Prácticas de Programación

Tema 5. Aspectos avanzados de los tipos de datos

5.1 Clases envoltorio

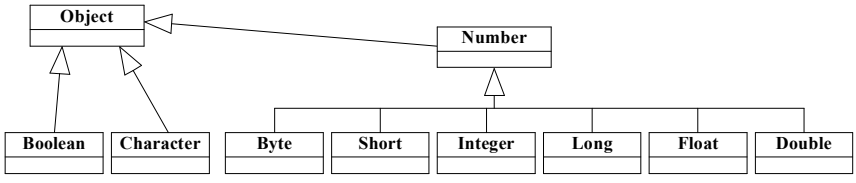
Java realiza una clara distinción entre *tipos primitivos* y objetos

Tipos primitivos:

- `int` (y sus variantes: `byte`, `short` y `long`)
- `float` y `double`
- `boolean`
- `char`

Para cada tipo primitivo existe una *clase envoltorio*

- **que almacena un valor de ese tipo (le sirve de envoltura)**



Utilidad de las clases envoltorio

- Proporcionan un "lugar" donde poner:
  - métodos de utilidad relacionados con el tipo primitivo
    - p.e.: conversiones de `String` y al tipo y viceversa
  - constantes relacionadas con el tipo primitivo
    - p.e.: máximo y mínimo valor que puede almacenar

- Permiten crear objetos que almacenan valores del tipo primitivo
  - que pueden ser usados en clases genéricas (p.e. `ArrayList`)

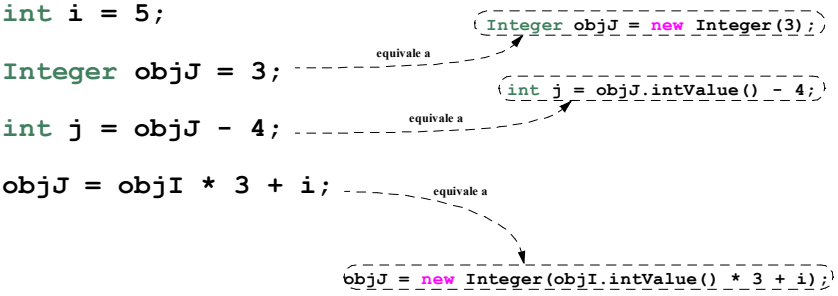
```
// ¡¡la siguiente línea es INCORRECTA!!
ArrayList<int> l=new ArrayList<int>();

// CORRECTO:
ArrayList<Integer> l=new ArrayList<Integer>();
```


Conversiones automáticas entre clases envoltorio y tipos primitivos

Sirven para facilitar el uso de las clases envoltorio

El código mostrado a continuación es correcto:

```
Integer objI = new Integer(2);
int i = 5;

Integer objJ = 3;

int j = objJ - 4;

objJ = objI * 3 + i;
```



5.2 Tipos enteros

Tipos primitivos enteros (`byte`, `short`, `int`, `long`):

Tipo	Tamaño	Mínimo valor	Máximo valor
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-2^{63}	2^{63}-1 (aprox. $9*10^{18}$)

Literales:

- `byte`, `short`, `int`: 12, -37, 037 (octal), 0x2A (hexadecimal)
- `long`: 12L, 12l

Clases envoltorio de los tipos enteros

Clases: `Integer`, `Byte`, `Short`, `Long`

- **Constructores:**

`Integer(int i)`
`Byte(byte b)`
`Short(short s)`
`Long(long l)`
	crea un objeto con el valor entero indicado

`Integer(String s)`
`Byte(String s)`
`Short(String s)`
`Long(String s)`
	crea un objeto transformando `s` a entero
	Lanza `NumberFormatException` si `s` no se puede convertir a entero

- **constantes estáticas:**

`MIN_VALUE`	Mínimo número entero	-2^{n-1} (Integer:-2^{31}=-2147483648)
`MAX_VALUE`	Máximo número entero	$2^{n-1}-1$ (Integer=$2^{31}-1$=2147483647)

- **operaciones estáticas:**

`int parseInt(String s)`
`int parseByte(String s)`
`int parseShort(String s)`
`int parseLong(String s)`
	Retorna la conversión de `s` a entero
	Lanza `NumberFormatException` si `s` no se puede convertir a entero

