



Tipos compuestos de datos,
enumeraciones

Índice



Tipos compuestos de datos, enumeraciones

1 Arrays de una dimensión	3
2 Bucle for-each	6
3 Cadenas	7
4 Los argumentos de la línea de comandos	12
5 Arrays multidimensionales	14
6 Enumeraciones	17
7 Envoltorios de tipos	20
8 Autoboxing y Unboxing	23

1. Arrays de una dimensión

Un array es un conjunto de posiciones de memoria, todas del mismo tipo, que se referencian con el mismo nombre y un índice para cada posición.

Los corchetes [] son los que denotan que la estructura de datos es array. Los arrays se declaran de acuerdo a uno de los dos siguientes formatos:

```
<tipo> [ ] <identificador_array>;  
<tipo> <identificador_array> [];
```

<identificador_array>, no es el array, es el nombre que se va a utilizar para poder acceder a los elementos del array. Con el formato anterior no se crea el array, sólo el nombre con el que se va a operar con dicho array.

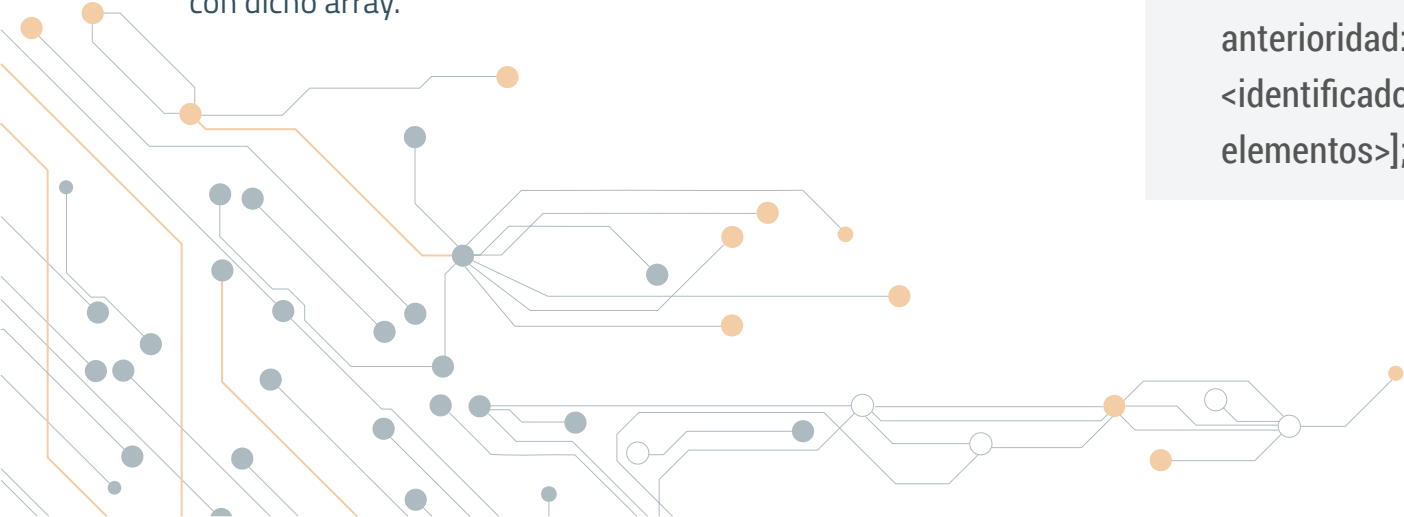
Los diferentes formatos de creación de arrays son:

```
<tipo> [ ] <identificador_array> = new <tipo>[<numero_ elementos>];
```

```
<tipo> <identificador_array> [ ] = new <tipo>[<numero_ elementos>];
```

Si el identificador de array ya esta declarado con anterioridad:

```
<identificador_array> = new <tipo>[<numero_ elementos>];
```



Ejemplos:

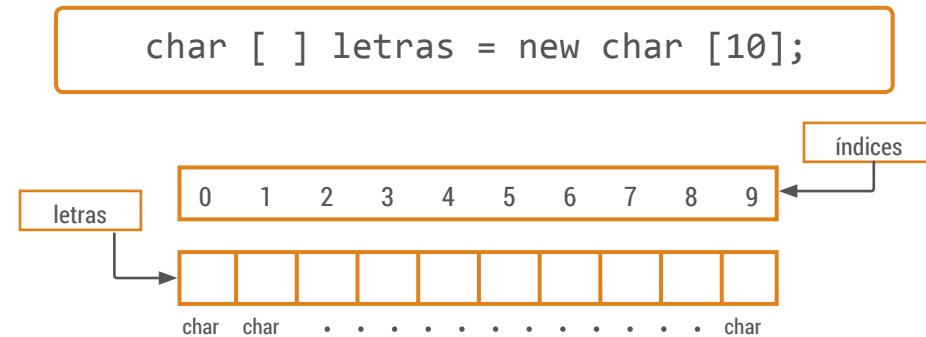
```
int listanumeros [ ]; //listanumeros identificador para
                    //acceder al array de int
listanumeros = new int [10]; //creación de array
para 10 int,
//que podrá ser utilizado con identificador
listanumeros

char letras = new letras [10];
```

Es el operador new el que crea las posiciones de memoria consecutivas y del mismo tipo que se podrán utilizar con el identificador declarado.

Cada elemento del array tiene asociado un índice, que siempre será un valor entero, al igual que el tamaño. El primer elemento tiene índice 0 y el último tiene como índice una unidad menos que el tamaño. En realidad el índice representa el desplazamiento con respecto al primer elemento.

