



**Dept. de Ingeniería de Sistemas Telemáticos**  
**E.T.S.I. de Telecomunicación**  
**Universidad Politécnica**  
**Madrid**

# **Java**

## **Vademécum**

**José A. Mañas**  
**14 de diciembre de 2010**

# Índice

Introducción.....	9
Vademécum.....	10
1. abstract (palabra reservada) .....	10
2. Accesor [getter] (concepto).....	12
3. Álgebra de Boole (concepto).....	12
4. Algoritmo [algorithm] (concepto) .....	12
5. Ámbito [scope].....	12
6. API (acrónimo).....	13
7. Argumentos [arguments] (concepto).....	13
8. Arrays (concepto).....	13
9. arraycopy (método) java.lang.System.arraycopy(...) .....	20
10. Asignación [assignment] (concepto).....	20
11. Atributo (concepto).....	21
12. Autoboxing (concepto).....	21
13. Bloque de sentencias (concepto) .....	22
14. boolean (palabra reservada).....	22
15. Booleanos (concepto) .....	23
16. break (palabra reservada) .....	23
17. Bucles [loops] (concepto).....	24
18. Bugs (concepto).....	25
19. byte (palabra reservada) .....	25
20. Bytecode .....	26
21. Cabecera (de un método) (concepto).....	26
22. Campo [field] (concepto).....	26
23. Caracteres (concepto).....	27
24. case (palabra reservada) .....	27
25. casting (concepto) .....	27
26. catch (palabra reservada) .....	28
27. char (palabra reservada).....	28
28. Clases (concepto).....	29
29. class (palabra reservada) .....	31
30. Código [code] (concepto) .....	31
31. clone (método) protected Object clone() .....	31
32. Codificación de caracteres [encoding] .....	32
33. Colas [queues] (estructura de datos) .....	34
34. Comentarios [comments] (concepto) .....	34

35.	compareTo (método) .....	35
36.	Compilación [compilation] (concepto) .....	36
37.	Command Line Interface (CLI) .....	36
38.	Composición [composition] (concepto) .....	36
39.	Conjuntos [set] (estructuras de datos) .....	36
40.	Consola .....	36
41.	Constantes [constant] (concepto) .....	38
42.	Constructores [constructors] (concepto) .....	38
43.	continue (palabra reservada) .....	38
44.	Cortocircuito [short-circuit] (concepto) .....	39
45.	Cuerpo (de un método) [method body] (concepto) .....	40
46.	Debug (concepto) .....	40
47.	Declaración .....	40
48.	Delegación [delegation] (concepto) .....	42
49.	Desbordamiento [overflow] (concepto) .....	42
50.	do ... while (palabra reservada) .....	43
51.	Documentación [documentation] (concepto) .....	43
52.	double (palabra reservada) .....	45
53.	Downcasting (concepto) .....	46
54.	Edición [edition] (concepto) .....	46
55.	Ejecución [execution] (concepto) .....	46
56.	Ejecución condicional [conditional execution] (concepto) .....	48
57.	Elección dinámica de método (concepto) .....	48
58.	else (palabra reservada) .....	49
59.	Encapsulación [encapsulation] (concepto) .....	49
60.	enum (palabra reservada) .....	49
61.	Enumerados (concepto) .....	49
62.	Envoltorios [wrappers] (concepto) .....	51
63.	equals (método) public boolean equals(Object) .....	52
64.	Error (clase) java.lang.Error .....	55
65.	Errores (concepto) .....	55
66.	Estilo de codificación [coding style] .....	56
66.1.	Nombres .....	56
66.2.	Sangrado y espacio en blanco [layout] .....	56
67.	Etiquetas [labels] (concepto) .....	58
68.	Excepciones [exceptions] (concepto) .....	59
69.	Excepciones y métodos .....	61

70.	Exception (clase) java.lang.Exception.....	62
71.	Expresiones [expressions] (concepto) .....	64
72.	extends (palabra reservada).....	66
73.	Extensión (concepto) .....	67
74.	Fábricas [factories] (concepto).....	67
75.	Fichero fuente [source code file].....	68
76.	Ficheros .java.....	68
77.	final (palabra reservada).....	69
78.	finally (palabra reservada) .....	69
79.	float (palabra reservada).....	72
80.	for (palabra reservada) .....	72
81.	format (método) void format(String, Object ...)	75
82.	Friendly.....	75
83.	Genéricos [generics] (concepto) .....	76
84.	getClass (método) public Class getClass()	80
85.	Getter (concepto).....	81
86.	hashCode (método) public int hashCode()	81
87.	Herencia [inheritance] (concepto) .....	84
88.	Identificadores [identifiers] (concepto).....	87
89.	if (palabra reservada) .....	87
90.	Igualdad (==) .....	89
91.	Implementación (concepto) .....	91
92.	implements (palabra reservada).....	93
93.	import (palabra reservada) .....	94
94.	Inheritance (concepto).....	96
95.	Inicialización (concepto) .....	96
96.	instanceof (palabra reservada).....	97
97.	int (palabra reservada) .....	97
98.	Interfaz de programación (concepto) .....	97
99.	interface (palabra reservada).....	98
100.	Interfases (concepto) .....	98
101.	Interpretación (concepto).....	99
102.	jar.....	99
103.	java (herramienta estándar) .....	99
104.	javac (herramienta estándar).....	99
105.	javadoc (herramienta estándar).....	100
106.	JDK (acrónimo) .....	100

