Capítulo 2. Variables y tipos de datos

A. J. Pérez

Variables y tipos de datos

Noción de tipo de dato

Los *tipos de datos* son un mecanismo que determina las características atribuidas a las *expresión* y *variables* de un programa permitiendo predecir su comportamiento. Este capítulo trata sobre *los tipos de datos* principales de Java, mostrando cómo declarar variables, asignar valores y combinar tipos en expresiones.

Identificadores

Los *identificadores* se utilizan para establecer los nombres de las *palabras reservadas,* los *paquetes*, los nuevos *tipos de datos* (*clases* habitualmente), los *métodos,* las *variables y las constantes* con nombre. **Establecen el nivel léxico-gráfico del lenguaje**.

Se recomienda que los *identificadores* se formen **secuencia de letras en mayúscula o minúscula del alfabeto anglosajón, números y los caracteres subrayado (_) , signo de dólar ($)** . **No se admite que un identificador empiece por número.**

Hay que tener en cuenta que:

1. **Java es sensible a las mayúsculas/minúsculas**, lo que significa que:
    - `VALOR` es diferente de `Valor` y de `valor`.
2. **Java admite cualquier *carácter Unicode* como parte de un identificador** a condición de que no tenga un uso especial en el lenguaje (generalmente operadores y delimitadores). No se recomienda.
3. Hay recomendaciones de **estilo en la comunidad de programadores Java** por el que algunos identificadores aun siendo correctos no serían recomendable. Ver la sección correspondiente en: Anexo 2. Convenciones de escritura de código fuente Java

Palabras reservadas

Las palabras reservadas son *identificadores especiales* que el lenguaje Java utiliza de forma exclusiva y con significados especiales dentro de los programas. Se utilizan para:

- Organizar la estructura de los programas.
- Manejar e identificar los tipos de datos primitivos.
- Controlar y manipular objetos.
- Controlar y modificar el acceso a datos en múltiples contextos.
- Controlar la secuencia de ejecución.

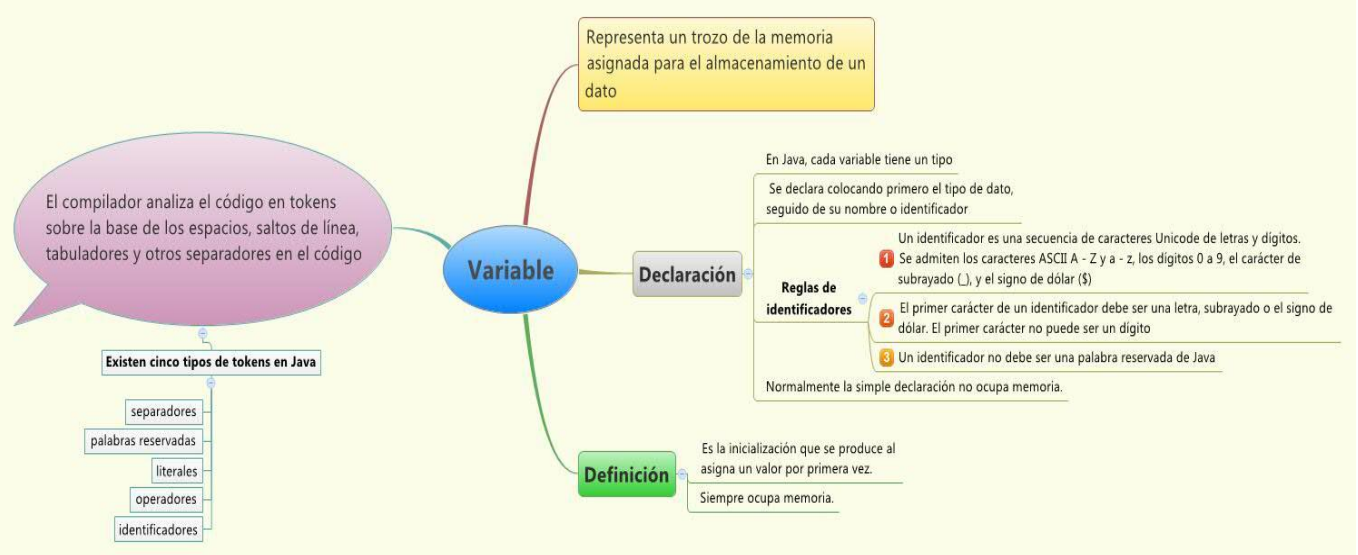
Estas palabras reservadas sólo se pueden utilizar para su propósito original y **no son admitidas como identificadores para otro fin.**

abstract	do	implements	protected	throws
assert [4]	double	import	public	transient
boolean	else	instanceof	requires [6]	true [2]
break	enum [5]	int	return	try
byte	extends	interface	short	var [1,7]
case	exports [6]	long	static	void
catch	false [2]	module [6]	strictfp [3]	volatile
char	finally	native	super	while
class	float	new	switch	throws
const [1]	for	null [2]	synchronized	
continue	goto [1]	package	this	
default	if	private	throw	

