| ***float*** | 32 bits |
| ***double*** | 64 bits |
| ***boolean*** | \*\*\* |

En el caso del tipo primitivo *boolean* los dos únicos valores que puede asumir son true o false y a diferencia de otros lenguajes no existe una equivalencia entre estos valores y números enteros. En cuanto al tamaño en número de bits del tipo *boolean*, este dependerá de la máquina virtual.

En el caso de las cadenas de caracteres, no existe un dato primitivo asociado a ellas como en otros lenguajes, sino que Java trabaja con las cadenas de caracteres como si se tratara de un objeto, concretamente objetos de la clase *String.*[[1]](#footnote-1) De esto se desprende que además de usar tipos primitivos, al asignar un valor a una variable, es posible asignar un objeto.

Ahora bien, ¿en qué se diferencia una variable de un tipo primitivo de una variable de tipo objeto? En que las variables de tipos primitivos son entidades elementales: un número, un carácter, un valor verdadero o falso… mientras que los objetos son entidades complejas que pueden estar formadas por la agrupación de muchas variables y métodos.

En los programas en Java puede ser necesario tanto el uso de datos elementales como de datos complejos. Por eso en Java se usa el término “Tipos de datos” para englobar a aquello que ocupa un espacio de memoria y que puede ir tomando distintos valores o características durante la ejecución del programa.

Veamos los tipos de datos en Java sobre un esquema de síntesis:

|  |  |  |  |
| --- | --- | --- | --- |
| **TIPOS DE DATOS EN JAVA** | **TIPOS PRIMITIVOS** | Sin métodos, no son objetos, no necesitan una invocación para ser creados. | Ejemplo: *byte – short – int – long – float – double – char – boolean* |
| **TIPOS OBJETOS** | Con métodos, necesitan una invocación para ser creados. | Tipos de la Biblioteca estándar de Java: *String – TreeSet – Scanner - ArrayList* |
| Tipos definidos por el programador/usuario |
| Arrays: Serie de elementos de tipo vector o matriz. Se los considera una clase especial porque carecen de métodos. |
| Tipos de Envoltorio o wrapper: equivalentes a los tipos primitivos pero tratados como objetos : *Byte – Short – Integer – Long – Float – Double – Character – Boolean* |

### Clases de Envoltorio o Wrapper

Como puede desprenderse del cuadro anterior, para cada una de los tipos primitivos Java proporciona una clase que lo representa. A esas clases se las conoce como clases de envoltorio, y se diferencian en la nomenclatura de los tipos primitivos ya que estos se escriben en minúscula, en tanto que los nombres de las clases de envoltorio se inician con una letra mayúscula (como por convención se escriben todos los nombres de una clase). Veámoslo en un ejemplo:

**publicint**edad;

String cadena = **new** String("Hola");

Un objeto es distinto a un tipo primitivo, aunque “porten” la misma información. Es importante tener siempre presente que los objetos en Java tienen un tipo de tratamiento y los tipos primitivos, otro. ¿Para qué tener esa aparente duplicidad entre tipos primitivos y tipos envoltorio? Esto es una cuestión propia de la concepción del lenguaje de programación. Un tipo primitivo es un dato elemental y carece de métodos, mientras que un objeto es una entidad compleja y dispone de métodos. Por otro lado, de acuerdo con la especificación de Java, es posible que necesitemos utilizar dentro de un programa un objeto que “porte” como contenido un número entero. Desde el momento en que sea necesario un objeto habremos de pensar en un envoltorio, por ejemplo *Integer.* Inicialmente puede costar distinguir cuándo usar un tipo primitivo y cuándo un envoltorio en situaciones en las que ambos sean válidos.

### Valores por defecto de una variable

El valor por defecto que asume una variable cuando es inicializada en forma automática se conoce como valor por defecto o predeterminado y depende del tipo de variable. Esta inicialización por defecto corresponde sólo a las variables de tipo privado (ejemplo, los campos o tributos); en caso de tratarse de variables locales (dentro de un bloque) es necesario inicializarlas antes de ser utilizadas en alguna instrucción del programa.