107.	JRE (acrónimo).....	101
108.	JVM (acrónimo) .....	101
109.	Keywords (palabras reservadas) .....	101
110.	Listas (estructura de datos) .....	101
111.	Listas encadenadas [linked lists] (estructura de datos).....	101
112.	long (palabra reservada) .....	103
113.	main (método) public static void main(String[]) .....	103
114.	Máquina virtual java (concepto) .....	104
115.	Método [method] (concepto).....	104
116.	Miembro [member] (concepto) .....	112
117.	new (palabra reservada) .....	112
118.	null (palabra reservada).....	113
119.	Números (concepto).....	113
120.	Objetos [objects] (concepto).....	116
121.	Ocultación [hiding] (concepto).....	117
122.	OO (acrónimo) .....	117
123.	OOP (acrónimo).....	117
124.	Operadores (concepto) .....	118
125.	Overflow .....	118
126.	Overloading.....	118
127.	@Override.....	118
128.	package (palabra reservada) .....	118
129.	Palabras reservadas [keywords].....	118
130.	Paquete [package] (concepto).....	119
131.	Parámetros (concepto).....	120
132.	Pilas [stacks] (estructura de datos) .....	120
133.	Polimorfismo [polimorphism] (concepto) .....	121
134.	print (método) void print(...) .....	122
135.	printf (método) void printf(String, Object ...) .....	122
136.	println (método) void println(...) .....	122
137.	private (palabra reservada) .....	122
138.	Programación orientada a objetos (concepto).....	123
139.	Programación estructurada (concepto).....	123
140.	Programa [program] (concepto).....	123
141.	Promoción [widening] (concepto) .....	123
142.	protected (palabra reservada) .....	124
143.	public (palabra reservada) .....	124

144.	Recursión (concepto).....	125
145.	Redefinición de métodos [method overriding] (concepto).....	125
146.	Reducción [narrowing] (concepto).....	128
147.	Refactoring (concepto).....	129
148.	Referencias [references] (concepto) .....	134
149.	return (palabra reservada).....	135
150.	RuntimeException (clase) java.lang.RuntimeException.....	135
151.	SDK (acrónimo) .....	135
152.	Setters (concepto) .....	135
153.	short (palabra reservada) .....	136
154.	Signatura (de un método) (concepto).....	136
155.	Sobrecarga de nombres [name overloading] (concepto) .....	136
156.	static (palabra reservada).....	137
157.	subclases (concepto) .....	137
158.	subtipo (concepto) .....	137
159.	super (palabra reservada) .....	138
160.	superclases (concepto).....	139
161.	superipo (concepto) .....	139
162.	sustitución (concepto).....	139
163.	switch (palabra reservada) .....	139
164.	this (palabra reservada) .....	141
165.	throw (palabra reservada) .....	142
166.	throws (palabra reservada) .....	142
167.	Tipos abstractos de datos (TAD) (concepto) .....	142
168.	Tipos formales [type parameters] .....	142
169.	Tipos primitivos [primitive data types] .....	142
170.	toString (método) public String toString() .....	142
171.	try catch finally (palabras reservadas) .....	143
172.	Underflow (concepto).....	143
173.	Unicode (concepto).....	144
174.	Upcasting (concepto) .....	145
175.	Variables [variables] (concepto).....	145
176.	Visibilidad [scope].....	150
177.	void (palabra reservada) .....	153
178.	while (palabra reservada).....	153
	Las Bibliotecas de Java.....	155
1.	ArrayList<E> (clase) java.util.ArrayList<E> .....	155

2.	Arrays (clase) java.util.Arrays .....	155
3.	Boolean (clase) java.lang.Boolean.....	156
4.	Calendar (clase) .....	156
5.	Character (clase) java.lang.Character .....	159
6.	Collator (clase) java.text.Collator .....	159
7.	Collection<E> (interface) java.util.Collection<E> .....	160
8.	Comparable<T> (interface) java.lang.Comparable<T> .....	161
9.	Comparator<T> (interface) java.util.Comparator<T> .....	162
10.	Date (clase) java.util.Date.....	163
11.	Double (clase) .....	164
12.	Enumeration<E> (interface) java.util Enumeration .....	164
13.	EnumSet (clase) java.util.EnumSet.....	164
14.	File (clase) java.io.File.....	165
15.	Formatter (clase) java.util.Formatter.....	166
16.	HashMap<K, V> (clase) java.util.HashMap<K, V> .....	171
17.	HashSet<E> (clase) java.util.HashSet<E> .....	171
18.	InputStream (clase abstracta) java.io.InputStream.....	171
19.	Integer (clase) java.lang.Integer .....	173
20.	Iterable<T> (interface) java.lang.Iterable<T> .....	173
21.	Iterator<E> (interface) java.util.Iterator<E> .....	174
22.	LinkedList<E> (clase) java.util.LinkedList<E> .....	178
23.	List<E> (interface) java.util.List<E> .....	178
24.	Map<K, V> (interface) java.util.Map<K, V> .....	181
25.	Math (clase) java.lang.Math.....	185
26.	Object (clase) java.lang.Object.....	186
27.	OutputStream (clase abstracta) java.io.OutputStream .....	187
28.	Properties (clase) java.util.Properties .....	189
29.	Random (clase) java.util.Randon .....	192
30.	Reader (clase abstracta) java.io.Reader .....	192
31.	Scanner (clase) java.util.Scanner.....	194
32.	Set<E> (interface) java.util.Set<E> .....	196
33.	SortedMap<K, V> (clase) java.util.SortedMap .....	198
34.	Stacks (concepto).....	198
35.	String (clase) java.lang.String.....	198
36.	StringBuffer (clase) java.lang.StringBuffer .....	200
37.	StringBuilder (clase) java.lang.StringBuilder.....	202
38.	System.err .....	203

39.	System.in .....	203
40.	System.out .....	204
41.	TreeMap<K, V> (clase) java.util.TreeMap<K, V> .....	205
42.	TreeSet<E> (clase) java.util.TreeSet<E> .....	205
43.	Vector<E> (clase) java.util.Vector<E> .....	205
44.	Writer (clase abstracta) java.io.Writer .....	205
Diccionario.....		208
1.	Acrónimos .....	208
2.	Términos en inglés .....	208



## Introducción

(Del latín *vade*, anda, ven, y *mecum*, conmigo).