1 *no usada*
2 *asociada a valores literales*
3 *desde Java 1.2*
4 *desde Java 1.4*
5 *desde Java 5.0*
6 *desde Java 9.0*
7 *desde Java 10.0*

Variables

Las variables son los elementos básicos de un programa para el manejo de los datos de manera flexible, genérica y efectiva. Una variable se define mediante la combinación de un *identificador*, un *tipo* y un *ámbito*.



Declaración de una variable

La forma básica de la declaración de una variable es:

tipo `identificador [ = valor ] [, identificador [ = valor ] ...]` **`;`**

El *tipo* puede ser:

- `byte, short, int, long`
- `float, double`
- `char, String`
- `boolean`
- Nombre de una **class**, **interface** o **enum**. Estos tres conceptos serán descritos más adelante.

Ámbito de una variable

Los bloques son un mecanismo del lenguaje para delimitar la visibilidad de las variables. Los bloques de instrucciones en Java se delimitan con dos llaves { }. Las variables de Java sólo son *visibles* dentro del bloque donde han sido declaradas correspondiente.
Los bloque de instrucciones se pueden anidar y cada uno puede contener su propio conjunto de *declaraciones de variables locales*. Si en un ámbito local anidado, se declara una variable con el mismo nombre que una de un ámbito superior; prevalece el acceso a la variable más local.

Inicialización de variables

Una vez declarada una variable, se puede establecer su valor asignando, con el operador = , un *literal* u otra variable.

```
int num1 = 2;       // Uso de un literal
int num2;
int num3;

num2 = num1;        // Uso de una variable ya definida

// La siguiente instrucción asigna 5 en cascada
num3 = num1 = 5;    // No se recomienda
```

Desde el punto de vista del mecanismo de la asignación, y lo que ocurre en la memoria del sistema, en Java hay dos grandes grupos de datos:

- los que se asignan siempre por valor (primitivos)
- Los que se asignan siempre como referencias (objetos)

Para cualquier variable de cualquier tipo de dato Java establece un *valor por defecto* cuando son declarados; esto está especificado en el comportamiento previsto del lenguaje.

Grupo	Tipo de datos	Valor por defecto
Primitivos	byte	0
	short	0
	int	0
	long	0L
	float	0.0f
	double	0.0d
	char	'\u0000'
	boolean	false
Referencias	String	null
	Object	null

Un ejemplo:

```
String nombre = new String();    // equivale a name = "";
String nombre;                   // equivale a name = null;
```

Primitivos vs. referencias

En realidad **toda asignación** con el operador **= siempre se resuelve como una copia del operando de la derecha**; lo que ocurre es que **si es un dato primitivo se clona su valor literal**, mientras que **si es un objeto se duplica su referencia**, –equivalente a una dirección de la memoria- **no el contenido** –valores- de esa posición de memoria. Esto tiene importantes consecuencias en la manera de manejar los datos almacenados.

Los tipos primitivos -los manejados por valor- se almacenan en la *pila del programa*; y contienen directamente un valor literal. Todas las variables de tipo primitivo son siempre de tamaño 1, 2, 4 u 8 *bytes* en la *pila* y se liberan al salir del ámbito de visibilidad.

Los objetos -los manejados siempre por referencia- se almacenan en dos partes:

1. **En el *stack (la pila del programa)* se almacena la dirección de memoria donde se localiza el objeto.**
2. **En el *heap (el montón del programa)*es donde se almacenan realmente el objeto.**

El *stack* almacena una lista de referencias (direcciones de memoria) que localizan la ubicación real de los objetos en el *heap*. Un tipo referencia sólo puede apuntar a un único objeto del *tipo declarado* o pueden tener un valor `null` que indica que no tiene asociado espacio en el *heap*; se dice también que es un objeto *nulo*.

Los tipos referencia utilizan memoria dinámica del *heap;* cuando se detecta que ya no son utilizados por el programa, la memoria es liberada por el *recolector de basura* (*garbage collector*). Dado que la asignación y liberación de la memoria dinámica es una sobrecarga para la máquina virtual Java, se puede decir que **los tipos referencia son más lentos que los primitivos manejados directamente por valor**.