- **operaciones no estáticas:**

`int intValue()`	**Retorna la conversión del valor interno a entero**
`double doubleValue()`	**Retorna la conversión del valor interno a real**
`String toString()`	**Retorna la conversión del valor interno a texto**

Ejemplo: Cálculo del máximo y el mínimo

```
// retorna el valor máximo en el array
int buscaMáximo(int[] array) {
    int max= Integer.MIN_VALUE;
    for(int i=0; i<array.length; i++)
        if (array[i]>max)
            max=array[i]; // nuevo máximo
    return max;
}
// retorna el valor mínimo en el array
int buscaMínimo(int[] array) {
    int min = Integer.MAX_VALUE;
    for(int i=0; i<array.length; i++)
        if (array[i]<min)
            min=array[i]; // nuevo mínimo
    return min;
}
```

Ejemplo: uso de `parseInt`

```
public static void main(String[] args) {
  String strNum="34";
  int i;

  try {

    i = Integer.parseInt(strNum);
    int j = i * 2;
    System.out.println("i:" + i + "   j:" + j);

  } catch(NumberFormatException e) {
    System.out.println("Error: " + strNum +
              " no se puede convertir a int");
  }
}
```

Puede lanzar `NumberFormatException`

5.3 Tipos reales

Tipos predefinidos:

- `float`: nº real de 32 bits
  - unos 6 dígitos significativos
  - rango: `-3.4e38 ... +3.4e38`
- `double`: nº real de 64 bits
  - unos 15 dígitos significativos
  - rango: `-1.8e308 ... +1.8e308`

Literales:

- `float`: `18.0f`
- `double`: `0.0, 2.3E-5, 18.3d`

Clases envoltorio de los tipos reales

Clases: `Float`, `Double`

- **Constructores:**

`Float(float f)` `Double(double d)`	crea un objeto con el valor indicado
`Float(String s)` `Double(String s)`	crea un objeto transformando `s` a número Lanza `NumberFormatException` si `s` no se puede convertir

- **constantes estáticas:**

`MIN_VALUE`	Mínimo número real *¡positivo!*	`Double: 4.9e-324` `Float: 1.4e-45`
`MAX_VALUE`	Máximo número real	`Double: 1.8e+308` `Float: 3.4e38`

Ejemplo: Cálculo del máximo y el mínimo

```
// retorna el valor máximo en el array
double buscaMáximo(double[] array) {
    double max= -Double.MAX_VALUE;
    for(int i=0; i<array.length; i++)
        if (array[i]>max)
            max=array[i];
    return max;
}
// retorna el valor mínimo en el array
double buscaMínimo(double[] array) {
    double min = Double.MAX_VALUE;
    for(int i=0; i<array.length; i++)
        if (array[i]<min)
            min=array[i];
    return min;
}
```


Clases envoltorio de los tipos reales (cont.)

- **operaciones estáticas:**

`boolean isInfinite(float f)` `boolean isInfinite(double d)`	**indica si el número es o no infinito**
`boolean isNaN(float f)` `boolean isNaN(double d)`	**indica si el número es o no *NaN* (*Not A Number*)[a]**
`double parseDouble(String s)` `float parseFloat(String s)`	**Retorna la conversión de `s` a número real** **Lanza `NumberFormatException` si `s` no se puede convertir**

a. NaN indica que el resultado de una operación fue incorrecto (ej., raíz cuadrada de número negativo, 0/0, ...)

Clases envoltorio de los tipos reales (cont.)

- **operaciones no estáticas:**

`int intValue()`	Retorna la conversión del valor interno a entero
`double doubleValue()` `float floatValue()`	Retorna la conversión del valor interno a real
`String toString()`	Retorna la conversión del valor interno a texto

Ejemplo: Comprobación de división por cero

```
int n=0;
double suma=..., media;

media=suma/n;
if (Double.isInfinite(media)) {

    System.out.println("n es cero");

} else if (Double.isNaN(media)) {

    System.out.println("n y suma son cero");

} else {

    System.out.println("La media es "+media);

}
```

La división entre números reales (`double` o `float`) no lanza `ArithmeticException` al dividir por `0.0`

Errores de precisión y redondeo

Al operar con números reales es preciso tener en cuenta que se pueden producir errores de redondeo o de precisión