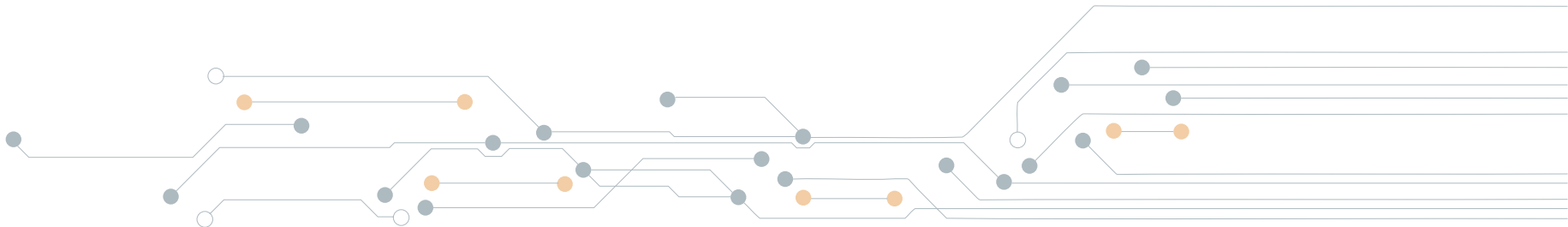
En la imagen siguiente se muestra la creación de un array de 10 elementos de tipo char. El identificador letras hace referencia al array.



```
letras.length //tamaño es 10
```

FIGURA 4.1: CREACIÓN DE UN ARRAY

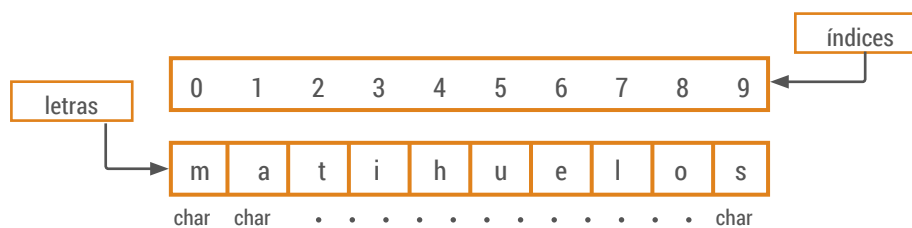
Todo identificador de array tiene asociado el atributo llamado **"length"** que almacena el número de elementos que tiene el array.



Una vez creado un array, hay que acceder a sus elementos para darles un valor u obtener el valor almacenado en dichos elementos. El formato de acceso a las posiciones del array es el siguiente:

`<identificador_array> [<indice>]`

Se utiliza **"length"** para poder controlar que el valor utilizado para el índice, nunca es igual o mayor al tamaño del array. Si se intenta acceder con un índice igual o superior al **"length"** se genera un error que lanza la excepción: **IndexOutOfBoundsException**.



`letras[2] ==> tiene la letra 't'`

`letras[8] = 'a' ==> se almacena 'a' en posición 9 e índice 8`

`letras[10] = 'a' ==> excepcion IndexOutOfBoundsException.`

En el código siguiente se muestra como se crea un array, se accede a sus elementos para almacenar un valor en cada uno y después se recorre para visualizar el valor almacenado en cada posición.

```
Random aleatorio = new Random();
int size = aleatorio.nextInt(25)+1;
int lista [] = new int[size];
//Recorrer el array y almacenar un valor en cada
posición
for (int i=0; i< lista.length; i++){
    lista[i] = aleatorio.nextInt(1000);
}
//Recorrer el array y obtener el valor de cada posición
for (int i=0; i< lista.length; i++){
    System.out.println("lista["+i+"] :"+lista[i]);
}
```

FIGURA 4.1: ACCESO A ELEMENTOS DE UN ARRAY.

Cuando se crea un array también puede inicializarse con valores concretos y determinados. el formato para realizar esto es el siguiente:

```
<tipo> [ ] <identificador_array> = { <lista_valores_tipo> }
```

Ejemplos:

```
//array de cinco char
char vocales [] = {'A', 'E', 'I', 'O', 'U'};
//array de 10 números, es opcional poner el tamaño
int lista [10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
//array de tres cadenas
String nombres [] = {"Pepe Pérez", "Gil Lopez", "Ana Ruiz"};
```

2. Bucle for-each

En la versión JDK 5.0 se incluye en Java el llamado "bucle for mejorado", la sentencia **for each**, que permite recorrer todos los elementos de una colección de datos, tales como un array, sin tener que recurrir a la utilización de índices, ni el valor del tamaño.

El formato de utilización de esta sentencia es:

```
for ( <tipo> <identificador> : <coleccion> ){
    <sentencias>
}
```

Asigna a la variable referenciada por **<identificador>** el valor de cada uno de los elementos de la **<coleccion>** y después ejecuta las **<sentencias>**.

En **<sentencias>** no puede haber ninguna que cambie el tamaño de **<coleccion>**, ni el valor de ningún elemento.

El bucle for each es para recorrer colecciones y acceder a los elementos para obtener su valor.

Ejemplo:

```
Random aleatorio = new Random();
    int size = aleatorio.nextInt(25)+1;
    int lista [] = new int[size];
    //Recorrer el array y almacenar un valor en cada posición
    for (int i=0; i< lista.length; i++){
        lista[i] = aleatorio.nextInt(1000);
    }
    //Recorrer el array y obtener el valor de cada posición
    int pos=0;
    for(int n : lista)
        System.out.println("Posicion "+(++pos)+" : "+n);
```

3. Cadenas

Hasta este momento se han considerado las cadenas como secuencias de caracteres almacenados en posiciones consecutivas de memoria y su representación de literales como secuencia de caracteres encerrados entre doble comillas (""). A partir de este momento las cadenas se van a considerar como lo que son objetos instanciados de la clase String.

El formato para crear un objeto de tipo String es el siguiente:

```
String <identificador_referencia> = new String(<cadena>);
String <identificador_referencia>= <cadena>;
```

El operador **new** es como en los arrays y para todos los objetos quien se encarga de crear el espacio necesario para almacenar la cadena, el array o el objeto.

El operador **new** devuelve una referencia al espacio creado en memoria, una cadena, un array o un objeto, por tanto lo habitual es colocar el operador new a la derecha de un operador de asignación (=) y a la izquierda del operador el **<identificador_referencia>** que va a permitir trabajar con la cadena, el array o el objeto.

El **<identificador_referencia>** tiene que ser del tipo que se crea con new, en este caso String. El **<identificador_referencia>** no es la cadena, es una referencia a la cadena.

En la imagen se explica la relación entre el objeto String y el **<identificador_referencia>** de tipo String.