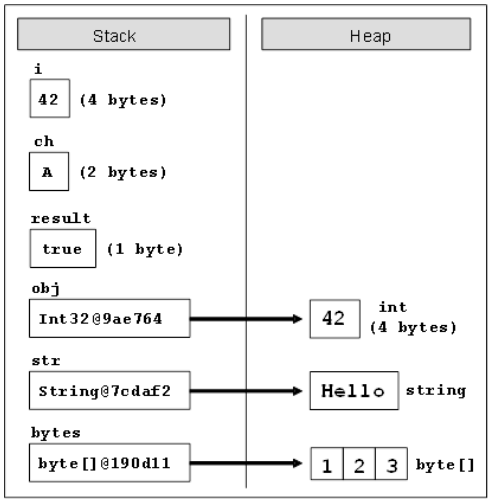
Los tipos referencia son siempre objetos generados a partir de las *class*, *enum* e *interface*; también son referencias los *arrays* de cualquier tipo.

Primitivos y tipos referencia en la memoria

Para entender la diferencia entre los *tipos primitivos* y los *tipos referencia* desde el punto de vista de cómo se almacenan en memoria se puede ver cómo se ejecutaría el siguiente código:

```
int i = 42;
char ch = 'A';
boolean result = true;
Object obj = (Integer) 42;
String str = "Hello";
byte[] bytes = {1, 2, 3};
```

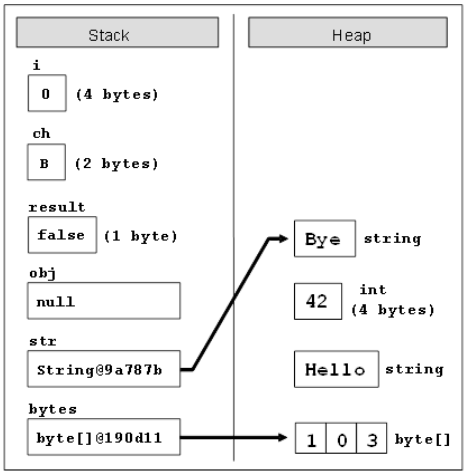
Las variables se crean en la memoria de la siguiente manera:



Si se ejecuta el siguiente código, que modifica los valores de las variables, se ve lo que ocurre con la memoria:

```
i = 0;
ch = 'B';
result = false;
obj = null;
str = "Bye";
bytes[1] = 0;
```

Queda:



Como se puede ver en la figura, en el caso de un tipo primitivo `i = 0;` cambia su valor directamente en la pila.

En el caso de un tipo referencia, modifica directamente su valor en la memoria dinámica `bytes[1] = 0;` y la variable que contiene la referencia se mantiene sin cambios `B@190d11.`

Al ejecutar `obj = null;` se anula la correspondiente referencia y se desconecta de su anterior valor –el espacio ocupado por 42 será liberado posteriormente- .

Al asignar un nuevo valor al objeto `str = "bye"`, se le asigna nuevo sitio dinámico, y el antiguo queda libre y su memoria será liberada en cualquier momento por el *garbage collector*.

Tipos primitivos

Java, que es un lenguaje de programación orientado a objetos, renunció -por cuestiones de eficiencia- al principio de uniformidad de que *todo es un objeto*. El rendimiento del sistema fue una meta fundamental en la creación de Java. Esta decisión condujo a la utilización de **tipos primitivos de Java que no están orientados a objeto, siendo análogos a los tipos simples de la mayoría de los otros lenguajes que no utilizan objetos.**

Los tipos primitivos representan datos simples (sin estructura), de un único valor, como números enteros, de punto o coma flotante, caracteres y valores booleanos. **Parte de la seguridad y robustez de Java viene del hecho de que Java es un lenguaje *fuertemente tipado*. Cada expresión tiene un tipo**. **Cada variable tiene un tipo y cada tipo está definido estrictamente**.

En todas las asignaciones de expresiones a variables, tanto explícitas como a través de paso de parámetros en llamadas a método, se comprueba la compatibilidad de los tipos. El compilador de Java comprueba estáticamente todo el código que compila para asegurar que los tipos sean correctos. Si no coinciden los tipos, se generan errores de compilador que se deben corregir antes de poderse completar la compilación de la clase. **Cada tipo tiene definido explícitamente un único conjunto de valores que puede expresar, junto con un conjunto definido de operaciones que están permitidas.** Por ejemplo, los tipos numéricos `int` y `float` se pueden sumar entre

ellos, pero los tipos `boolean` , no.

Números enteros

Los tipos numéricos son los previstos para manejar información numérica que requiere tratamiento aritmético. Los tipos numéricos pueden tener dos formatos de representación, los que guardan números de valor completo sin parte fraccionaria, llamados *enteros*, y los que tienen una parte fraccionaria, llamados números *reales*. La magnitud del rango o el grado de precisión de la componente fraccionaria necesaria dependerá de su aplicación.