- **Resultado esperado de un cálculo:** `2.00000000000000`
- **Resultado posible:** `2.00000000000001`
- **Resultado posible:** `1.99999999999999`

Estos errores pueden ser acumulativos

Tienen mucha importancia en la comparación de igualdad

- **En general, nunca comparar dos nº reales para igualdad:**

```
if (x==2.0) ... // mal
if (Math.abs(x-2.0)<1.0e-10) //cercanía: OK
```


5.4 Conversiones entre tipos numéricos

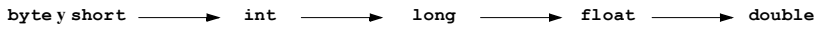
Java realiza *conversiones implícitas de tipos* numéricos en distintas situaciones:

- ***Promoción numérica*: cuando los operandos de una expresión numérica son de distintos tipos**
  - **p.e. sumamos un `int` con un `float`**
- ***Conversión durante la asignación*: cuando se asigna un valor de un tipo a una variable de otro tipo**
  - **p.e. asignamos el resultado de sumar dos variables de tipo `float` a un `double`**
- ***Conversión de parámetros de métodos*: cuando el tipo del parámetro no coincide con el valor que se pasa**
  - **p.e. un método que espera un `int` recibe un `byte`**

Promoción numérica

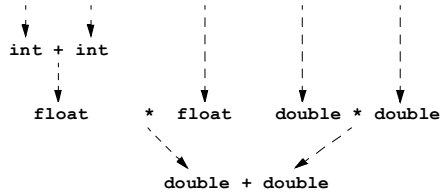
Ocurre cuando se mezclan operandos de distintos tipos en una operación

- **se produce una promoción implícita hacia el tipo que tiene mayor rango de valores**



Ejemplo:

```
double d; float f; int i; short s; byte b;
d = (s + b)  *  f  +  i  *  d
```



Conversión durante la asignación

Ocurre cuando la variable a la izquierda del "=" no es del mismo tipo que la expresión de la derecha

- **se convierte el resultado de la expresión al valor de la variable**
- **pero sólo si la variable es de un tipo más "amplio" que la expresión**
  - **en caso contrario es un error de compilación ("*possible loss of precision*")**

Ejemplos:

```
double d; float f; int i; short s; byte b;
i = s; // "s" se convierte a int
d = i*f+b; // el resultado de la expresión (float)
           // se convierte a double
i = f/d; // ¡ERROR! el tipo int es "más estrecho"
         // que double
```


Conversión de parámetros de métodos

Las reglas son iguales que las citadas para la conversión durante la asignación:

- **conversión implícita hacia tipos más "amplios"**

Ejemplo:

```
float f; int i;

double max(double d1, double d2, double d3) {...}

double dMax = max(i, f, 10);
// i, f y 10 se convierten a double
```

Conversión explícita (*casting*)

Para convertir de un tipo de mayor rango a otro de menor es necesario hacer una conversión explícita o *casting*

```
(tipo) expresión
```

Ejemplos:

```
double d=5.9; int i=7;
byte b = (byte)i; // asigna 7 a b
i = (int)d;       // asigna 5 a i
```

- **la conversión de números en coma flotante a enteros trunca el valor (no redondea)**

Atención: si el valor no cabe, ocurre un error no detectado

```
i = Byte.MAX_VALUE + 1;
d = Integer.MAX_VALUE+3;
b = (byte)i; // almacena -128 en b
i = (int)d;  // almacena -2147483646 en i
```


El *casting* tiene más precedencia que los operadores aritméticos (+,-,*,/)

- **Ejemplo: división de enteros con decimales**

```
int i1=100, i2=1000;

double d = i1/i2; // d=0.0
// se realiza la división entera y el resultado
// se convierte implícitamente a double

double d = (double)(i1/i2); // d=0.0
// se realiza la división entera y el resultado
// se convierte explícitamente a double

double d = (double)i1/i2; // d=0.1
// i1 se convierte a double antes de dividir, por
// lo que i2 es promocionado también a double
// antes de realizar la división
```