1. m. Libro de poco volumen y de fácil manejo para consulta inmediata de nociones o informaciones fundamentales.
2. m. Cartapacio en que los niños llevaban sus libros y papeles a la escuela.

Java es un lenguaje de programación orientado a objetos, como otros muchos, con los que comparte los conceptos fundamentales. Pero, además de los conceptos, cada lenguaje de programación conlleva multitud de pequeños detalles que a menudo se olvidan.

Estas notas repasan los detalles conceptuales y sintácticos de Java. Se han ordenado alfabéticamente para que sean fáciles de localizar. Cada entrada se explica someramente y se incluyen pequeños ejemplos ilustrativos de cómo se usa.

Algunas entradas se refieren a clases de la biblioteca java; en estos casos se comentan los elementos más frecuentemente utilizados. Para mayor detalle, hay que recurrir a la especificación completa.

## Derechos de autor

© 2005-2010, José A. Mañas <jmanas@dit.upm.es>

El autor cede a los lectores el derecho de reproducir total o parcialmente cualquier parte de este documento siempre y cuando se cite la fuente y la reproducción sea fiel al original o se indique que ha sido modificado con las palabras "*inspirado en*".

El autor cede a los lectores el derecho de utilizar el código incluido en los ejemplos, incluso comercialmente, siempre y cuando se cite la fuente. No obstante, el autor declina responsabilidad alguna respecto de la idoneidad del código para fin alguno o la posible existencia de errores que pudieran repercutir en sus usuarios. Cualquier uso del código que aparece en este documento será bajo la entera responsabilidad del usuario del mismo.

# Vademécum

## 1. *abstract (palabra reservada)*

Hay métodos abstractos y clases abstractas.

Los métodos abstractos deben estar en clases abstractas.

Las clases abstractas pueden tener métodos abstractos.

### Métodos abstractos

---

Se dice que un [método](#) es "**abstract**" cuando se proporciona su [signatura](#), resultado y posibles excepciones; pero no su cuerpo. Es decir, cuando se indica cómo usarlo, pero no se proporciona el código que lo materializa.

```
abstract String cifra(String texto, Clave clave);
```

### Clases abstractas

---

Se dice que una clase es "**abstract**" cuando alguno de sus métodos es "**abstract**".

Las clases "**abstract**":

- no permiten generar objetos; es decir, no se puede hacer un "[new](#)"
- pueden tener [métodos](#) "**abstract**" y métodos normales
- pueden [extenderse](#); si la nueva clase proporciona el cuerpo de todos los métodos, será una clase normal, de lo contrario seguirá siendo "**abstract**"

Aunque conceptualmente sean cosas distintas, formal o sintácticamente podemos decir que una "[interface](#)" es una clase "abstract" en la que:

- todos los métodos son "**public abstract**"; de hecho no hay ni que decirlo
- todos los campos son "**public static final**"; de hecho no hay ni que decirlo

### Subclases abstractas

---

Cuando una clase abstracta se extiende, pero no se proporciona el cuerpo de algún método, heredado o propio.

## Ejemplo

```
public abstract class Punto {

    public abstract double getX();

    public abstract double getY();

    public abstract double getModulo();

    public abstract double getAngulo();

    public double distancia(Punto p) {
        double dx = getX() - p.getX();
        double dx2 = dx * dx;

        double dy = getY() - p.getY();
        double dy2 = dy * dy;

        return Math.sqrt(dx2 + dy2);
    }
}

public class PuntoC extends Punto {
    private double x;
    private double y;

    public double getX() { return x; }

    public double getY() { return y; }

    public double getModulo() { return Math.sqrt(x * x + y * y); }

    public double getAngulo() { return Math.atan2(y, x); }

    public PuntoC(double x, double y) {
        this.x= x;
        this.y= y;
    }

    public String toString() {
        return "cartesianas: " + x + ", " + y;
    }
}
```

```

public class PuntoP extends Punto {
    private double m;
    private double a;

    public double getX() { return m * Math.cos(a); }

    public double getY() { return m * Math.sin(a); }

    public double getModulo() { return m; }

    public double getAngulo() { return a; }

    public PuntoP(double m, double a) {
        this.m = m;
        this.a = a;
    }

    public String toString() {
        return "polares: " + m + ", " + a;
    }
}

```

## 2. **Accesor [getter] (concepto)**

Ver "[métodos / métodos de acceso \(getters\)](#)".

## 3. **Álgebra de Boole (concepto)**

Ver "[boolean](#)".

## 4. **Algoritmo [algorithm] (concepto)**

Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

Para hallar la solución de un cierto problema, pueden haber multitud de algoritmos que, siendo todos correctos, requieran diferente esfuerzo, tiempo de cálculo o datos intermedios.

Los algoritmos, cuando se codifican en un lenguaje de programación, se convierten en [programas](#).

Un mismo algoritmo puede programarse de múltiples maneras, dependiendo del programador o del problema concreto al que se aplique. Cuando el problema a resolver es pequeño, suele ser indiferente emplear uno u otro de los algoritmos conocidos para resolverlo; pero cuando el problema crece, las diferencias de esfuerzo requerido por diferentes algoritmos puede llevar a programas que, siendo todos correctos, tarden más o menos tiempo, o requieran más o menos memoria para ejecutar. Por ello, en aplicaciones reales conviene elegir cuidadosamente el algoritmo que se va a programar.

## 5. **Ámbito [scope]**

Zona del texto de un programa donde un elemento es visible; es decir, donde se puede utilizar. Se aplica a [variables](#), [métodos](#) y [clases](#).

Ver "[visibilidad](#)".

Ver "[ocultación](#)" (homónimos en ámbitos anidados).

Ver "[redefinición](#)" (mecanismo de herencia).

## 6. **API (acrónimo)**

*Application Programming Interface*. Interfaz de programación.