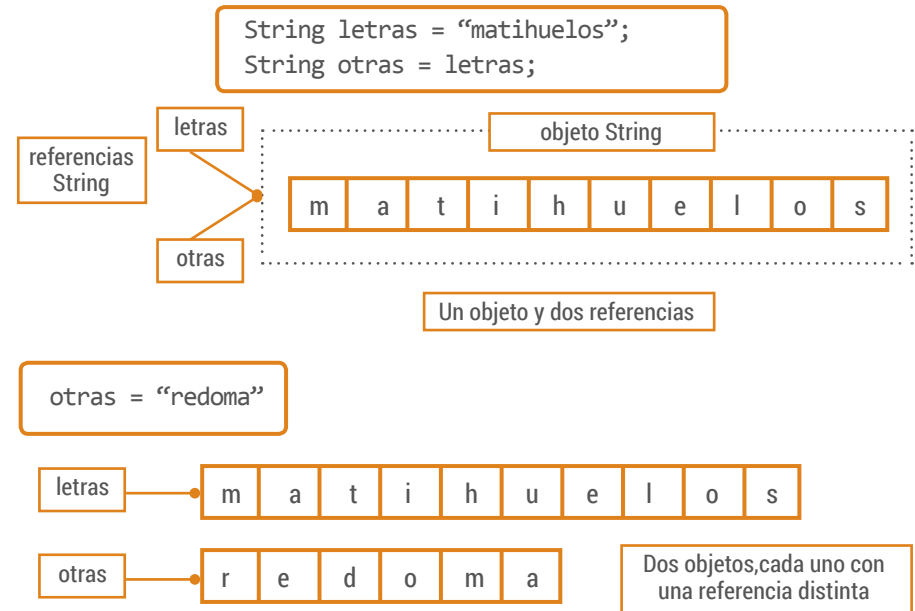


FIGURA 4.1: ACCESO A ELEMENTOS DE UN ARRAY.

La característica más importante de la clase **String** es que es **immutable**, esto quiere decir que no se puede cambiar el contenido, cada vez que asignamos un valor diferente a una referencia a cadena, se esta creando un objeto cadena nuevo, con un nuevo contenido.

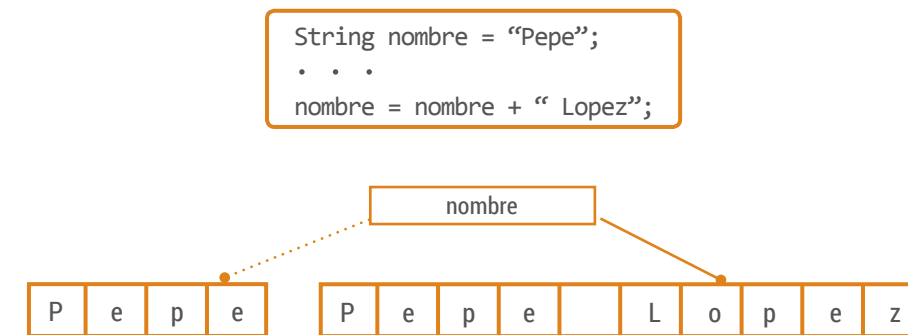


FIGURA 4.3: STRING ES IMMUTABLE.

Tal y como se ha estado utilizando el operador de concatenación "+", cuando uno de sus operandos es una cadena devuelve una nueva cadena resultado de la concatenación de los dos operandos.

Además de esta funcionalidad, como las cadenas son objetos de la clase String, tienen como funcionalidades todos los métodos definidos en String. Algunos son los que se indican en la tabla siguiente:

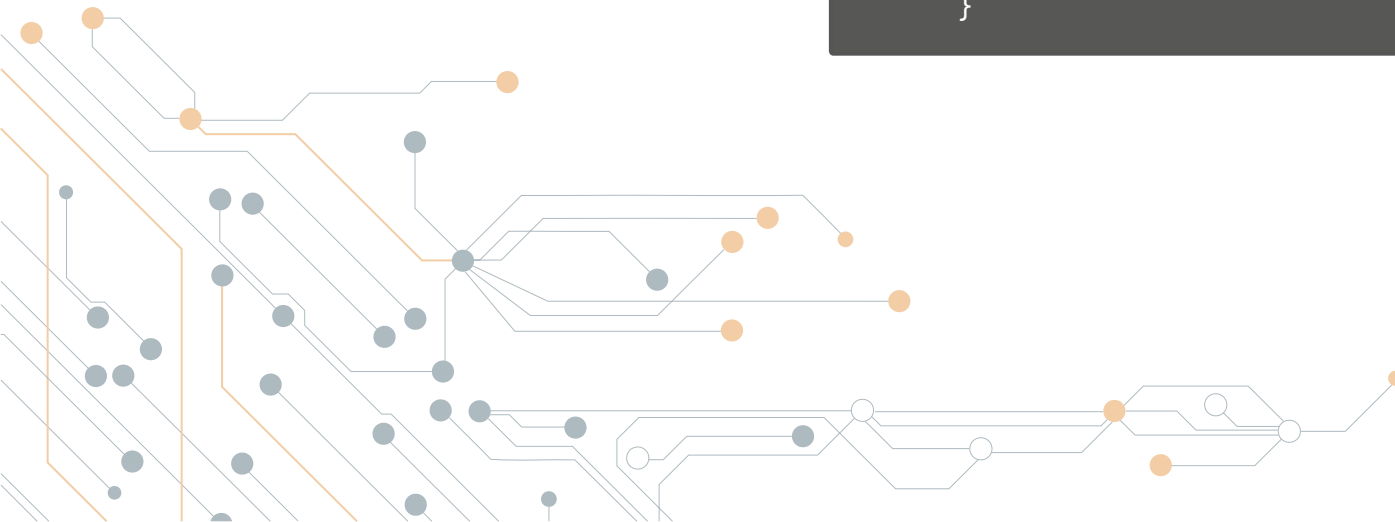


Método	Funcionalidad
boolean equals(<cadena>)	Retorna true si la cadena que invoca contiene los mismos caracteres que <cadena>. El operador == no compara los caracteres. compara si las dos referencias referencian al mismo objeto.
int length()	Obtiene la longitud de la cadena
char charAt(<índice>)	Devuelve el carácter cuyo índice en la cadena es <índice>, los índices siempre son enteros.
int compareTo(<cadena>)	Retorna un valor negativo si la cadena que invoca es menor que <cadena>, valor positivo si es mayor que <cadena> y cero si son iguales.
int indexOf(<cadena>)	Busca en la cadena que invoca la subcadena especificada por <cadena>. Devuelve el índice de la primera coincidencia o -1 en caso de encontrarla.
int lastIndexOf(<cadena>)	Busca en la cadena que invoca la subcadena especificada por <cadena>. Devuelve el índice de la última coincidencia o -1 en caso de encontrarla.
String substring(<inicio_índice>)	Devuelve un nuevo String que contiene todos los caracteres del String que invoca desde el índice <inicio_índice> hasta el final.
String substring(<inicio_índice>, <fin_índice>)	Devuelve un nuevo String que contiene todos los caracteres del String que invoca desde el índice <inicio_índice> hasta <fin_índice>, sin incluir este.
String toLowerCase()	Devuelve un nuevo String que tiene todos los caracteres del que invoca pero en minúsculas.
String toUpperCase()	Devuelve un nuevo String que tiene todos los caracteres del que invoca pero en mayúsculas.
String trim()	Devuelve un nuevo String que igual que el invocado pero habiendo eliminado todos los espacios en blanco.
String valueOf(<valor_de_un_tipo>)	Devuelve un nuevo String resultado de convertir a cadena el valor que recibe como parámetro. Esta función es static, por tanto no se invoca con ningún objeto sino con el nombre de la clase String: <code>String.valueOf(1_345_231) //retorna "1345231"</code>
String replace(<caracter1>, <caracter2>)	Devuelve un nuevo String en el que se han remplazado las apariciones de <caracter1> por <caracter2>.

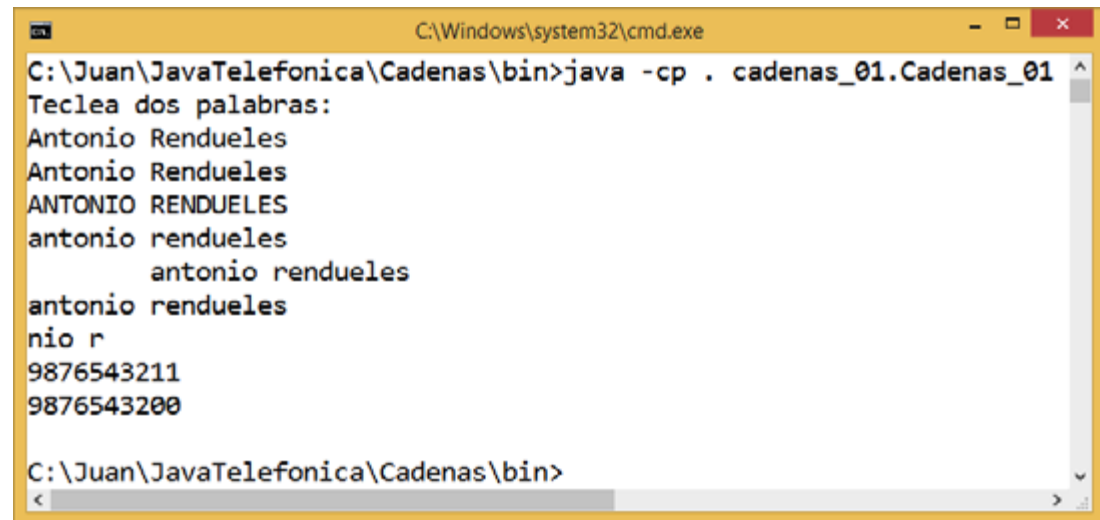
Tabla 4.1: Métodos de la clase String.

Ejemplos con métodos de clase String:

```
public static void main(String[] args) {  
    Scanner teclado = new Scanner(System.in);  
    System.out.println("Teclea dos palabras: ");  
    String palabras = teclado.nextLine();  
    System.out.println(palabras);  
    System.out.println(palabras.toUpperCase());  
    palabras = palabras.toLowerCase();  
    System.out.println(palabras);  
    palabras= "      "+palabras;  
    System.out.println(palabras);  
    palabras = palabras.trim();  
    System.out.println(palabras);  
    String str1 = palabras.substring(4, 9);  
    System.out.println(str1);  
    long num = 9_876_543_211L;  
    String str2= String.valueOf(num);  
    System.out.println(str2);  
    str2= str2.replace('1', '0');  
    System.out.println(str2);  
    teclado.close();  
}
```



La salida por consola de este código es la siguiente:



```

C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\Cadenas\bin>java -cp . cadenas_01.Cadenas_01
Teclea dos palabras:
Antonio Rendueles
Antonio Rendueles
ANTONIO RENDUELES
antonio rendueles
    antonio rendueles
antonio rendueles
nio r
9876543211
9876543200
C:\Juan\JavaTelefonica\Cadenas\bin>

```

FIGURA 4.4: RESULTADO DE LA EJECUCIÓN DEL CÓDIGO DEL EJEMPLO DE USO DE STRING.

4. Los argumentos de la línea de comandos

La función **main** recibe como parámetro un array de cadenas. Cada elemento de este array es uno de las cadenas tecleadas a la derecha del nombre de la clase en la que se define main, son los llamados argumentos de la línea de comandos.

Dada una clase "Programa.class", en el paquete "ejemplo", con una función **main** se puede ejecutar el código de dicha función en la forma siguiente:

```
javac -cp . ejemplo/Programa uno dos tres
```

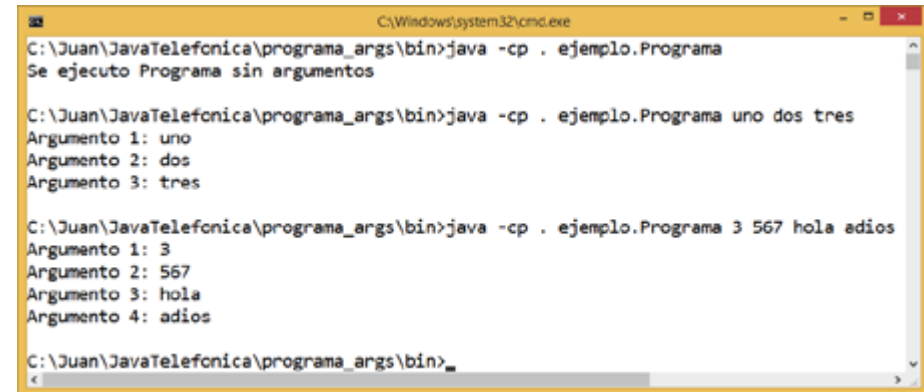
Se han utilizado tres argumentos: "uno", "dos" y "tres".