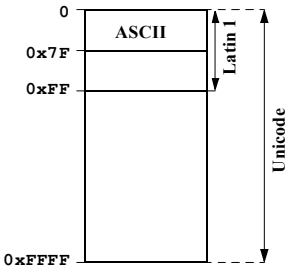

Ejemplo: conversión a entero de una expresión en coma flotante

```
double d=2.3; int i1=100, i2=1000;

int i = d/i1*i2; // error de compilación
// trata de convertir de double a int:
// "possible loss of precision"

int i = (int)d/i1*i2; // i=0
// convierte d a int antes de realizar las
// operaciones

int i = (int)(d/i1*i2); // i=23
// Divide d/i1 (con i1 promocionado a double)
// Después multiplica el resultado por i2 (con i2
//                          promocionado a double)
// Finalmente convierte el resultado a entero
```

5.5 Caracteres

Tipo primitivo `char`:

- **caracteres de 16 bits (formato *Unicode*)**
- **el código ASCII es un subconjunto del código *Unicode***



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

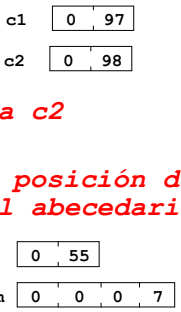
Caracteres (cont.)

Literales:

`'a'` (letra 'a')	`'2'` (dígito '2')	`'\t'` (tabulador)
`'\n'` (salto de línea)	`'\"'` (comillas dobles)	`'\''` (comilla simple)
`'\\'` (contrabarra)		

Java trata los caracteres como enteros

```
int i = 'a'; // asigna 97 a i
char c1 = 'a';
char c2 = (char)(c1+1); // asigna 'b' a c2
char c3 = 'c';
int pos = c3-'a'+1; // asigna 3 a pos: posición de
                    // la letra 'c' en el abecedario
char c4 = '7';
int num = c4-'0';
```



Clase envoltorio `Character`

Elementos:

- **constructores:**

`Character(char c)`	crea un objeto con el valor `c`

- **operaciones estáticas booleanas**

`boolean isDigit(char c)`	indica si `c` es o no un dígito numérico
`boolean isLetter(char c)`	indica si `c` es o no una letra
`boolean isLetterOrDigit(char c)`	indica si `c` es o no una letra o un dígito
`boolean isUpperCase(char c)` `boolean isLowerCase(char c)`	indica si `c` es o no una mayúscula o minúscula, respectivamente

Clase envoltorio `Character` (cont.)

- operaciones estáticas que retornan char:

`char toLowerCase(char c)`	retorna la conversión de `c` a minúscula
`char toUpperCase(char c)`	retorna la conversión de `c` a mayúscula

- operaciones no estáticas:

`char charValue()`	retorna el valor interno del objeto

5.6 Strings

La clase `String` permite *almacenar secuencias de caracteres*

- un *string* es *inmutable*: una vez que se crea ya no se puede modificar

Literales de *string*: `"esto es un string"`, `"una línea\n"`

Creación de objetos de la clase `String`

- normalmente se crean a partir de un literal de *string*:

```
String str = "abc";
```

- el lenguaje Java nos "oculta" el operador `new`, pero realmente la línea anterior es equivalente a:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```


Concatenación de *strings*

El operador "+" crea un *nuevo* objeto de la clase `String` resultado de añadir el *string* de la derecha al de la izquierda

```
String str1="uno", str2="dos";
String str3=str1+str2;
```

equivale a:

```
char data[] = {'u','n','o','d','o','s'};
String str3 = new String(data);
```

Se puede concatenar un *string* con otro objeto o tipo primitivo

- que se convierte a `String` utilizando su método `toString()`

```
str1=str2+2;
```

equivale a:

```
str1=str2+Integer.toString(2);
```

Ojo con los paréntesis:

```
"cuatro = " + 2 + 2   => "cuatro = 22"
"cuatro = " + (2+2)   => "cuatro = 4"
```

Algunos métodos de la clase `String`

Operaciones no estáticas

`int length()`	Longitud, en caracteres
`char charAt(num)`	Carácter individual (`num` empieza en cero)
`String substring (int startIndex, int endIndex)`	Rodaja de un string: desde `startIndex` hasta el *anterior* al `endIndex`
`String substring (int startIndex)`	Rodaja de un string: desde `startIndex` hasta el final
`boolean equals(String s)`	Comparación de igualdad con `s`
`boolean equalsIgnoreCase (String s)`	Idem, pero sin distinguir mayúsculas de minúsculas

