**Variables y tipos de datos en Java**

Tipos de datos primitivos en Java

Java cuenta con un pequeño conjunto de tipos de datos primitivos. Podríamos considerarlos fundamentales, ya que la mayor parte de los demás tipos, los tipos estructurados o complejos, son composiciones a partir de estos más básicos. Estos tipos de datos primitivos sirven para gestionar los tipos de información más básicos, como números de diversas clases o datos de tipo verdadero/falso.

De estos tipos primitivos, ocho en total, seis de ellos están destinados a facilitar el trabajo con números. Podemos agruparlos en dos categorías: tipos numéricos enteros y tipos numéricos en punto flotante. Los primeros permiten operar exclusivamente con números enteros, sin parte decimal, mientras que el segundo grupo contempla también números racionales o con parte decimal.

* Tipos numéricos enteros

En Java existen cuatro tipos destinados a almacenar números enteros. La única diferencia entre ellos es el número de bytes usados para su almacenamiento y, en consecuencia, el rango de valores que es posible representar con ellos. Todos ellos emplean una representación que permite el almacenamiento de números negativos y positivos. El nombre y características de estos tipos son los siguientes:

1. **byte**: como su propio nombre denota, emplea un solo byte (8 bits) de almacenamiento. Esto permite almacenar valores en el rango [-128, 127].
2. **short**: usa el doble de almacenamiento que el anterior, lo cual hace posible representar cualquier valor en el rango [-32.768, 32.767].
3. **int**: emplea 4 bytes de almacenamiento y es el tipo de dato entero más empleado. El rango de valores que puede representar va de -231 a 231-1.
4. **long**: es el tipo entero de mayor tamaño, 8 bytes (64 bits), con un rango de valores desde -263 a 263-1.

* Tipos numéricos en punto flotante

Los tipos numéricos en punto flotante permiten representar números tanto muy grandes como muy pequeños además de números decimales. Java dispone de 2 tipos concretos en esta categoría:

1. **float**: conocido como tipo de precisión simple, emplea un total de 32 bits. Con este tipo de datos es posible representar números en el rango de 1.4x10-45 a 3.4028235x1038.
2. **double**: sigue un esquema de almacenamiento similar al anterior, pero usando 64 bits en lugar de 32. Esto le permite representar valores en el rango de 4.9x10-324 a 1.7976931348623157x10308.

* Booleanos y caracteres

1. **boolean**: tiene la finalidad de facilitar el trabajo con valores "verdadero/falso" (booleanos), resultantes por regla general de evaluar expresiones. Los dos valores posibles de este tipo son true y false.
2. **char**: se utiliza para almacenar caracteres individuales (letras, para entendernos). En realidad está considerado también un tipo numérico, si bien su representación habitual es la del carácter cuyo código almacena. Utiliza 16 bits y se usa la codificación UTF-16 de Unicode.

* Tipos de datos estructurados

Los tipos de datos primitivos que acabamos de ver se caracterizan por poder almacenar un único valor. Salvo este reducido conjunto de tipos de datos primitivos, que facilitan el trabajo con números, caracteres y valores booleanos, todos los demás tipos de Java son objetos, también llamados tipos estructurados o "Clases".

Los tipos de datos estructurados se denominan así porque en su mayor parte están destinados a contener múltiples valores de tipos más simples, primitivos. También se les llama muchas veces "tipos objeto" porque se usan para representar objetos. Puede que te suene más ese nombre.

* Cadenas de caracteres

Aunque las cadenas de caracteres no son un tipo simple en Java, sino una instancia de la clase String, el lenguaje otorga un tratamiento bastante especial a este tipo de dato, lo cual provoca que, en ocasiones, nos parezca estar trabajando con un tipo primitivo.

Aunque cuando declaramos una cadena estamos creando un objeto, su declaración no se diferencia de la de una variable de tipo primitivo.

Las cadenas de caracteres se delimitan entre comillas dobles, en lugar de simples como los caracteres individuales. En la declaración, sin embargo, no se indica explícitamente que se quiere crear un nuevo objeto de tipo String, esto es algo que infiere automáticamente el compilador.