La función **main**, tiene como parámetro el array: "**String[] args**", según este ejemplo tiene un elemento por cada uno de los argumentos de la llamada. La referencia a este array típicamente es llamado **args**, pero se puede utilizar cualquier otro identificador valido que se quiera utilizar.

El ejemplo siguiente muestra un ejemplo de código que utiliza el array args y muestra cada uno de los argumentos de la línea de comandos, independientemente de su número. package ejemplo;

```
public class Programa {
    public static void main(String[] args) {
        int nArgumentos = args.length;
        if(nArgumentos == 0){
            System.out.println("Se ejecuto
Programa sin argumentos");
        }
        else{
            int num=1;
            for(String argumento: args){
                System.out.print("Argumento
" + num + ": ");
                System.out.
println(argumento);
                num++;
            }
        }
    }
}
```

La imagen siguiente muestra varias llamadas al programa anterior.



```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\programa_args\bin>java -cp . ejemplo.Programa
Se ejecuto Programa sin argumentos

C:\Juan\JavaTelefonica\programa_args\bin>java -cp . ejemplo.Programa uno dos tres
Argumento 1: uno
Argumento 2: dos
Argumento 3: tres

C:\Juan\JavaTelefonica\programa_args\bin>java -cp . ejemplo.Programa 3 567 hola adios
Argumento 1: 3
Argumento 2: 567
Argumento 3: hola
Argumento 4: adios

C:\Juan\JavaTelefonica\programa_args\bin>
```

FIGURA 4.5: RESULTADO DE LA EJECUCIÓN DEL CÓDIGO DEL EJEMPLO DE USO DE ARGUMENTOS.

5. Arrays multidimensionales

Los array multidimensionales utilizan más de un índice para acceder a sus elementos. La estructura más usada es la de dos dimensiones, normalmente llamada tabla. Los formatos de creación de una tabla son los siguientes:

1) Creación de la tabla con las dos dimensiones fijadas, el tamaño del espacio que ocupa la tabla es: [num_filas]*[num_columnas]*tamaño(<tipo>):

```
<tipo> <identificador_tabla> [ ] [ ] = new <tipo> [num_filas] [num_columnas];
```

2) Declaración del identificador para acceder a los elementos de la tabla y posteriormente creación de la tabla:

```
<tipo> <identificador_tabla> [ ] [ ];
```

...

```
<identificador_tabla> = new <tipo> [num_filas] [num_columnas];
```

3) Creación de una tabla en la que el número de columnas no es el mismo para cada fila:

```
<tipo> <identificador_tabla> [ ] [ ] = new <tipo> [num_filas] [ ];
```

...

```
<identificador_tabla> [fila_n] = new <tipo> [num_columnas];
```

En ejemplo siguiente se crea una tabla, se introducen valores en sus elementos con dos “for anidados”, el externo controla el índice de acceso a las filas, el interno controla el acceso a las columnas en cada fila. Se recorre la tabla visualizando cada elemento utilizando “for each anidados”.

```
Random aleatorio = new Random();
    int apuestas = aleatorio.nextInt(10) +
1;

    // Crear array de dos dimensiones
    int[][] numeros = new int[apuestas][6];
    // Recorrer array de dos dimensiones
    // almacenando un valor en cada posición
    System.out.println("Apuestas: " +
numeros.length);
    for (int i = 0; i < numeros.length; i++)
    {
        for (int j = 0; j < numeros[i].
length; j++) {
            numeros[i][j] = aleatorio.
nextInt(49) + 1;
        }
    }
    // Recorrer array de dos dimensiones
    // accediendo a cada posición
    for (int[] apuesta : numeros) {
        for (int numero : apuesta) {
            System.out.print(numero +
"\t");
        }
        System.out.println();
    }
```

Los array bidimensionales también pueden ser inicializados en su creación utilizando el formato de lista de literales del tipo de los elementos entre llaves { }.

En esta forma, cada fila deberá tener su lista de valores de inicialización y para cada lista de cada columna un literal para elemento de la columna.

En el ejemplo siguiente se ilustra el uso de esta forma de inicializar tablas:

```
String grupo [] []= { {"Pepe",
"Madrid", "coordina"},
{"Ana", "Sevilla", "colabora"},
{"Luis", "Lugo", "escribe"}
};
for(String[] persona : grupo){
    for (String dato: persona){
        System.out.print(dato +
"\t\t");
    }
    System.out.print("\n");
}
```

Lo que debe quedar claro, de forma similar a los arrays de una dimensión, es lo siguiente:

- **<identificador_tabla>**, no es la tabla, es el identificador para referenciar la tabla.
- **<identificador_tabla> [<indice>]**, no es una fila, es la referencia a la fila <indice>.
- **<identificador_tabla> [<indice_fila>] [<indice_columna>]**, referencia al elemento de la tabla de la columna [<indice_columna>] y fila [<indice_fila>].
- El tipo de <identificador_tabla> es <tipo> [] [].
- El tipo de <identificador_tabla> [<indice>] es <tipo> [].
- El tipo de <identificador_tabla> [<indice_fila>] [<indice_columna>] es <tipo>.

En la imagen siguiente se muestra como están en realidad almacenados los elementos de una tabla y como las referencias permiten acceder a sus valores.

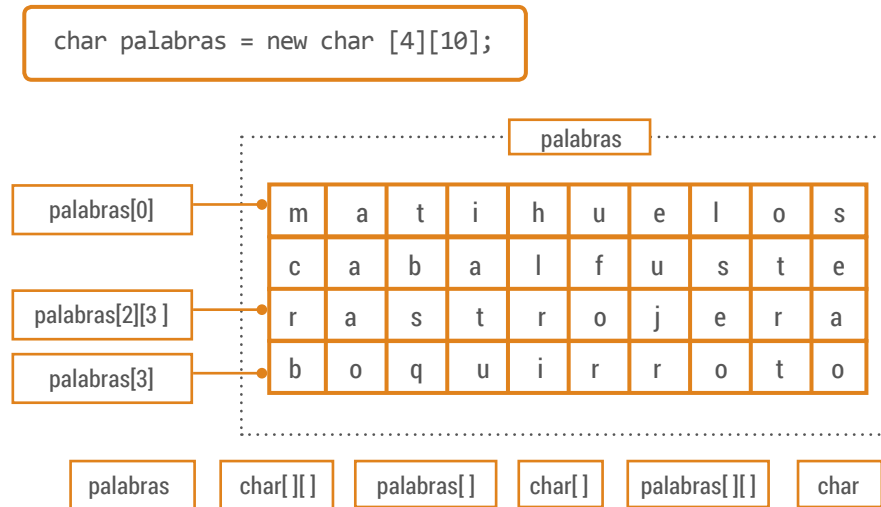


FIGURA 4.6: REPRESENTACIÓN DE UN ARRAY DE DOS DIMENSIONES (TABLA).