Según el tipo de variable los valores que asume por defecto son:

* 1. Numéricos enteros (*byte, short, int, long*) : 0
  2. Carácter (*char*): ‘/u0000’
  3. Numéricos decimales (*float, double*): 0.0
  4. Lógicos: false
  5. Objetos: null

Los tipos no primitivos corresponden a objetos creados a partir de una clase y por ser objetos asume como valor por defecto null.

**void** método(){

int n;

n=n + 1 // error de compilación

}

Es importante destacar la diferencia que existe entre realizar una operación de asignación entre variables del tipo objeto y variables de tipo primitivo. En el caso del tipo primitivo,, por ejemplo un *int*, esta operación implica que el dato contenido en una variable se copia en otra. En el caso en que se trate de un tipo objeto debemos recordar que lo que es está copiando es una referencia al objeto, no el objeto. Por lo tanto, al asignar el valor de una variable de tipo objeto a otra lo que se está copiando es la referencia al objeto, no el objeto. De esta manera, no tenemos dos copias del objeto, sino un único objeto referenciado por dos variables.

## Conversiones de tipo

Java es un lenguaje fuertemente tipado, o sea estricto a la hora de asignar valores a las variables. De acuerdo a esta característica, el compilador sólo admite asignar a una variable un dato del tipo declarado en la variable, aunque, en ciertas circunstancias permite realizar conversiones para almacenar en una variable un dato de tipo diferente al declarado. En java es posible realizar conversiones entre todos los tipos básicos, con excepción de *boolean*. Las conversiones pueden realizarse de dos maneras: en forma implícita o explícita.

### Conversiones implícitas

Las conversiones implícitas son las que se realizan en forma automática a través del compilador, antes de almacenarlo en una variable. Ejemplo: un dato de tipo *byte* es almacenado en la variable b y luego es convertido a *int* al asignarlo a la variable i:

int i;

byte b=30;

i=b;

Para que una conversión pueda realizarse en forma automática o sea en forma implícita, es necesario que el tipo de variable destino sea de tamaño igual o superior al tipo de origen, aunque como toda regla presenta excepciones:

* Cuando la variable destino es entera y el origen es decimal (*float o double*) la conversión no puede ser automática.
* Cuando la variable destino es char y el origen es numérico, independientemente del tipo específico, la conversión no puede ser automática.

int k=5, p;

short s=10;

char c=’ñ’;

float h;

p=c; // conversión implícita de char a int;

h=k; // conversión implícita de int a float;

h=s; // conversión implícita de short a int;

Ejemplos de conversiones implícitas que darían lugar a error:

int n;

long c=20;

float ft=2.4f;

char k;

byte s=4;

n=c; // error, conversión implícita de long a int;

k=s; // error, conversión implícita de byte a char;

n=ft; //error, conversión implícita de float a int;

### Conversiones explícitas

En caso en que no se cumplan las condiciones para una conversión implícita, ésta puede realizar se en forma explícita utilizando la expresión:

***variable\_destino=(tipo\_destino)dato\_origen*;**

Con esta expresión se le indica al compilador que convierta *dato\_origen* a *tipo\_destino* para que puede ser almacenado en la variable destino. A esta operación se la denomina casting o estrechamiento ya que al convertir un dato de un tipo en otro de tipo inferior se realiza una reducción, un estrechamiento que en algunos casos puede llevar a una pérdida de datos o precisión, pero que evitaría posibles errores de ejecución. Ejemplos:

char c;

byte k;

int p=400;

double d=34.6;

c=(char)d; // se elimina la parte decimal (truncado)

k=(byte)p; // se produce una pérdida de datos, pero la //conversión es posible;

En el caso de los objetos no es posible realizar conversiones, pero es posible asignar a un objeto de una clase a una variable de clase diferente. Esto será posible solamente en una relación de herencia entre clases.