Todos estos tipos en Java tienen definido un rango explícito y un comportamiento matemático. La mayor parte del código no portable de otros lenguajes está repleto de problemas debidos al comportamiento no especificado del tipo entero. El origen de este problema es en un concepto llamado *tamaño de palabra de la máquina*. El tamaño de palabra de una CPU dada, está determinado por el *número de bits* que utiliza internamente en sus registros más básicos, que se utilizan para almacenar y manipular los números. Debido a la evolución de los PC y al tamaño de palabra variable de estos, algunos compiladores implementan enteros con un *tamaño de palabra de 32 bits*, otros puede que sólo tengan *16 bits*, y los compiladores de los sistemas más modernos puede que incluso utilicen *64 bits*. **En Java, no hay una relación directa entre el tamaño de palabra de la máquina y el rango de un tipo numérico**. Un valor **`int` siempre tiene 32 bits** en todos los intérpretes Java, independientemente de la plataforma física de la que se trate. Esto permite que los programas que se escriban tengan garantizada su ejecución en cualquier arquitectura sin tener que hacer ninguna portabilidad.

Todos los tipos numéricos de Java son valores con signo y cada uno de los cuales utiliza 1, 2, 4 y 8 bytes para su almacenamiento. Son cuatro tipos: **`byte`**, **`short`**, **`int`** y **`long`**.

Nombre	Tamaño	Rango
`byte`	**`8 bits`**	**`-128 a 127`**
`short`	**`16 bits`**	**`-32.768 a 32.767`**
`int`	**`32 bits`**	**`-2.147.483.648 a 2.147.483.647`**
`long`	**`64 bits`**	**`-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807`**

byte

Es un tipo de ***8 bits con signo***. Su rango comprende desde `-128` a `127`. Es especialmente útil cuando se tiene un *flujo de bytes* externos recibidos desde una red o archivo. Si se necesita analizar un protocolo de red o un formato de archivo, o resolver problemas de ordenación de *bytes*, el tipo ***`byte`*** es el apropiado.

Las variables **`byte`** se declaran utilizando la palabra clave **`byte`**. Por ejemplo, el código siguiente declara dos variables **`byte`**:

```
byte num1;
```

```
byte num2 = 0x55;                //hexadecimal
num1 = 123;                      //decimal
num1 = 0123;                     //octal
num1 = 0b11110011;               //binario -a partir de java 7-
```

En general, se debería evitar la utilización del tipo **`byte`** excepto cuando se trabaje con manipulación de *bits*. Para los enteros normales, que se utilizan para contar y operar, **`int`**, que se describe más adelante, es una opción mucho más adecuada.

short

Es un tipo de **16 bits con signo.** Su rango comprende desde `-32768` a `32767`. Probablemente es el tipo de Java menos utilizado. Ahora que los ordenadores de 16 *bits* empiezan a estar en desuso ya no hay muchos datos **`short`** con los que trabajar.

Ejemplos de declaraciones de variables **`short`**:

```
short num1;
short num2 = 0x55AA;             //hexadecimal
num1 = -13456;                          //decimal
```

int

Es un tipo de **32 *bits* con signo**. Su rango comprende desde `-2.147.483.648` a `2.147.483.647`. Es el tipo más utilizado habitualmente para almacenar valores enteros simples. Con un rango de miles de millones, es ideal para la mayoría de las iteraciones con *arrays* y para contar. Siempre que se tenga una expresión con *enteros* que incluya **`byte`**, **`short`**, **`int`** y *números literales*, la expresión completa se promociona y convierte a **`int`** antes de realizar el cálculo. Ejemplos de declaraciones de variables **`int`**:

```
int num1;
int num2 = 0x55AA0000;           //hexadecimal
num1 = 343566;                   //decimal
```

long

Es un tipo de **64 *bits* con signo**. Hay algunas ocasiones en las que un tipo **`int`** no es lo suficientemente grande como para guardar el valor esperado. Cuando se calculan expresiones enteras con números grandes, una multiplicación puede generar algunos valores intermedios de miles de billones. También, cuando se calcula el tiempo, el número de milisegundos en un año es de cerca de 30.000 millones; se desbordará un **`int`** de 32 *bits*. En estos casos se necesita utilizar un **`long`**.