## 7. **Argumentos [arguments] (concepto)**

Es la colección de variables que se le pasan a un [método](#) para que ejecute. Permiten parametrizar su comportamiento adecuando la ejecución a lo que interesa en cada momento.

```
double suma(int a, double b) { return a + b; }
```

```
double c = suma(2, 3.14);
```

### **argumentos formales [formal arguments]**

Se denomina así a las variables que aparecen en la cabecera del método.  
En el ejemplo de arriba: a y b.

### **argumentos reales [actual arguments]**

Se denomina así a los valores concretos que se usan en la llamada al método.  
En el ejemplo de arriba: 2 y 3.14.

Ver "[métodos](#)".

## 8. **Arrays (concepto)**

Son colecciones de objetos numerados, todos del mismo tipo. Pueden ser

- unidimensionales (en álgebra se suelen llamar vectores),
- bidimensionales (en álgebra se suelen llamar matrices),
- tridimensionales (a veces se les llaman tensores)
- de un número superior de dimensiones

Por ser más precisos, un "array" multidimensional se concibe como un "vector de vectores" lo que permite que cada vector sea de un tamaño diferente.

Así, una matriz puede verse como una serie de filas, cada una con una serie de columnas. Como no todas las filas tienen que tener el mismo número de columnas, la matriz no necesariamente tiene que ser rectangular.

Para poder utilizar un array hay que seguir unos ciertos pasos:

1. declararlo: nombre del array, número de dimensiones y tipo de datos que contiene
2. crearlo o ubicar memoria (*memory allocation*): número de datos que contiene
3. se puede acceder al array: lectura y escritura

### **declaración**

---

En la declaración se identifica el nombre, el número de dimensiones y el tipo de datos.

```
int[] vector;           // vector de enteros
double[][] matriz;      // matriz de reales
Tipo[]...[] variable;   // N dimensiones
```

Sintaxis alternativa (para programadores de C)

```
int vector[];           // vector de enteros
double matriz[][];      // matriz de reales
Tipo variable[]...[];   // N dimensiones
```

Un array declarado vale NULL hasta que se [cree](#).

## creación

---

Tras [declarar](#) un array hay que crearlo, lo que supone asignar memoria en todas y cada una de sus dimensiones:

```
vector = new int[100]; // 0..99
matriz = new double[2][3];
```

Se pueden crear arrays de tamaño 0.

[Declaración](#) y creación pueden llevarse a cabo juntas:

```
int[] vector = new int[100]; // 0..99
double[][] matriz = new double[2][3];
```

Una vez creados de un cierto tamaño, no se puede cambiar el tamaño.

Al crear el array, sus elementos reciben un valor por defecto que depende de su tipo:

- enteros: valor 0
- reales: valor 0.0
- booleanos: valor **false**
- caracteres: valor **(char)0**
- objetos: valor **null**

En el siguiente ejemplo se crea una matriz con un número aleatorio de filas y columnas:

```
/**
 * Crea una matriz aleatoria.
 * Número aleatorio de filas (1 .. 10).
 * Número aleatorio de columnas en cada fila (1 .. 10).
 * Datos aleatorios en cada posición (-9 .. +99).
 */
static int[][] crea() {
    Random random = new Random();
    int filas = 1 + random.nextInt(9);
    int[][] resultado = new int[filas][];
    for (int i = 0; i < filas; i++) {
        int columnas = 1 + random.nextInt(9);
        resultado[i] = new int[columnas];
        for (int j = 0; j < columnas; j++)
            resultado[i][j] = -9 + random.nextInt(109);
    }
    return resultado;
}
```

La dimensión de un array viene dada por el “campo” length, que puede ser diferente en diferentes filas:

```
vector.length

matriz.length      // número de filas
matriz[0].length   // columnas de la primera fila
matriz[1].length   // columnas de la segunda fila
```

### creación con inicialización

---

```
int[] primos = { 2, 3, 5, 7, 11, 13, 17 };

String[] dialogo = { "hola", "adiós" };

int[][] matriz = {
    { 1, 2, 3 },
    { 4, 5, 6 }
};

int[][] tianguloPascal = {
    { 1, 1 },
    { 1, 2, 1 },
    { 1, 3, 3, 1 },
    { 1, 4, 6, 4, 1 }
};
```

Se puede inicializar con cualquier expresión:

```
int[] cuadrados = { 1*1, 2*2, 3*3, 4*4 };

String nombre = "Pepito";
String[] dialogo = {
    "Hola D. " + nombre,
    "Adiós D. " + nombre
};
```

### acceso

---

Si un "array" unidimensional a tiene N miembros, al primero se accede escribiendo "a[0]", mientras que al último se accede escribiendo "a[N-1]". El acceso puede ser

- para lectura (extraer el valor); por ejemplo "System.out.println(a[3]);"
- para escritura (cargar un valor); por ejemplo "a[7] = 99;"

Cuando un "array" tiene varias dimensiones, hay que poner tantos índices entre corchetes como dimensiones tiene.

- en una matriz, el elemento de la primera fila y primera columna es en "x[0][0]"; es mero convenio decir que las filas son el primer índice o el segundo

Si se intenta acceder a un elemento del array fuera de su rango (0 – array.length-1), se provoca una [ArrayIndexOutOfBoundsException](#).