Operaciones no estáticas

`int compareTo(String s)`	Compara por orden alfabético; retorna: `this < s` ⇨ <0 `this.equals(s)` ⇨ 0 `this > s` ⇨ >0
`String toLowerCase()`	Retorna la conversión a minúsculas
`String toUpperCase()`	Idem a mayúsculas
`String trim()`	Retorna el string sin espacios iniciales ni finales
`String[] split (String regex)`	Divide el string en trozos separados por `regex` (ver ejemplo en pág. 35)

Operaciones no estáticas de búsqueda

`int indexOf(char c)` `int indexOf(char c, int fromIndex)`	retorna el índice de la primera casilla que contiene `c`
`int lastIndexOf(char c)`	retorna el índice de la última casilla que contiene `c`
`int indexOf(String s)` `int indexOf(String s, int fromIndex)`	retorna el índice de la primera casilla donde empieza una rodaja igual a `s`
`int lastIndexOf(String s)`	retorna el índice de la última casilla donde empieza una rodaja igual a `s`
`boolean contains(String s)`	retorna `true` si el string contiene a `s`

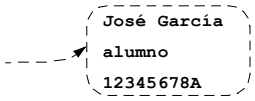
- Las funciones tipo "`indexOf`" retornan `-1` si no encuentran lo que se busca

Ejemplo: obtener primera y última palabras

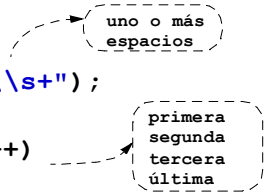
```
String frase=...;

// quita espacios del principio y del final
frase=frase.trim();

// busca el primer espacio
int posEspacio=frase.indexOf(' ');

// obtiene la primera palabra
String primera=frase.substring(0,posEspacio);

// busca el último espacio
posEspacio = str.lastIndexOf(' ');

// obtiene la última palabra
String última=str.substring(posEspacio+1);
```


Ejemplo: separar palabras con `split`

```
String línea="José García:alumno:12345678A";
// divide en trozos
String[] trozos = línea.split(":");
// muestra los trozos
for(int i=0; i<trozos.length; i++)
  System.out.println(trozos[i]);
```




```
String frase=" primera   segunda tercera  última";
// quita espacios del principio y del final
frase = frase.trim();
// divide en palabras
String[] palabras = frase.split("\\s+");
// muestra todas
for(int i=0; i<palabras.length; i++)
  System.out.println(palabras[i]);
```



Ejemplo: obtener número

```
String frase = " ... altura:4.56 ... ";
final String strAltura = "altura:";
// busca el comienzo del número
int posIni = frase.indexOf(strAltura);
posIni += strAltura.length();
// busca el final del número
int posFin = frase.indexOf(' ', posIni);
// convierte el string a número y le muestra
try {
  double altura = Double.parseDouble
                  (frase.substring(posIni,posFin));
  System.out.println("altura:"+altura);
} catch (NumberFormatException e) {
  System.out.println("Error de formato");
  System.exit(-1);
}
```

La clase `StringBuilder`

Similar a la clase `String`, pero

- **es *posible modificar* sus contenidos**

Tiene todos los métodos de `String` y alguno más:

`StringBuilder(String str)`	Constructor
`StringBuilder insert (int pos, char c)` `StringBuilder insert (int pos, String s)`	inserta el carácter o el string en la posición `pos`
`StringBuilder deleteCharAt (int pos)` `StringBuilder delete (int start, int end)`	elimina el carácter o el conjunto de caracteres indicados (hasta `end-1`)
`StringBuilder setCharAt (int pos, char c)`	cambia el carácter que ocupa la posición `pos`

Concatenación `StringBuilder` vs. `String`

- `String`: **se usa el operador "+"**
  - **es muy lento para strings grandes**

```
String str = "";
for (int i = 0; i < 20000; i++)
  str = str + "hola ";
```

tarda más de 3 segundos

- `StringBuilder`: **se usa el método `append`**
  - **mucho más rápido**

```
StringBuilder strB = new StringBuilder();
for (int i = 0; i < 20000; i++)
  strB.append("hola ");
```

¡sólo tarda 14 milisegundos!