Ejemplos de declaraciones de variables **`long`**:

```
long num1;
long num2 = 0x55AA000055AA0000;   //hexadecimal
num1 = 343567454566;             //decimal
```

Números en punto flotante

Los números en punto o coma flotante, también conocidos como números reales en otros lenguajes, se utilizan cuando se realizan cálculos que requieren precisión fraccionaria. Los cálculos complejos, como la raíz cuadrada, o trigonométricas, como el seno y el coseno, tienen como resultado un valor cuya precisión requiere un tipo en coma flotante. Hay dos clases de tipos en coma flotante, **`float`** y **`double`**, como se describen:

Nombre	Tamaño	Rango
`float`	32 bits	`±1.40129846432481707e-45 a ±3.40282346638528860e+38`
`double`	64 bits	`±4.94065645841246544e-324d a ±1.79769313486231570e+308d`

float

El formato de precisión simple, especificado por la palabra clave **`float`**, utiliza 32 *bits* para representar un valor; es más rápido en algunos procesadores y ocupa la mitad de espacio que el formato de doble precisión, pero puede tener problemas de precisión cuando los valores sean muy grandes o muy pequeños.

Ejemplos de declaraciones de variables **`float`**:

```
float num1;
float num2 = 3.14f;
```

double

El formato de doble precisión, especificado por la palabra clave **`double`**, utiliza 64 *bits* para almacenar un valor. Realmente la precisión doble es más rápida que la simple en algunos procesadores modernos que han sido optimizados para cálculos matemáticos a alta velocidad.

Cuando se necesita mantener la precisión tras muchos cálculos iterativos, o se están manipulando números de gran tamaño, **`double`** es la mejor opción.

Cualquier valor literal que representa un número con decimales, Java asumirá que es un `double`.

Ejemplos de declaraciones de variables **`double`**:

```
double num1;
double num2 = 3.14159365358979323846;
```

Conversión de tipos numéricos

Hay situaciones en las cuales se tiene un valor de un tipo dado y se desea almacenar ese valor en una variable de un tipo diferente –también llamado *moldeado* (*casting*)-. **En algunos casos es posible asignar el valor sin una conversión de tipos aparente; es lo que se denomina conversión automática. Esto sólo es posible en Java si el compilador reconoce que la variable**

destino tiene la suficiente precisión para contener el valor origen, como almacenar un valor **`byte`** en una variable **`int`**. A esto se le llama ***ensanchamiento* o *promoción*, dado que el tipo más pequeño se ensancha o promociona al tipo compatible más grande**. Si por el contrario, se desea asignar un valor de variable **`int`** a una variable **`byte`** se necesita realizar una **conversión de tipos explícita por estrechamiento**, dado que se estrecha explícitamente el valor para que quepa en el destino.

La conversión de un tipo se realiza poniendo delante un nombre de tipo entre paréntesis, por ejemplo: `(<tipo>) valor`.

El siguiente código muestra la conversión de tipos de **`int`** a **`byte`**. **Si el valor del entero fuese mayor que el rango de un `byte`, se reduciría al módulo (resto de la división entera) del rango de `byte`.**

```
int num1 = 100;
double num2 = 2345.45;
byte num3 = (byte) num1;
int num4 = (int) num2;
byte num5 = (byte) 3456.556f;
```

Caracteres

Java utiliza *Unicode* para representar los caracteres de una cadena, el tipo **`char`** es de **16 *bits* sin signo**. El rango de un carácter es de `0` a `65536`. No hay caracteres negativos. *Unicode* es una unificación de decenas de conjuntos de caracteres, incluyendo el *latín, griego, arábigo, cirílico, hebreo, katakana, hangul* y muchos más.

Ejemplos de declaraciones **`char`**:

```
char c1;
char c2 = 0xf132;
char c3 = 'a';
char c4 = '\n';
```

Aunque no se utilicen los caracteres como enteros, se puede operar con ellos -como si lo fueran- utilizando *conversión explícita*. Esto permite sumar dos caracteres o incrementar el valor de una variable **`char`**.

```
int tres = 3;
char uno = '1';
char cuatro = (char) (tres + uno);
int codigoCuatro = (tres + uno);
```

La variable `cuatro` termina con un carácter `'4'` almacenado en ella. Obsérvese que la variable uno fue promocionada a **`int`** al formar parte de la expresión: `tres + uno`, por lo que se requiere la conversión de tipos explícita `(char)` para volver a **`char`** antes de la asignación a `cuatro`. Por otro lado la variable `codigoCuatro` contiene el valor entero `52` que corresponde al *código Unicode* del carácter `'4'`.

Se llega a la conclusión de que cualquier valor de **`int`** puede ser interpretado como un carácter visualizable según la *tabla Unicode*. No hay que olvidar que un **`int`** tiene más capacidad que un **`char`** y hay valores **`int`** que no corresponden a *códigos Unicode*.

Secuencias de escape

Una secuencia de escape es una serie de caracteres que empiezan con el carácter \ seguido de una letra, un número o la letra u seguida de un número que representa el código de un carácter que puede emplearse en cualquier lugar en un programa de Java para representar un carácter *Unicode*.