### recorrido

---

Es muy frecuente recorrer los términos de un array, visitando todos y cada uno de sus elementos en orden. Se puede utilizar un bucle con contador o iterar sobre los elementos:

<i><b>recorrido de un array</b></i>
<pre>int maximo = Integer.MIN_VALUE;</pre>

<i>recorrido de un array</i>
<pre>for (int i = 0; i &lt; vector.length; i++) {     if (vector[i] &gt; maximo)         maximo = vector[i]; }</pre>
<pre>int maximo = Integer.MIN_VALUE; for (int n: vector) {     if (n &gt; maximo)         maximo = n; }</pre>

Otro ejemplo: impresión de una matriz no cuadrada, fila por fila:

<pre>int[][] matriz = ...; for (int[] fila : matriz) {     for (int dato : fila)         System.out.print(dato + " ");     System.out.println(); }</pre>
--

### **paso de valores**

Un array se trata a todos los efectos como un objeto, pasándose referencias entre variables:

- cuando se llama a un método y se le pasa un array, el método hace su copia de la referencia; pero comparte el array

<i>copia de arrays</i>	<i>ejecución</i>
<pre>void caso1(int[] x) {     x[0] *= 10; }</pre>	
<pre>void test1() {     int[] a = new int[]{1, 2, 3};     System.out.println(Arrays.toString(a));     caso1(a);     System.out.println(Arrays.toString(a)); }</pre>	<pre>[1, 2, 3] [10, 2, 3]</pre>

### **copia de arrays**

Cuando una variable de tipo array se hace igual a otro, se copia la referencia; pero se comparte el array:

<i>copia de arrays</i>	<i>ejecución</i>
<pre>void copia1() {     int[] a = new int[]{1, 2, 3};     System.out.println(Arrays.toString(a)); }</pre>	<pre>[1, 2, 3]</pre>



<pre>int[] b = a; System.out.println(Arrays.toString(b)); a[0] *= 10; System.out.println(Arrays.toString(b)); }</pre>	<pre>[1, 2, 3] [10, 2, 3]</pre>
---	---------------------------------

Si no basta con compartir la referencia, sino que se necesita otra copia de un array, se puede recurrir al método [clone\(\)](#). Si los elementos del array son de un tipo primitivo, se copia su valor. Si son objetos, se copia la referencia, compartiéndose el objeto.

<b><i>copia de arrays</i></b>	<b><i>ejecución</i></b>
<pre>void copia2() {     int[] a = new int[]{1, 2, 3};     System.out.println(Arrays.toString(a));     int[] b = a.clone();     System.out.println(Arrays.toString(b));     a[0] *= 10;     System.out.println(Arrays.toString(b)); }</pre>	<pre>[1, 2, 3] [1, 2, 3] [1, 2, 3]</pre>
<pre>void copia2Objetos() {     Punto[] a = new Punto[]         {new Punto(1, 2), new Punto(3, 4)};     System.out.println(Arrays.toString(a));     Punto[] b = a.clone();     System.out.println(Arrays.toString(b));     a[0].multiplica(-1);     System.out.println(Arrays.toString(b)); }</pre>	<pre>[(1,2), (3,4)] [(1,2), (3,4)] [(-1,-2), (3,4)]</pre>
<pre>class Punto {     private int x, y;      public Punto(int x, int y) {         this.x = x;         this.y = y;     }      public void multiplica(int factor) {         x *= factor;         y *= factor;     }      public String toString() {         return String.format("(%d, %d)", x, y);     } }</pre>	

La misma situación ocurre cuando tenemos arrays de dos o más dimensiones (efectivamente, se trata de arrays de arrays):

<b><i>copia de arrays</i></b>	<b><i>ejecución</i></b>
<pre> void copia2ArraysBiDi() {     int[][] a = new int[][]{{1, 2}, {3, 4}};     System.out.println(Arrays.deepToString(a));     int[][] b = a.clone();     System.out.println(Arrays.deepToString(b));     a[0][0] *= -1;     System.out.println(Arrays.deepToString(b)); } </pre>	<pre> [[1, 2], [3, 4]] [[1, 2], [3, 4]] [[-1, 2], [3, 4]] </pre>

Se puede usar el método estándar “[arraycopy](#)”:

```

T[] a = ...;
T[] b = new T[a.length];
System.arraycopy(a, 0, b, 0, a.length);

```

que es realmente equivalente al método clone(), con los mismos problemas cuando se trata de arrays de objetos:

<b><i>copia de arrays</i></b>	<b><i>ejecución</i></b>
<pre> void copia20() {     int[] a = new int[]{1, 2, 3};     System.out.println(Arrays.toString(a));     int[] b = new int[a.length];     System.arraycopy(a, 0, b, 0, a.length);     System.out.println(Arrays.toString(b));     a[0] *= 10;     System.out.println(Arrays.toString(b)); } </pre>	<pre> [1, 2, 3] [1, 2, 3] [1, 2, 3] </pre>
<pre> void copia21Objetos() {     Punto[] a = new Punto[]         {new Punto(1, 2), new Punto(3, 4)};     System.out.println(Arrays.toString(a));     Punto[] b = new Punto[a.length];     System.arraycopy(a, 0, b, 0, a.length);     System.out.println(Arrays.toString(b));     a[0].multiplica(-1);     System.out.println(Arrays.toString(b)); } </pre>	<pre> [(1,2), (3,4)] [(1,2), (3,4)] [(-1,-2), (3,4)] </pre>

Se puede usar el método estándar copyOf() de [Arrays](#):

```

T[] a = ...;
T[] b = Arrays.copyOf(a, a.length);

```

que es realmente equivalente al método clone(), con los mismos problemas cuando se trata de arrays de objetos:

<b><i>copia de arrays</i></b>	<b><i>ejecución</i></b>
<pre>void copia30() {     int[] a = new int[]{1, 2, 3};     System.out.println(Arrays.toString(a));     int[] b = Arrays.copyOf(a, a.length);     System.out.println(Arrays.toString(b));     a[0] *= 10;     System.out.println(Arrays.toString(b)); }</pre>	<pre>[1, 2, 3] [1, 2, 3] [1, 2, 3]</pre>
<pre>void copia31Objetos() {     Punto[] a = new Punto[]         {new Punto(1, 2), new Punto(3, 4)};     System.out.println(Arrays.toString(a));     Punto[] b = Arrays.copyOf(a, a.length);     System.out.println(Arrays.toString(b));     a[0].multiplica(-1);     System.out.println(Arrays.toString(b)); }</pre>	<pre>[(1,2), (3,4)] [(1,2), (3,4)] [(-1,-2), (3,4)]</pre>