Las cadenas, por tanto, son objetos que disponen de métodos que permiten operar sobre la información almacenada en dicha cadena. Así, encontraremos métodos para buscar una subcadena dentro de la cadena, sustituirla por otra, dividirla en varias cadenas atendiendo a un cierto separador, convertir a mayúsculas o minúsculas, etc.

* Vectores o arrays

Los vectores son colecciones de datos de un mismo tipo. También son conocidos popularmente como arrays e incluso como "arreglos" (aunque se desaconseja esta última denominación por ser una mala adaptación del inglés).

Un vector es una estructura de datos en la que a cada elemento le corresponde una posición identificada por uno o más índices numéricos enteros.

Los elementos de un vector o array se empiezan a numerar en el 0, y permiten gestionar desde una sola variable múltiples datos del mismo tipo.

Por ejemplo, si tenemos que almacenar una lista de 10 números enteros, declararíamos un vector de tamaño 10 y de tipo entero, y no tendríamos que declarar 10 variables separadas de tipo entero, una para cada número.

* Tipos definidos por el usuario

Además de los tipos estructurados básicos que acabamos de ver (cadenas y vectores) en Java existen infinidad de clases en la plataforma, y de terceros, para realizar casi cualquier operación o tarea que se pueda ocurrir: leer y escribir archivos, enviar correos electrónicos, ejecutar otras aplicaciones o crear cadenas de texto más especializadas, entre un millón de cosas más.

Todas esas clases son tipos estructurados también.

Y por supuesto puedes crear tus propias clases para hacer todo tipo de tareas o almacenar información. Serían tipos estructurados definidos por el usuario.

* Tipos envoltorio o wrapper

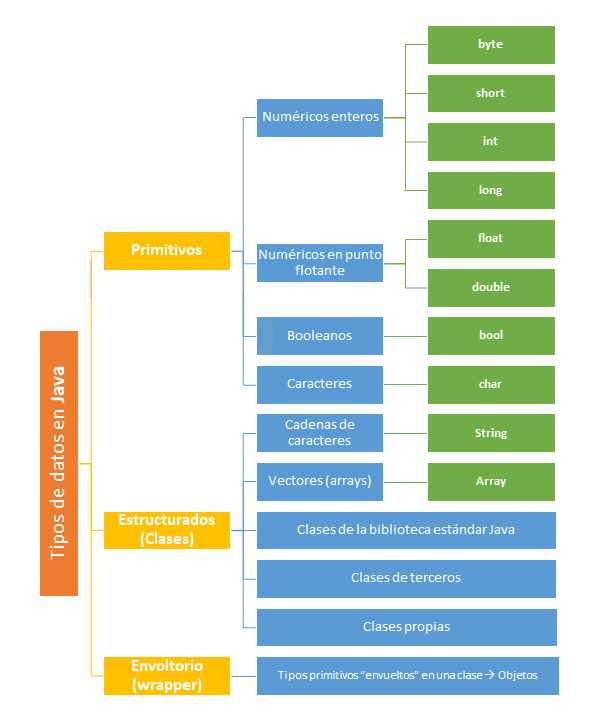
Java cuenta con tipos de datos estructurados equivalentes a cada uno de los tipos primitivos que hemos visto.

Así, por ejemplo, para representar un entero de 32 bits (int) de los que hemos visto al principio, Java define una clase llamada Integer que representa y "envuelve" al mismo dato pero le añade ciertos métodos y propiedades útiles por encima.

Además, otra de las finalidades de estos tipos "envoltorio" es facilitar el uso de esta clase de valores allí donde se espera un dato por referencia (un objeto) en lugar de un dato por valor (para entender la diferencia entre tipos por valor y tipos por referencia lee este artículo. Aunque está escrito para C#, todo lo explicado es igualmente válido para Java).