Secuencia escape	Significado
`\\`	Barra hacia atrás o backslave (equivalente a: `\u005C`)
`\'`	Comilla simple. Equivalente a: `\u0027`)
`\"`	Doble comilla. Equivale a: `\u0022`
`\b`	Retroceso o backspace. Equivalente a: `\u0008`
`\f`	Salto de página. Equivale a: `\u000C`
`\n`	Nueva línea. Equivale a: `\u000A`
`\t`	Tabulador. Equivale a: `\u0009`
`\r`	Retorno de carro. Equivale a: `\u000D`
`\xxx`	Carácter correspondiente al código octal xxx (x es un dígito entre `0` y 7)
`\uxxxx`	Carácter correspondiente al código hexadecimal xxxx según Unicode (x es un dígito entre `0` y F)

Booleanos

Java tiene el tipo primitivo **`boolean`** para representar los valores lógicos. Sólo puede tomar uno de estos dos posibles valores, **`true`** (verdadero) o **`false`** (falso) que son palabras reservadas.

Un **`boolean`** es el tipo de dato que se obtiene con las *expresiones de comparación* y *expresiones lógicas* que se requieren en sentencias de *control de flujo* que se verán después. Por defecto, si no se especifica, un **`boolean`** toma valor **`false`**.

Los datos de tipo `boolean` en Java ocupan 4 bytes en la memoria cuando se usan como variable suelta y 1 byte si están dentro de un array.

Ejemplo de una declaración de tipo **`boolean`**:

```
boolean terminado = false;
```

Envoltorios

Java utiliza tipos primitivos como por razones de rendimiento. A veces, se necesitará una **representación como objetos de los tipos primitivos.**

Number

La clase abstracta `Number` representa una interface para todos los tipos numéricos escalares estándar:

byte
short
int
long
float
double

`Number` tiene métodos de acceso que devuelven el valor de tipo primitivo, posiblemente redondeado, del objeto convertido a tipo numérico que se indique:

- `doubleValue()` devuelve un **`double`**.
- `floatValue()` devuelve un **`float`**.
- `intValue()` devuelve un **`int`**.
- `longValue()` devuelve un **`long`**.

Double y Float

`Double` y `Float` son subclases de `Number`. tienen un constructor para inicializar con una representación en forma de texto o `String` del valor:

```
Double dato1 = new Double(3.14159);
Double dato2 = new Double("314159E-5");
```

Desde Java 5 con el **autoboxing** se puede asignar tipos numéricos primitivos sin necesidad de utilizar expresamente los constructores.

```
Double d1 = 3.14159;
```

Integer y Long

La clase `Integer` es un envoltorio de **`int`**, **`short`** y **`byte`**.

La clase `Long` es un envoltorio para el tipo **`long`**.

Además de los métodos heredados de `Number`, `Integer` y `Long` tienen otros muy útiles:

- `parseInt(String)` convierte la variable `String` en el valor `int` que representa.
- `parseInt(String, base)` hace lo mismo que el método anterior, pero especifica una base distinta de la décima.
- `toString(int)` convierte el `int` que se pasa al método en su representación como cadena.
- `toString(int, base)` igual al anterior, pero puedo cambiar de base.

Character

Character es un envoltorio de un `char`. Tiene varios métodos estáticos que se pueden utilizar para probar varias condiciones de un carácter:

- `isLowerCase(char ch)` devolverá **`true`** si el carácter es una letra minúscula.
- `isUpperCase(char ch)` devolverá **`true`** para letras mayúsculas.
- `isDigit(char ch)` e `isSpace(char ch)` devuelven **`true`** para caracteres numéricos y espacios en blanco respectivamente.
- `toLowerCase(char ch)` y `toUpperCase(char ch)` convierten entre mayúscula y minúscula y viceversa.

Boolean

Boolean es un envoltorio muy fino alrededor de valores boolean, que sólo es útil para situaciones de paso por referencia.

Constructores

Cada clase envoltorio (menos `Character`) tienen dos constructores: uno que admite el tipo primitivo como parámetro y otro que admite un `String`.

```
Integer a = new Integer(500);
Integer b = new Integer("500");
Float c = new Float(7.5f);
Float d = new Float("7.5f");
Double e = new Double(3.14159);
Double f = new Double("314159E-5");
Character g = new Character('t');
```

Para el constructor de `Boolean` cuando el `String` es **`true`** (sin importar mayúsculas o minúsculas) será **`true`**, cualquier otro valor será **`false`**.

```
Boolean g = new Boolean(false);
Boolean i = new Boolean("TrUe"); //i será true.
Boolean j = new Boolean("NO");   //j será false.
```

ValueOf

Otra forma de construir un objeto de una clase envoltorio es mediante este método estático, `valueOf()`. Este método puede aceptar un `String`, o un `String` y un parámetro que indique la base de numeración.

```
Integer decimal = Integer.valueOf("150");
Integer binario = Integer.valueOf("1010", 2);
```

Conversión

Envoltorios a primitivos

Métodos para obtener el valor numérico primitivo convertido, a partir del valor interno.