Por último, la copia se puede programar explícitamente, que es probablemente la forma más segura de trabajar:

```
T[] a = ...;
T[] b = new T[a.length];
for (int i = 0; i < a.length; i++)
    b[i] = a[i];
```

### **valores de tipo array**

Se pueden generar directamente valores que son arrays, tanto para cargar una variable como para pasárselos a un método (como argumento de llamada):

```
vector = new int[] { 2, 4, 6, 8, 16, 32, 64, 128 };

opciones(new String[] { "si", "no" });

new Triangulo(new Punto[] {new Punto(0, 0),
                           new Punto(2, 0),
                           new Punto(1, 1) } );
```

### **¿arrays o listas?**

Los arrays son de tamaño fijo, mientras que las [listas](#) son de tamaño variable.

Si no sabemos el tamaño de un array al crearlo, tenemos 2 opciones

1. crearlo muy grande, de forma que nos quedan los datos en el peor caso posible; el precio que pagamos es desperdiciar espacio
2. crearlo de un tamaño reducido, pero prever que si llegan más datos habrá que ampliarlo (o sea, crea un array mayor y copiar los datos); el precio que pagamos es tiempo de ejecución.

Las listas son una forma cómoda de aplicar la segunda opción.

Ver "[listas](#)".

Ver [list frente a arrays](#).

## 9. *arraycopy (método) java.lang.System.arraycopy(...)*

Este método sirve para copiar unos cuantos datos de un [array](#) en otro.

```
void arraycopy(Object origen, int p1,  
               Object destino, int p2, int n)
```

copia “n” datos del array “origen” al array “destino”. El primer dato que se copia del origen es el que esté en la posición “p1”, que va a la posición “p2” del destino.

Los arrays deben existir y disponer de suficientes datos en origen y espacio en destino para trasladar los n valores solicitados. Además, los objetos en “origen” deben poder asignarse al array “destino” (es decir, ser de tipos compatibles en asignación).

arraycopy es cuidadoso con los casos en que origen y destino sean el mismo array.

Los siguientes fragmentos de código son equivalentes en cuanto a su función; pero “arraycopy” es notablemente más rápido:

```
void arraycopy(Object origen, int p1, Object destino, int p2, int n)  
  
public class A {}  
public class B extends A {}  
  
public void copiador(B[] origen, int p1, A[] destino, int p2, int n) {  
    A[] temporal = new A[n];  
    for (int k = 0; k < n; k++)  
        temporal[k] = origen[p1 + k];  
    for (int k = 0; k < n; k++)  
        destino[p2 + k] = temporal[k];  
}
```

Ver "[arraycopy\(Object, int, Object, int, int\)](#)".

Ver "[copia de arrays](#)".

## 10. *Asignación [assignment] (concepto)*

Se llaman sentencias de asignación a las que cargan un nuevo valor en una variable:

```
variable = expresión ;
```

El tipo de la variable debe ser

- igual al de la [expresión](#)
- en tipos primitivos:
  - assignable por promoción (ver "[promoción](#)")
  - assignable por reducción (ver "[reducción](#)")
- en objetos:
  - assignable por [upcasting](#) (ver "[casting](#)")
  - assignable por [downcasting](#) (ver "[casting](#)")

### Notación compacta

---

Java proporciona una notación más compacta para algunas asignaciones típicas

forma compacta	forma equivalente
<code>x += expr;</code>	<code>x = x + expr;</code>
<code>x -= expr;</code>	<code>x = x - expr;</code>
<code>x *= expr;</code>	<code>x = x * expr;</code>
<code>x /= expr;</code>	<code>x = x / expr;</code>
<code>x %= expr;</code>	<code>x = x % expr;</code>
<code>x &amp;= expr;</code>	<code>x = x &amp; expr;</code>
<code>x  = expr;</code>	<code>x = x   expr;</code>
<code>x ++</code>	<code>x = x + 1</code>
<code>++x</code>	
<code>x --</code>	<code>x = x - 1</code>
<code>--x</code>	

## Incrementos y decrementos

La notación "++" y "--", aparte de incrementar y decrementar, respectivamente, el valor contenido en la variable, se puede usar como una expresión en sí misma, devolviendo bien el valor antes de la operación, o el valor después de la operación.

Veamos un ejemplo:

pre- y post- / incremento y decremento		
programa	imprime	valor de "x"
<code>int x = 0;</code> <code>System.out.println( x++ );</code>	0	de 0 pasa a 1
<code>int x = 0;</code> <code>System.out.println( ++x );</code>	1	de 0 pasa a 1
<code>int x = 0;</code> <code>System.out.println( x-- );</code>	0	de 0 pasa a -1
<code>int x = 0;</code> <code>System.out.println( --x );</code>	-1	de 0 pasa a -1

## 11. Atributo (concepto)

Ver "[campo](#)".

## 12. Autoboxing (concepto)

Java dispone de una serie de [tipos primitivos](#) y de objetos [envoltorio](#) para cada uno de ellos. Por ejemplo, los enteros se pueden manejar como el tipo primitivo "int" o como el envoltorio "Integer".

Cuando se necesita convertir entre uno y otro formato, Java lo hace automáticamente:

conversión explícita	autoboxing
<pre>void inserta(List&lt;Integer&gt; lista,             int valor) {     lista.add(new Integer(valor)); }  int extrae(Lista&lt;Integer&gt; lista) {     Integer x = lista.extract(0);     return x.intValue(); }</pre>	<pre>void inserta(List&lt;Integer&gt; lista,             int valor) {     lista.add(valor); }  int extrae(Lista&lt;Integer&gt; lista) {     return lista.extract(0); }</pre>

### 13. Bloque de sentencias (concepto)

Se denomina bloque a un conjunto de sentencias entre llaves.

```
{
    double dx = p.x - q.x;
    double dy = p.y - q.y;
    distancia = Math.sqrt(dx*dx + dy*dy);
}
```

Un bloque define un [ámbito](#):

- las [variables](#) que se declaran dentro de un bloque no existen fuera del bloque: aparecen al entrar "{" y desaparecen al terminar "}"
- si existiera una variable del mismo nombre fuera del bloque, queda oculta por la interna; es decir, es inaccesible

Muchas construcciones sintácticas de Java se basan en bloques:

- declaración de [interfaces](#), [clases](#) y [tipos enumerados](#)
- declaración de [métodos](#)
- fragmentos de sentencias condicionales: [if](#), [while](#), etc.