6. Enumeraciones

Una enumeración es una lista de constantes con nombre que definen un tipo nuevo. Una variable de tipo enumeración sólo puede almacenar uno de los valores de la lista. Para crear una enumeración se utiliza la palabra reservada **enum**, de acuerdo al formato siguiente:

```
enum <identificador> { <lista_cosntantes> }
```

Algunos ejemplos de creación y uso de enumeraciones son los siguientes:

```
enum EstadoCivil {  
    SOLTERO, CASADO, VIUDO, SEPARADO, DIVORCIADO  
}  
EstadoCivil ec = CASADO;  
if (ec == EstadoCivil.SOLTERO){    /*sentencias*/  
switch( ec ){  
    case SOLTERO:  
        //...  
    case CASADO:  
        //...  
}
```

En Java es convención estilística poner las constantes en mayúsculas.

Todas las enumeraciones cuentan con métodos predefinidos:

- **values()**, devuelve un array que contiene la lista de las constantes de la enumeración.
- **valueOf(String cadena)**, devuelve la constante de enumeración que se corresponde con la cadena pasada con parámetro.
- **ordinal()**, devuelve el número de la posición de la constante en la lista.

Las enumeraciones se crean en el ámbito de la clase, no se pueden crear dentro del ámbito static de la función main.

En el código que se muestra a continuación se utiliza la enumeración `EstadoCivil` y sus funciones **`ordinal()`** y **`values()`**, así como la función **`toString()`** que implícitamente es llamada para los objetos u enumeraciones que se utilizan en los parámetros de la expresión de concatenación de las funciones "print" o "println" de `System.out`.

```
Random aleatorio = new Random();
int num = aleatorio.nextInt(9)+1;

int posicion=0;
EstadoCivil [] valores_enumeracion = EstadoCivil.values();
for(EstadoCivil estado: valores_enumeracion){
    posicion = estado.ordinal();
    System.out.println("La constante número "+ posicion+
        " es "+estado.toString());
}

EstadoCivil [] estados = new EstadoCivil[num];
int nestados = EstadoCivil.values().length;
for (int i=0; i<num; i++ ){
    estados[i]=valores_enumeracion[aleatorio.
nextInt(nestados)];
}

posicion=0;
for(EstadoCivil estado: estados){
    System.out.println("estados["+posicion+"]:
"+estado);
    posicion++;
}
```

Una de las posibles salidas que puede producir el código anterior, dependiendo del número de enumeraciones EstadoCivil que se creen, es la siguiente:

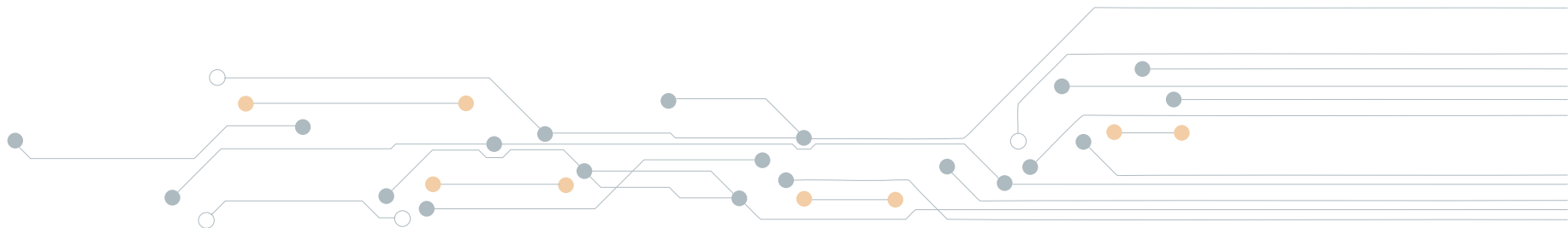
```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\enumeraciones\bin>java -cp . enumeraciones.Enumeracion
La constante número 0 es CASADO
La constante número 1 es SOLTERO
La constante número 2 es VIUDO
La constante número 3 es SEPARADO
La constante número 4 es DIVORCIADO
estados[0]: SEPARADO
estados[1]: SOLTERO
estados[2]: DIVORCIADO
estados[3]: SEPARADO
estados[4]: SOLTERO
estados[5]: SEPARADO
estados[6]: VIUDO

C:\Juan\JavaTelefonica\enumeraciones\bin>dir enumeraciones
El volumen de la unidad C es TI31419800A
El número de serie del volumen es: 4E45-886F

Directorio de C:\Juan\JavaTelefonica\enumeraciones\bin\enumeraciones
10/10/2015  20:41    <DIR>          .
10/10/2015  20:41    <DIR>          ..
10/10/2015  21:05             1.296 Enumeracion$EstadoCivil.class
10/10/2015  21:05             1.758 Enumeracion.class
```

FIGURA 4.7: EJECUCIÓN DE CÓDIGO UTILIZANDO ENUM.

Se observa que en el directorio del paquete “enumeraciones”, Java ha creado una clase para la enumeración: Enumeracion\$EstadoCivil.class, es por lo que las enumeraciones se comportan como objetos y el tipo de la enumeración en realidad es una clase.