- `byteValue()`
- `shortValue()`
- `intValue()`
- `longValue()`
- `floatValue()`
- `doubleValue()`

String a primitivos

Métodos **`static`** para obtener el valor numérico primitivo a partir del texto proporcionado como parámetro.

- `parseByte()`
- `parseShort()`
- `parseInt()`
- `parseLong()`
- `parseFloat()`
- `parseDouble()`

Envoltorio a String cambiando base de numeración

Sólo para `Integer` y `Long`. Los siguientes métodos **`static`** proporcionan el texto resultante de cambiar la base de numeración del número proporcionado como parámetro.

- `toBinaryString()`
- `toOctalString()`
- `toHexString()`

Autoboxing y unboxing

Tradicionalmente en Java, para pasar de un tipo primitivo a su objeto equivalente se necesita utilizar las clases envoltorio. Para obtener de un objeto envoltorio su tipo primitivo se necesitan usar los métodos de las clases envoltorio.

Todas estas operaciones pueden complicar excesivamente el código. Por ello a partir de Java 5 se introdujo una conversión automática (autoboxing) que permite asignar y obtener los tipos primitivos sin necesidad de utilizar los métodos de las clases envoltorio.

```
int enteroPrimitivo = 420;
Integer Entero = enteroPrimitivo;  //Asignación directa. autoboxing.
int otroEnteroPrimitivo = Entero;  //Asignación directa. Se llama unboxing.
```

Actualmente en Java el compilador según el caso se encarga de crear el objeto envoltorio o de extraer el tipo primitivo. También se permite en el paso de parámetros y en expresiones aritméticas.

La asignación entre variables de tipo primitivo y envoltorios siempre se realiza automáticamente con la copia independiente del valor asignado en un nuevo espacio de memoria, en la pila del programa.

```
public class AsignacionPrimitivos
{

   public static void main(String argumentos[]) {
       int dato1;             //variable tipo primitivo
       dato1 = 3;

       int dato2 = dato1;    //copia variable tipo primitivo, se duplica el
valor

       dato1 += 2;           //los cambios en dato1 no afectan a dato2

       System.out.println("Primitivos...");
       System.out.println("valor dato1: " + dato1);
       System.out.println("valor dato2: " + dato2
                                          + "\n\tlos cambios en dato1 "
                                          + "no afectan a dato2, "
                                          + "son independientes\n");


       Integer dato3;           //variable tipo envoltorio
       dato3 = 3;

       Integer dato4 = dato3;  //copia variable tipo envoltorio, se duplica
```

```
        dato3 += 2;             //los cambios en dato3 no afectan a dato4

        System.out.println("Envoltorios...");
        System.out.println("valor dato3: " + dato3);
        System.out.println("valor dato4: " + dato4
                                           + "\n\tlos cambios en dato3 "
                                           + "no afectan a dato4, "
                                           + "son independientes\n");
    }

} //class
```

Ejercicios

1. **¿Cuáles de los siguientes nombres no son identificadores válidos en Java? Explica por qué no son válidas con comentarios en el programa.**
   - Para probarlo crea un fichero llamado `Identif.java`
   - En el programa, declara variables, por ejemplo: **`int`** `_alpha;` (da igual el tipo), utilizando esos identificadores e intenta compilar. El compilador dará errores en los identificadores no válidos.

_alpha **FLOAT** **1_de_muchos**

maxValor **cuantos** **"dado"**

Nbytes **pink.panter** **int**

qué_dices? **Número** **cadena 2**

Cañón **café** **Mesa-3**

Return **While** **__if**

Bloque#4 **c o s a** **_CaPrIcHoSo_**

8ªRoca **3d2** **Hoja3/5**

2. **¿Cuáles de las siguientes constantes literales no son válidas en Java? Explica por qué no son válidas con comentarios en el programa.**

   - Para probarlo crea un fichero llamado *`Literal.java`*
   - En el programa, declara variables de tipo adecuado para inicializarlas con las constantes literales proporcionadas, e intenta compilar. El compilador dará errores con las constantes literales no válidas.