### 14. boolean (palabra reservada)

Tipo de datos que ya viene definido en el lenguaje. Se dice que es un [tipo primitivo](#).

Valores:

- true
- false

OJO: Siempre se escriben en minúsculas.

Operaciones:

- conjunción lógica (es español, Y; en ingles, AND): el resultado es "true" si y sólo si ambos operandos son "true"
  - `x & y` siempre se evalúan "x" e "y"
  - `x && y` siempre se evalúa "x" si "x" es "true", también se evalúa "y" se denomina "cortocircuito"

- alternativa lógica (en español, O; en inglés, OR): el resultado es "true" si algún operando es "true"
  - $x \mid y$  siempre se evalúan "x" e "y"
  - $x \mid\mid y$  siempre se evalúa "x", si "x" es "false", también se evalúa "y" se denomina "cortocircuito"
- disyunción lógica (en inglés, XOR): el resultado es "true" si los operandos son distintos
  - $x \wedge y$
- negación lógica (en inglés, NOT: el resultado es lo contrario del operando
  - $!x$

Ejemplo. Un año es bisiesto si es múltiplo de 4, excepto si es múltiplo de 100. No obstante, lo múltiplos de 400 son también bisiestos.

```
boolean bisiesto(int año) {
    boolean multiplo4 = año % 4 == 0;
    boolean multiplo100 = año % 100 == 0;
    boolean multiplo400 = año % 400 == 0;
    return (multiplo4 && (! multiplo100)) || multiplo400;
}
```

## 15. Booleanos (concepto)

Ver "[boolean](#)".

## 16. break (palabra reservada)

Se emplea para forzar la terminación de un [bucle](#). Es útil en bucles cuya condición de terminación no se puede chequear cómodamente ni al principio (bucles "[while](#)") ni al final (bucles "[do ... while](#)").

```
for ( ; ; ) {
    String linea = entrada.readLine();
    if (linea == null)
        break;
    // se hace algo con la línea leída
}
```

Aunque es muy práctico, puede dar lugar a programas de difícil lectura.

**Se recomienda NO usar sentencias "break" salvo que sea evidente su necesidad.**

<i>sin break</i>	<i>con break</i>
<pre>String linea =     teclado.readLine(); while (linea != null) {     procesa(linea);     linea = teclado.readLine(); }</pre>	<pre>for ( ; ; ) {     String linea =         teclado.readLine();     if (linea == null)         <b>break;</b>     procesa(linea); }</pre>

<i>sin break</i>	<i>con break</i>
	}

Las sentencias “break” también se emplean en construcciones “[switch](#)”.

Ver “[continue](#)”.

## 17. Bucles [loops] (concepto)

Fragmentos de código que se ejecutan repetidamente.

Java proporciona diferentes tipos de bucles:

- “[while](#)”, que se ejecutan cero o más veces; la condición de terminación se chequea al principio
- “[do ... while](#)”, que se ejecutan una o más veces; la condición de terminación se chequea al final
- “[for](#)”, que se ejecutan N veces
- “[for each](#)”, o iteradores, que se ejecutan sobre todos y cada uno de los términos de un array o colección de datos: listas, conjuntos, ...

Cuando un bucle está lanzado, java proporciona dos formas de forzarlo desde dentro:

- “[break](#)”: provoca la terminación del bucle: lo aborta
- “[continue](#)”: provoca el comienzo de una nueva repetición: aborta esta pasada

### bucles “while”

---

Los bucles “while” se ejecutan cero o más veces.

Ver “[while](#)”.

### bucles “do ... while”

---

Los bucles “do ... while” se ejecutan una o más veces.

Ver “[do](#)”.

### bucles “for”

---

Los bucles “for” se ejecutan un número determinado de veces.

Ver “[for](#)”.

### bucles “n + ½”

---

En algunas ocasiones no es cómodo escribir la condición de terminación ni al principio (“while”) ni al final (“do ... while”) sino que aparece de forma natural por en medio:

```
for ( ; ; ) {
    String linea = consola.readLine();
    if (linea == null)
        break;
    procesa(linea);
}
```

```
while (true) {
    String linea = consola.readLine();
    if (linea == null)
```



```

        break;
        procesa(linea);
    }

String linea = consola.readLine();
while (linea != null) {
    procesa(linea);
    linea = consola.readLine();
}

for (String linea = consola.readLine();
     linea != null;
     linea = consola.readLine()) {
    procesa(linea);
}

```

### errores frecuentes

- Hacer una vez de más o de menos
- Equivocarse en el caso inicial o final
  - hay que probar siempre los casos extremos: condición que no se satisface, sólo se satisface una vez,...
- No comprobar que las variables están correctamente inicializadas
- Poner condiciones exactas: es mejor curarse en salud:
  - while (i != 10) ...
  - while (i < 10) ... // por si acaso
- Poner condiciones exactas trabajando con reales
  - condiciones == o !=
  - es mejor recurrir siempre a <, <=, > o >=

## 18. Bugs (concepto)

Dícese de los errores de los programas que provocan un fallo en ejecución, bien deteniéndolos, bien llevándoles a conclusiones incorrectas.

La palabra viene del inglés, "bicho", y es reminiscencia de aquellos primeros ordenadores en los que anidaban bichos al calor de los circuitos, provocando fallos de la máquina.