Estos tipos equivalentes a los primitivos pero en forma de objetos son: Byte, Short, Integer, Long, Float, Double, Boolean y Character (8 igualmente).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **DE DATOS EN JAVA** |  | **NOMBRE** | **TIPO** | **OCUPA** | **RANGO APROXIMADO** |
| **TIPOS PRIMITIVOS** | byte | Entero | 1 byte | -128 a 127 |
| (sin métodos; no son objetos; no necesitan una invocación para ser creados) | short | Entero | 2 bytes | -32768 a 32767 |
|  | int | Entero | 4 bytes | 2\*109 |
|  | long | Entero | 8 bytes | -263 a 263-1 |
|  | float | Decimal simple | 4 bytes | 1. 1.4x10-45 a 3.4028235x1038. |
|  | double | Decimal doble | 8 bytes | 4.9x10-324 a 1.7976931348623157x10308 |
|  | char | Carácter simple | 2 bytes | --- |
|  | boolean | Valor true o false | 1 byte | --- |
|  |  |  |  |  |
| **TIPOS OBJETO** | Tipos de la biblioteca estándar de Java | String (cadenas de texto) | | |
| (con métodos, necesitan una invocación para ser creados) | Muchos otros (p.ej. Scanner, TreeSet, ArrayList…) | | |
|  | Tipos definidos por el programador / usuario | Cualquiera que se nos ocurra, por ejemplo Taxi, Autobus, Tranvia | | |
|  | arrays | Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos. | | |
|  | Tipos envoltorio o wrapper (Equivalentes a los tipos primitivos pero como objetos.) | Byte | | |
|  | Short | | |
|  | Integer | | |
|  | Long | | |
|  | Float | | |
|  | Double | | |
|  | Character | | |
|  | Boolean | | |



**Casteo** (casting): es un procedimiento para transformar una variable primitiva de un tipo a otro, o transformar un objeto de una clase a otra clase siempre y cuando haya una relación de herencia entre ambas.

**Conversión de tipo Implícita**

Se realiza de forma automática al asignar un valor de un tipo a otro compatible. Ejemplo:

byte x = 5;  
int y = x;

El valor almacenado en x (byte) se convertirá a entero al asignarse a y (int).

**Conversión de tipo Explícita**

Tu tarea como programador es de especificar el nuevo tipo al que se va a transformar el dato. Se escribe de forma explicita entre paréntesis delante del dato. Ejemplo:

byte a = 20;  
int x = (int) a;

Al escribir int entre paréntesis se fuerza a cambiar el dato de tipo byte a int.

**String a Integer**

Integer entero = Integer.valueOf(cadena);

o

int entero = Integer.parseInt(cadena);

Ejemplo:

Integer a = Integer.valueOf("900");

o

int b = Integer.parseInt("900");

**Integer a String**

String cadena = Integer.toString(entero);

o

String cadena = String.valueOf(entero);

Ejemplo:

String importe = Integer.toString(900);

o

int entero = 900;

String mensaje = String.valueOf(entero);

**char a String**

String cadena = Character.toString(char);

Ejemplo:

char codigo = 'A';

String cadena = Character.toString(codigo);

**String a char**

char caracter = cadena.charAt(0); //Solo primer caracter

Ejemplo:

String codigo= "E";

char caracter = cadena.charAt(0);

**String a Double**

double doble = Double.parseDouble(cadena);

Ejemplo:

double doble = Double.parseDouble("900.1");

**Double a String**

String cadena = String.valueOf(doble);

Ejemplo:

double totalDoble = 900.5;

String totalString = String.valueOf(doble);

**String a Float**

float flotante = Float.parseFloat(cadena);

Ejemplo:

float importe = Float.parseFloat("900.5");

**Float a String**

String cadena = Float.toString(flotante);

Ejemplo:

String total = Float.toString(900.1f);

**String a Boolean**

Boolean boolean = Boolean.valueOf(cadena);

o

boolean boolean = Boolean.parseBoolean(cadena);

Ejemplo:

Boolean boolean = Boolean.valueOf("true");

o

boolean boolean = Boolean.parseBoolean("false");

**Boolean a String**

String cadena = String.valueOf(b);

o

String cadena = Boolean.toString(b);

Ejemplo:

boolean b = true;

String cadena = String.valueOf(b);

o

boolean b = false;

String cadena = Boolean.toString(b);