-11.1	**-88.28**	**.3e27**	**23e2.3**
"cañón"	**0377**	**9999**	**099**
+521.6	**1.26**	**5E-002**	**0xFE**
0b101010	**1.26f**	**‘\n’**	**while**
\xFE	**1234**	**.567**	**0xFFFE**
XGA	**"a"**	**'abc'**	**02.45**
'a'	**xF2E**	**0xf**	**abc**
ab"c	**"abc**	**"abc'**	**a'**
"abc;"	**"abc'"**	**"abc""**	**""abc"**
"true"	**True**	**false**	**'\\'**

3. **Adapta el programa del ejercicio anterior para utilizar tipos de dato envoltorio (Wrapper) de Java y mostrar cada caso por pantalla.**

- Para probarlo, copia *`Literal.java`* y renómbralo como *`Literal2.java`*.
- En el programa, declara variables de tipo *envoltorio* adecuadas e inicialízalas de igual manera que se hizo con los tipos de datos primitivos. Esto funcionará sin errores con versiones de Java 1.5 en adelante. Es lo que se denomina *autoboxing*.
    - `Double dato = -11.1;  //Ejemplo autoboxing`
    - Como alternativa, un poco más avanzada -válida en todas las versiones antiguas de Java-, se puede utilizar el operador **`new`** y el método constructor más adecuado al valor literal que corresponda.
    - Ejemplo de esto, para el primer caso, sería: `Double dato = new Double(-11.1);`

- Muestra por pantalla el valor con el que ha quedado cada una de las variables.

4. **Escribe un programa simple Java pida y visualice el nombre y la edad de dos personas en dos líneas diferentes.**

- La nueva clase debe llamarse `Nombre1`.
- Ajusta adecuadamente la información que aparece en la cabecera del fichero del programa.
- La entrada de datos se realiza mediante el método `teclado.nextLine()`.

5. **Escribe un programa simple Java que lea dos números y que escriba el mayor de los dos.**

- La clase debe llamarse: `MayorDeDos`.
- Ajusta adecuadamente la información que aparece en la cabecera de comentarios del fichero del programa.
- La entrada de datos se realiza mediante el método `teclado.nextLine()`.
- Para almacenar en una variable numérica un dato leido por teclado hay que convertir los caracteres leídos en texto, para ello se pueden utilizar las clases envoltorio.

6. **Escribe un programa simple Java que lea un número e indique si es par o no.**

- La clase debe llamarse `ParImpar`.
- Ajusta adecuadamente la información que aparece en la cabecera de comentarios del fichero del programa.
- La entrada de datos se realiza mediante el método `teclado.nextLine()`.

7. **Escribe un programa simple Java que incorpore las sentencias necesarias para leer los datos de tres personas (nombre, apellidos, nif y dirección postal) desde el teclado, los guarde y los muestre, en pantalla; se deben mostrar en orden inverso, primero los datos de la última persona.**

- La clase debe llamarse **`TresUsuarios`.**
- La entrada de datos se realiza mediante el método `teclado.nextLine()`.

8. **Escribe un programa simple Java que lea tres números y escriba el mayor de los tres.**

- La nueva clase debe llamarse `MayorDeTres`.
- La entrada de datos se realiza mediante el método `teclado.nextLine()`.

9. ➢ ¿Cuál es la salida del siguiente código Java?

```
public class Envoltorios1 {

    public void metodoEnvoltorio(Integer a) {
        System.out.println("Integer "+ a);
    }
```

```
    public void metodoEnvoltorio(int a) {
        System.out.println("int " + a);
    }

    public void metodoEnvoltorio(double a) {
        System.out.println("double " + a);
    }

    public void metodoEnvoltorio(Double a) {
        System.out.println("Double " + a);
    }

    public static void main(String[] args) {
        Envoltorios e = new Envoltorios1();
        Integer a1 = new Integer(1);
        int a2 = 1;
        Double b1 = new Double(1);
        int b2 = 1;
        e.metodoEnvoltorio(a1);
        e.metodoEnvoltorio(a2);
        e.metodoEnvoltorio(b1);
        e.metodoEnvoltorio(b2);
    }
}
```

10. ➢ ¿Cuál es la salida del siguiente código Java?

```
public class Envoltorios2 {

    public void metodoEnvoltorio(Integer a) {
        System.out.println("Integer " + a);
    }

    public static void main(String[] args) {
        Envoltorios e = new Envoltorios2();
        int a1 = 1;
        e.metodoEnvoltorio(a1);
    }
}
```

11[1].

➢ ¿Cuál es la salida del siguiente código Java?

```
public class Envoltorios3 {

    public void metodoEnvoltorio(Integer a) {
        System.out.println("Integer " + a);
    }

    public static void main(String[] args) {
        Envoltorios e = new Envoltorios3();
        short a1 = 1;
        e.metodoEnvoltorio(a1);
    }
}
```

1

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en linea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- CARRERES, J. *Manual de Java*. [en línea] http://www.oocities.org/collegepark/quad/8901/indice.html
- [2]Java Hispano. [en línea] http://www.javahispano.org/certificacion/2011/11/4/clases-envoltorio-y-boxing.html

2