7. Envoltorios de tipos

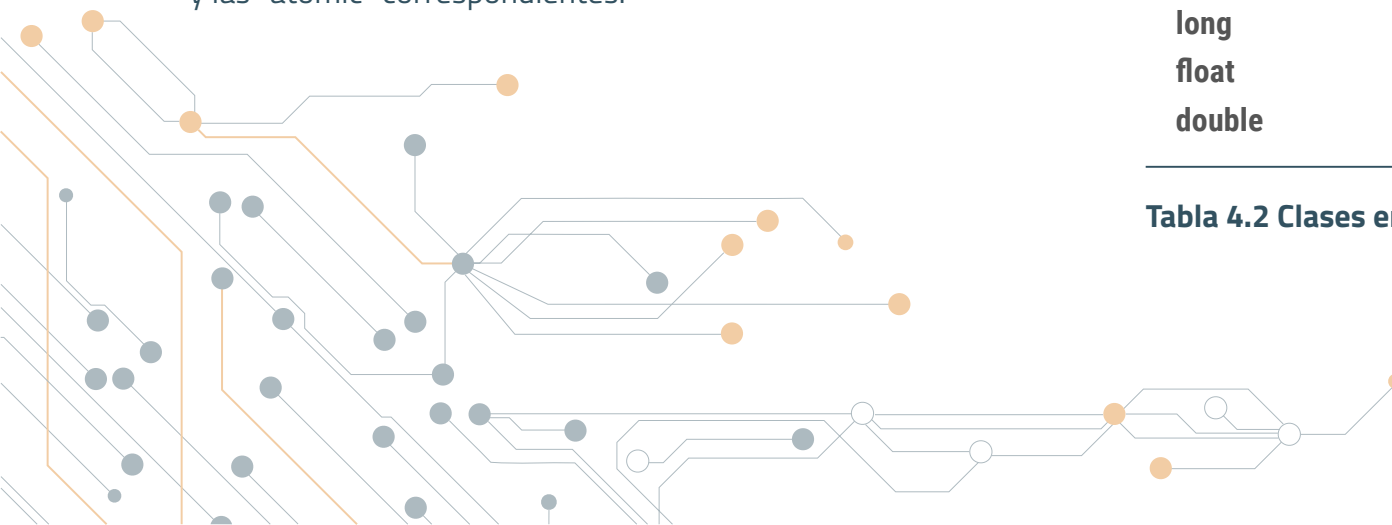
Como se mostro en la tabla de los tipos primitivos de Java, cada tipo tiene la equivalente clase envoltorio o “wrapper”, esto es así porque en algunas situaciones no se van a poder utilizar variables de tipos primitivos sino objetos. Por ejemplo cuando se utilicen colecciones, se verá que son un conjunto de objetos y por tanto si queremos utilizar una colección de int, no se podrá, en su lugar se utilizaran objetos de la clase Integer.

Las clases envoltorio encapsulan el funcionamiento y la representación de los tipos primitivos. Cada objeto de una clase envoltorio es inmutable, quiere decir que representa un único valor, una vez construido, su valor interno no puede ser cambiado. Para resolver esto las clase envoltorio para algunos enteros y boolean tienen una equivalente cuyo nombre comienza por “Atomic”. Así **Integer** tiene la equivalente **AtomicInteger**.

La tabla siguiente muestra los tipos primitivos, sus clases envoltorios y las “atomic” correspondientes:

Tipo primitivo	Clase envoltorio	Clase atomic
boolean	Boolean	AtomicBoolean
byte	Byte	
char	Character	
short	Short	
int	Integer	AtomicInteger
long	Long	AtomicLong
float	Float	
double	Double	

Tabla 4.2 Clases envoltorio.



Se van a estudiar estas clases de acuerdo a las funcionalidades más comunes:

1.- Construcción:

Todas las clases envoltorio, excepto **Character** tienen dos constructores:

- El primero toma el dato primitivo como parámetro, así de un dato primitivo se obtiene un objeto. Este es el único que tiene **Character**.

```
float sueldo = 1_245,8F  
Float oSueldo= new Float(sueldo);
```

- El segundo toma un **String** como parámetro, así a partir de una cadena se obtiene un objeto que encapsula un tipo primitivo.

```
String sueldo = "1245,8";  
Float oSueldo = new Float(sueldo);
```

2.- Métodos **valueOf**:

Son métodos **static**, de la clase, y devuelven un objeto del tipo envoltorio a partir de la cadena que reciben como parámetro.

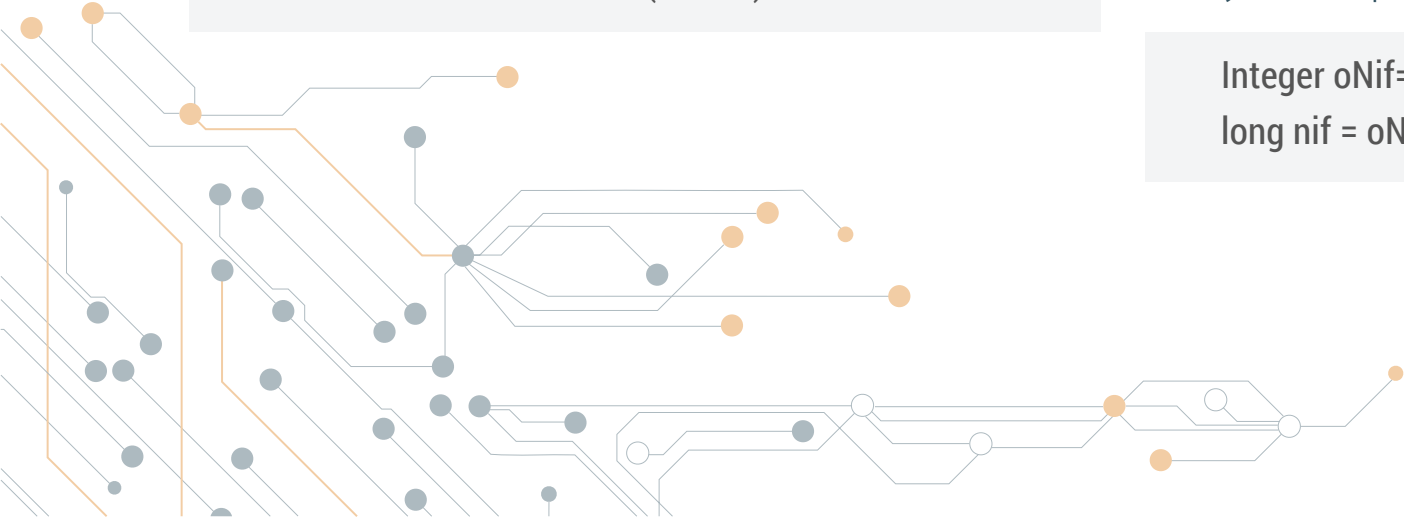
Tienen dos formas. Una recibe sólo la cadena a convertir a objeto tipo envoltorio. La otra recibe como segundo representa la base (binaria, octal, hexadecimal) en la que esta representada el primer argumento.

```
Float peso = Float.valueOf("73,2");  
Byte unbyte = Float.valueOf("00101011",2);
```