## Constantes

Una constante es una variable cuyo valor no puede ser modificado Para definir una constante en Java se utiliza la palabra reservada final, delante de la declaración del tipo, de acuerdo a la siguiente expresión:

final tipo nombre\_constante=valor;

Ejemplo:

final double pi=3.1416;

Una constante se puede declarar en los mismos lugares que una variable: al principio de una clase o al interior de un método, en forma local. Existen constantes que pueden ser utilizadas desde el exterior de la clase donde se han declarado; para que además no sea necesario crear objetos de la misma para hacer uso de esas constantes, se las declara como campos públicos y estáticos. Este es el caso de la constante PI, declarada en la clase Math. Su notación en ese caso sería:

public static final double PI=3.1416;

## Métodos y campos estáticos

Una variable *static* representa información en toda la clase (todos los objetos de la clase comparten el mismo dato). La declaración de una variable static comienza con la palabra clave *static*. Ahora bien, un método declarado como *static* no puede tener acceso a los miembros no *static* de una clase, ya que un método *static* puede llamarse aún cuando no se hayan creado instancias de objetos de la clase. Por la misma razón, esta referencia *this* no puede usarse en un método *static*; debe referirse a un objeto especíﬁco de la clase, y a la hora de llamar a un método *static*, podría no haber objetos de su clase en la memoria. La referencia *this* se requiere para permitir a un método de una clase acceder a otros miembros no *static* de la misma clase.

La clase **Math** cuenta con métodos *static* para realizar cálculos matemáticos comunes; además, declara dos campos que representan constantes matemáticas de uso común: Math.PI y Math.E. La constante Math.PI (3.14159265358979323846) es la relación entre la circunferencia de un círculo y su diámetro. La constante Math.E (2.7182818284590452354) es el valor de la base para los logaritmos naturales (que se calculan con el método *static* Math log). Math.PI y Math.E se declaran con los modiﬁcadores *public*, *ﬁnal* y *static*. Al hacerlos *public*, otros programadores pueden usar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave *ﬁnal* es constante; su valor no se puede modificar una vez que se inicializa el campo. Tanto PI como E se declaran *ﬁnal,* ya que sus valores nunca cambian. Al hacer a estos campos *static*, se puede acceder a ellos a través del nombre de la clase **Math** y un separador punto (.), justo igual que con los métodos de la clase **Math**.

La sintaxis para incluir atributos o métodos estáticos en una clase es la siguiente:

**static tipo** campon;

**static tipo** métodoX(parámetros)

{

//codigo métodoX

}

A diferencia de los atributos en general que suelen ser de tipo privados, los atributos estáticos suelen llevar el modificador de acceso *public(*o *protected* o ninguno) lo cual permite que puedan ser accedidos por afuera de la clase. **Dado que el método estático no hace referencia a ningún objeto en particular, no puede hacer referencia a campos y métodos que si dependan de un objeto.**

A continuación se detalla el código de la clase Alumno donde puede observarse un atributo de tipo estático llamado CANTIDAD\_DE\_CLASES\_BRINDADAS, inicializado en 0 y una constante de tipo estático llamada CANTIDAD\_DIAS\_DE\_CLASE, inicializada en 20. Mientras el primero es de tipo *private*, la constante estática es de tipo *public*, por lo cual puede ser accedida en forma directa desde fuera de la clase Alumno. Luego puede observarse es el método habilitarDiaDeClase, de tipo estático y público que permite incrementar la variable CANTIDAD\_DE\_CLASES\_BRINDADAS. Es importante observar que para llamar a la variable no se utiliza la palabra reservada *this* sino a través del nombre de la clase, por los motivos que se explicaron en párrafos anteriores.

**public** **class** Alumno {

**private** **static** **int** *CANTIDAD\_DE\_CLASES\_BRINDADAS* = 0;

**public** **static** **final** **int** ***CANTIDAD\_DIAS\_DE\_CLASES*** = 20;

**public** **static** **void** habilitarDiaDeClase(){

Alumno.*CANTIDAD\_DE\_CLASES\_BRINDADAS*++;

}

…

## Arrays

En Java, un arreglo o *array* es un grupo de variables (llamadas elementos o componentes) que contienen valores, todos del mismo tipo. Los arreglos son objetos (pertenecen al grupo de tipos objeto), por lo que se consideran como tipos de referencia. Cada uno de los elementos del *array* tiene asignado un índice numérico según su posición, siendo 0 el índice del primero. La sintaxis para declarar una *array* es la siguiente:

int variable\_array[]; o bien int[] vatiable\_array;

Ejemplos:

int[ ] k;