Ejemplo de uso de la clase `StringBuilder`

```
StringBuilder str = new StringBuilder("mesilla");
System.out.println(str); // escribe: mesilla

str.delete(0,2);
System.out.println(str); // escribe: silla

str.setCharAt(1,'e');
System.out.println(str); // escribe: sella

str.insert(2,"mi");
System.out.println(str); // escribe: semilla
```

5.7 Arrays

Permiten guardar una tabla con muchos datos del mismo tipo

- **cada elemento se distingue por uno o varios índices enteros**

Disponen de un atributo `length` que indica cuántos elementos pueden almacenar

Existe una forma especial del lazo `for` para tratar arrays y colecciones de datos (lazo "`for-each`")

```
for (tipo elemento: nombreArray) {
  ...
}
```

- **en la variable `elemento` (del tipo `tipo`) se van *copiando* sucesivamente cada uno de los valores contenidos en el array**
- **debemos tener cuidado si pretendemos modificar los elementos del array con el lazo `for-each`**
  - **ver "Ejemplo: inicializador y lazo for-each" en la página 42**

Inicializadores

Pueden inicializarse en el momento de su creación:

```
String[] animales = {"oso", "lobo", "corzo"};
```

- **es equivalente a:**

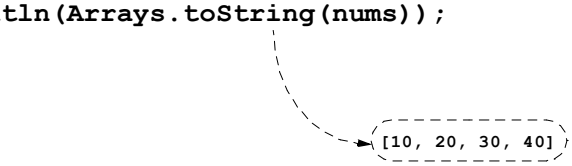
```
String[] animales = String[3];
animales[0]="oso";
animales[1]="lobo";
animales[2]="corzo";
```

También pueden inicializarse arrays de más de una dimensión:

```
double[][] matrizUnidad2x2 = { {1.0, 0.0},
                               {0.0, 1.0} };
```


Ejemplo: inicializador y lazo `for-each`

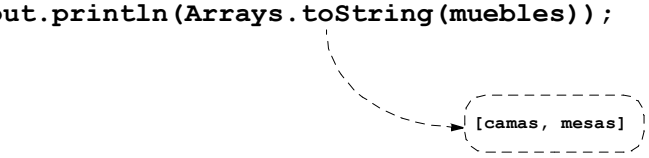
```
// lazo for-each y array de tipos primitivos
int[] nums={10,20,30,40};

// TRATA de sumar uno a cada componente del array
for(int i: nums)
  i = i + 1; // i es una COPIA de cada uno
             // de los elementos del array

System.out.println(Arrays.toString(nums));
```



Los contenidos del array NO se han modificado

Ejemplo: inicializador y lazo for-each (cont.)

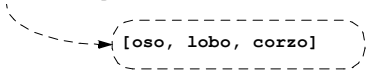
```
// lazo for-each y array de objetos
StringBuilder[] muebles = {
    new StringBuilder("cama"),
    new StringBuilder("mesa")};

// añade una s la final de cada elemento
for(StringBuilder str: muebles)
  str.append("s"); // str es una referencia a cada
                   // uno de los elementos del
                   // array

System.out.println(Arrays.toString(muebles));
```



Los contenidos del array SÍ se han modificado

Ejemplo: inicializador y lazo for-each (cont.)

```
String[] animales = {"oso", "lobo", "corzo"};

// TRATA de añadir una 's' al final de cada
// nombre para hacer el plural
for (String str: animales) {
  str = str + 's';
}

System.out.println(Arrays.toString(animales));
```



¡los contenidos del array NO se han modificado!

¿cuál es la razón?

Clase `Arrays`

Contiene métodos estáticos para manipular arrays

`int binarySearch (tipo[] a, tipo key)`	Búsqueda binaria. Retorna la posición que ocupa el elemento de valor `key` (y un número negativo si el elemento no existe) El array debe estar ordenado
`tipo[] copyOf (tipo[] original, int newLength)`	Retorna una copia del array `original`, truncado o completado con ceros dependiendo del valor del `newLength`
`boolean equals(tipo[] a1, tipo[] a2)`	Retorna `true` si los arrays son iguales (usa `equals` para comparar objetos)
`void fill(tipo[] a, tipo val)`	Rellena el array con el valor `val`
`void sort(tipo[] a)`	Ordena el array en orden creciente
`String toString(tipo[] a)`	Retorna un `String` con los contenidos del array en el formato: [3,67,8,234,-3,7]