3.- Métodos de conversión **xxxValue**:

Las clases envoltorio **Byte**, **Short**, **Integer**, **Long**, **Float** y **Double**, tienen todas los métodos **byteValue**, **shortValue**, **intValue**, **longValue**, **floatValue** y **doubleValue**, que devuelven el valor del tipo primitivo que se corresponde con el método a partir del valor del objeto correspondiente de la clase envoltorios

```
Integer oNif= new Long("05221791");  
long nif = oNif.longValue();
```



4.- Métodos de conversión parseXXX:

Los métodos **parseXXX** son static, se aplican a las clases envoltorio y devuelven el valor del tipo primitivo que se corresponde con la clase del parámetro String que recibe.

Cada clase envoltorio tiene su método static "parse", por ejemplo Integer tiene parseInt.

```
int numero = Integer.parseInt("89789");
```

5.- Conversión a String con toString

Como ya se menciona con anterioridad, cuando un literal o variable o objeto forma parte como operando de una expresión de concatenación, para los objetos se ejecuta implícitamente toString() y los valores de tipos primitivos se convierten a String. Todas las clases envoltorios tienen su método toString que devuelve la cadena que se corresponde con el valor del objeto de la clase envoltorio.

Pero también tienen el método toString con un parámetro del tipo primitivo correspondiente que lo devuelve en forma de cadena.

```
Double d = new Double(897_789_456E14);  
String strd = d.toString();  
System.out.println(strd); //es lo mismo que System.  
out.println(d);  
String cadena = "cadena de Java";  
String cadena1 = cadena + ": tiene de longitud "  
                + Long.toString(cadena.length);
```

6.- Conversión a binario, octal o hexadecimal con toXXXString:

Las clases envoltorio **Integer** y **Long**, tienen los métodos static **toBinaryString**, **toOctalString** y **toHexString**, que devuelven la cadena con la representación en binario, octal o hexadecimal del **int** o **long** que reciben como parámetros.

```
String octal = Long.toOctalString(4567895689L);  
String binario = Integer.toBinaryString(44444);  
String hexadecimal = Integer.toHexString(65535);
```



7.- Rangos de valores y tamaño de los tipos primitivos:

- Todas las clases envoltorio excepto Boolean, tienen las constantes static MIN_VALUE, MAX_VALUE, SIZE y BYTES, que almacenan el valor menor, el valor mayor, el tamaño en bites y en bytes, respectivamente, de los tipos primitivos que encapsulan.

```
System.out.println("byte ==> MAYOR " + Byte.MAX_
VALUE
    + " MENOR: " + Byte.MIN_VALUE+ " SIZE:
"+Byte.SIZE
    + " BYTES: " + Byte.BYTES);
// La salida en pantalla es:
// byte ==> MAYOR 127          MENOR: -128
// SIZE: 8                    BYTES: 1
```

8. Autoboxing y Unboxing

En Java se utilizan las sentencias break y continue para implementar saltos ocasionales en el flujo de control del programa. La sentencia break es utilizada en la estructura de switch, para interrumpir la ejecución de sentencias de cada una de las alternativas.

“Autoboxing” consiste en la conversión automática de tipo primitivo a objeto clase envoltorio, se puede utilizar con todos los tipos base. Esto no significa que los tipos primitivos sean clases o que las clases envoltorio (**wrapper**) no se tienen que utilizar ya. Como se menciono con anterioridad estas conversiones serán útiles cuando se utilicen las colecciones de datos.

```
Integer oint = 43;
Float ofloat = 125.8F
Boolean oboolean = false;
```

“Unboxing” es el procedimiento opuesto al **“autoboxing”**, se utiliza cuando es necesario volver a utilizar el valor, que con el **“autoboxing”**, se convirtió a objeto de la clase envoltorio. En versiones anteriores a la 5, se conseguía con las funciones **“xxxValue”**.

```
Integer oint = 125;
...
int i = oint;
```

La imagen siguiente ilustra los conceptos de "autoboxing" y "unboxing".

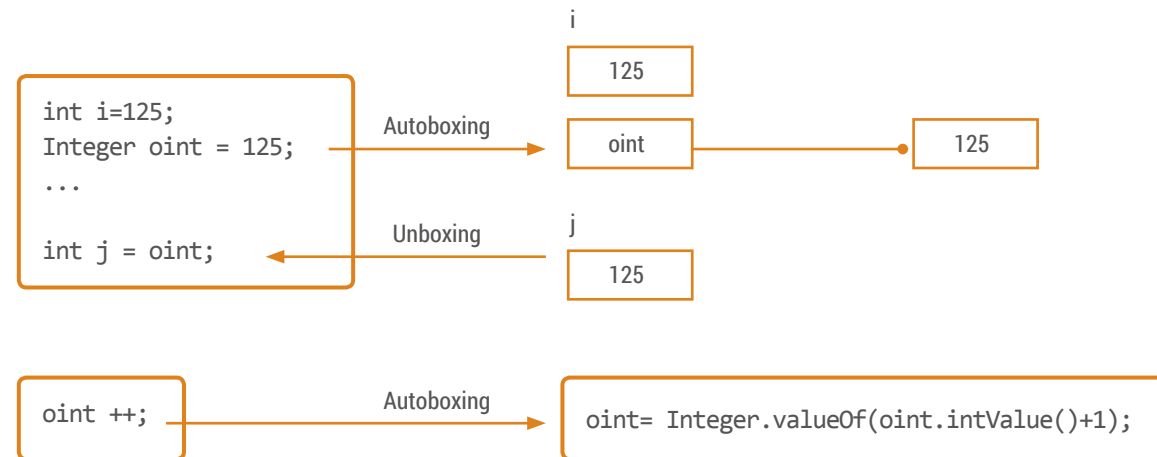


FIGURA 4.8: AUTOBOXING Y UNBOXING.

Telefonica

EDUCACIÓN DIGITAL