String [ ] p;

char cads[ ];

Los arrays pueden declararse como atributos de una clase o como variables locales, al ser una variable de tipo objeto se inicializa implícitamente con valor *null*. En caso en que el array se encuentre dimensionado o sea que se indique la cantidad de posiciones con que cuenta el array, todos sus elementos son inicializados explícitamente al valor por defecto del tipo correspondiente, independientemente de que el array corresponda a un atributo o a una variable local. Para dimensionar un array se utiliza la siguiente expresión:

variable\_array= new tipo [tamaño];

Ejemplos:

K=new int[5];

Cads=new char[10];

String [] nombres= new String [10];

Existe una forma de declarar, dimensionar e inicializar un array en una misma sentencia. La siguiente instrucción crea un array de tres enteros y los inicializa en los valores indicados entre llaves:

int [ ]nums= {10,20,30};

Todos los objetos array exponen un atributo público ***length*** que permite conocer el tamaño del array. Este atributo resulta muy útil al querer recorrer el array. En el próximo ejemplo se puede observar cómo se utiliza *length* para recorrer un array y cargarlo con números enteros pares consecutivos, empezando por el 0.

int[ ] nums=new int [10];

for (int i=0; i<nums.length; i++){

nums[i] = i\*2;

}

## Tipos Enumerados

Los tipos numerados constituyen una nueva característica incluida en la versión 5 de J2SE, que permite definir nuevos tipos de datos cuyos posibles valores están limitados a un determinado conjunto dado, o sea a un conjunto de valores constantes. En versiones anteriores de Java, la forma de definir un determinado conjunto de valores mediante la utilización de constantes, como se muestra en este ejemplo del estado final de un alumno después de una cursada:

**public class** Estado {

**public static final int** INSCRIPTO = 0;

**public static final int** CURSANDO = 1;

**public static final int** PROMOCIONADO = 2;

**public static final int** PENDIENTE\_DE\_RENDICION\_DEL\_FINAL = 3;

**public static final int** APROBADO = 4;

**public static final int** AUSENTE = 5;

**public static final int** DESAPROBADO = 6;

}

De esta forma, para referirse a alguno de los valores definidos en las constantes anteriores se utilizaría la expresión:

Nombre\_clase.CONSTANTE, por ejemplo Estado.APROBADO

Este sistema es inseguro, ya que nada impide que se asigne a través de un método setter un nuevo valor de estado a la clase. Gracias a la incorporación de los tipos numerados, es posible definir un conjunto de valores que puede almacenar una variable de este tipo, generando un error de compilación cualquier intento de querer asignar un valor no definido en la enumeración. Un tipo numerado se define según el siguiente formato :

[public] enum Nombre\_tipo {VALOR1,VALOR2,…}

siendo Nombre\_tipo el nombre que se va a signar al tipo numerado y VALORN los posibles valores que puede asumir. La declaración de tipo numerado corresponde a las clases nunca puede estar incluida dentro de un método. Para el ejemplo anterior, el tipo numerado correspondiente sería el siguiente:

**public** **enum** Estado{***INSCRIPTO***, ***CURSANDO***, ***PROMOCIONADO***, ***PENDIENTE\_DE\_RENDICION\_DEL\_FINAL***, ***APROBADO***, ***AUSENTE***, ***DESAPROBADO***};

Una enumeración es un tipo especial de clase que hereda de **java.lang.Enum**. A diferencia de las clases estándar, una clase de enumeración no permite el uso del operador *new* para la creación de objetos, dado que cada uno de los valores de la enumeración representan uno de los posibles objetos de la clase, los que fueron creados en forma implícita al declarar la enumeración. Además de los métodos heredados de la clase Enum, todas las enumeraciones disponen del método estático values(), que devuelve un array con todos los objetos de la clase.

1. Una cadena de caracteres es un objeto. El tipo *String* en Java nos permite crear objetos que contienen texto (palabras, frases, etc.). El texto debe ir siempre entre comillas. Muchas veces se cree erróneamente que el tipo *String* es un tipo primitivo por analogía con otros lenguajes donde *String* funciona como una variable elemental. En Java no es así. [↑](#footnote-ref-1)