- `tipo` puede ser cualquier tipo primitivo (o la clase `Object`)

Ejemplo de uso de la clase `Arrays`

```
import java.util.*;

public class PruArrays {
  public static void main(String[] args){
    int[] a = {4,30,-5,0,23,-7};
    System.out.println("a:"+Arrays.toString(a));
    // muestra "a:[4, 30, -5, 0, 23, -7]"
    Arrays.sort(a);
    System.out.println("a:"+Arrays.toString(a));
    // muestra "a:[-7, -5, 0, 4, 23, 30]"
    int[] a1 = Arrays.copyOf(a, 8);
    System.out.println("a1:"+Arrays.toString(a1));
    // muestra "a1:[-7, -5, 0, 4, 23, 30, 0, 0]"
  }
}
```


5.8 Tipos enumerados

Un tipo enumerado es aquel en el que los posibles valores son un conjunto finito de palabras

- **por ejemplo un Color (Rojo, Verde, Azul)**
- **o un día de la semana (lunes, martes,...)**

En lenguajes sin enumerados se usan constantes enteras

- **pero esta solución puede dar errores si se usan enteros no previstos**

```
final int ROJO=0;
final int VERDE=1;
final int AZUL=2;

...

void cambiaColor(int color) { ... }
```

Podríamos confundirnos y pasar un valor distinto de 0, 1 o 2

Declaración de clases enumeradas

Declarar una clase enumerada

```
public enum Nombre
   {palabra1, palabra2, palabra3, ...}
```

Ejemplo:

```
public enum Color {ROJO, VERDE, AZUL}

...

void cambiaColor(Color color) { ... }
```

Elementos de una clase enumerada

La clase `Color` tiene los siguientes elementos:

- variables estáticas: `ROJO, VERDE, AZUL`
- método estático `values()`, que retorna un array con todas las constantes del tipo enumerado
- método estático `valueOf(String s)` que retorna la conversión a enumerado del string `s`
  - lanza `IllegalArgumentException` si `s` no corresponde a ningún valor del tipo
- métodos `equals()`, `toString()`, etc.

Los tipos enumerados pueden usarse en una instrucción `switch`

- usando directamente los valores (ej.: `ROJO`, `VERDE`, ...)

Ejemplo

```
import fundamentos.*;

public class UsaColores
{
  // clase enumerada
  public enum Color {ROJO, VERDE, AZUL};

  public static void main(String[] args)
  {

    // mostrar la lista de todos los colores
    for(Color c: Color.values()) {
      System.out.println("Color "+c);
    }
```

```
Color ROJO
Color VERDE
Color AZUL
```



```
    // leer un valor del tipo enumerado
    Color c;
    Lectura l=new Lectura("Leer un color");
    l.creaEntrada("color","");
    boolean entradaCorrecta = false;
    do {
      try {
        l.esperaYCierra();
        String s=l.leeString("color");
        c=Color.valueOf(s);
        entradaCorrecta=true;
      } catch(IllegalArgumentException e) {
         Mensaje m = new Mensaje("Error");
         m.escribe("Color incorrecto");
      }
    } while(!entradaCorrecta);
```

```
      // usar el enumerado en un switch
      switch (c) {
        case ROJO:
        instrucciones para ROJO...;
        break;
        case VERDE:
        instrucciones para VERDE...;
        break;
        case AZUL:
        instrucciones para AZUL...;
        break;
        default:
        // buena práctica para detectar que nos hemos
        // olvidado de tratar uno de los valores
        throw new AssertionError("color inesperado");
      }
    }
}
```


5.9 Bibliografía

- Ken Arnold, James Gosling, David Holmes, “El lenguaje de programación Java”, 3ª edición. Addison-Wesley, 2000.
- Francisco Gutiérrez, Francisco Durán, Ernesto Pimentel. “Programación Orientada a Objetos con Java”. Paraninfo, 2007.
- Eitel, Harvey M. y Deitel, Paul J., “Cómo programar en Java”, quinta edición. Pearson Educación, México, 2004.
- King, Kim N. “Java programming: from the beginning”. W. W. Norton & Company, cop. 2000