## 19. byte (palabra reservada)

<i>byte : números enteros</i>	
<b>ocupación</b>	1 byte = 8 bits
<b>mínimo</b>	Byte.MIN_VALUE = $-2^7$ = -128
<b>máximo</b>	Byte.MAX_VALUE = $2^7 - 1$ = +127

Ver "[Números](#)".

## 20. Bytecode

Se dice del código interpretable, que es ...

- lo que hay en los ficheros .class
- lo que genera el compilador (javac)
- lo que pasamos al intérprete (java)
- lo que ejecuta la máquina virtual (JVM)

Ver "[código](#)".

## 21. Cabecera (de un método) (concepto)

Son las primeras líneas de un [método](#), donde se indicaremos:

- su visibilidad: public, ..., private
- el tipo de valor que devuelve (o void si no devuelve nada)
- el nombre del método
- los argumentos formales: tipo y nombre

```
public String cifra(String texto, byte[] clave) ...
```

## 22. Campo [field] (concepto)

Son contenedores (variables) para los valores internos de las [clases](#) y/o [objetos](#).

- de objeto: cuando cada objeto de una clase tiene su propia copia; es decir, muchos objetos de la misma clase tienen cada uno su conjunto de campos privado
- de clase (**static**): cuando todos los objetos de una clase comparten una copia común

<i>class Circulo</i>
<pre>public class Circulo {     public <b>static</b> final double <b>PI</b> = 3.1416;     private double <b>radio</b>;      public double area() { return PI * r * r; } }</pre>

- "PI" es un campo de clase porque pone "**static**". Todos los objetos de clase Circulo comparten el mismo valor de "PI". Además, es público ("public") e inmutable ("final"); es decir, una constante.
- "radio es un campo de objeto porque no pone "**static**". Cada objeto de clase Circulo tendrá su propio radio. Además, es privado ("private").

Cada campo puede [declararse](#) como:

1. **public**, **protected**, *de paquete* o **private**, según la [visibilidad](#) que se desee
2. **static** si es de clase; si no se dice nada es de objeto
3. **final** si queremos que su valor sea inmodificable; si no se dice nada, podrá modificarse

Se recomienda que

- todos los campos de objeto sean **private**
- todos los campos de objeto queden inicializados en su construcción
- los nombre de los campos de objeto empiecen por una letra minúscula
- los nombres de los campos de clase empiecen por una letra minúscula, salvo que sean constantes (**static final**) en cuyo caso se escribirán con letras mayúsculas

## 23. Caracteres (concepto)

Ver "[char](#)": caracteres individuales.

Ver "[String](#)": cadenas de caracteres.

## 24. case (palabra reservada)

Componente sintáctico de sentencias condicionales de tipo "[switch](#)".

## 25. casting (concepto)

Se dice de la conversión entre tipos de datos.

Con [tipos primitivos](#), se denomina promoción (cuando se pasa de un tipo más pobre a un tipo más rico) y reducción (cuando se pasa de un tipo más rico a un tipo más pobre). Ver "[promoción](#)" y "[reducción](#)".

Con [objetos](#), el objeto en sí no se modifica; pero sí se modifica el tipo de objeto que la variable cree que referencia.

En los párrafos siguientes usaremos las definiciones

```
class A { ... }
class B extends A { ... }
```

### upcasting

Se dice del caso en el que una variable de tipo A recibe el valor de una variable de tipo B.

```
B b = ...;
A a = (A)b;    // explícito; no es necesario
A a = b;      // implícito
```

Esta operación siempre se puede hacer, sin necesidad de indicárselo explícitamente al compilador. Es una operación segura que jamás causa ningún error.

Ver "[sustitución](#)".

### downcasting

Se dice del caso en el que una variable de tipo B recibe el valor de una variable de tipo A.

```
A a = ...;;
B b = (B)a;    // explícito; es necesario
```

Esta operación sólo se puede hacer, si el objeto referenciado por "a" es realmente de tipo B; es decir, sólo es posible cuando el *downcasting* deshace un *upcasting* anterior. De lo contrario se provoca un error en tiempo de ejecución, lanzándose una excepción de tipo `ClassCastException`.

<i>factible</i>	<i>ClassCastException</i>
-----------------	---------------------------

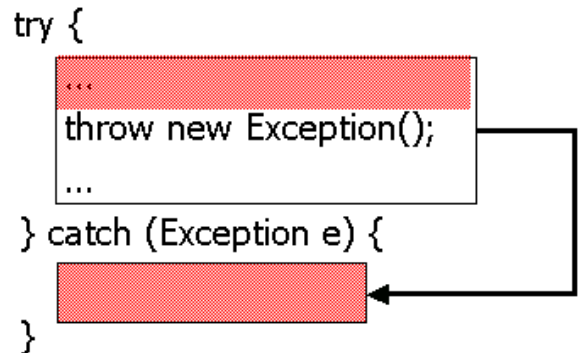
<i><b>factible</b></i>	<i><b>ClassCastException</b></i>
<pre>A a = new B(); B b = (B) a;</pre>	<pre>A a = new A(); B b = (B) a;</pre>

## 26. ***catch*** (palabra reservada)

Componente sintáctico de sentencias ***try***.

Básicamente permite capturar las excepciones lanzadas durante de la ejecución del código dentro del bloque ***try***. De hecho, interrumpe la ejecución secuencial del código y pasa a ejecutarse el bloque ***catch***.

Se pueden acumular varios bloques ***catch*** asociados al mismo bloque ***try***. Se prueban en el mismo orden que aparecen. La primera excepción que casa, es la que dispara el código del bloque asociado. Casar significa que la excepción a capturar es igual a o subclase de la excepción reflejada en la cabecera del ***catch***:



```
class A extends Exception { ... }
class B extends A { ... }
```

```
void metodo() throws A, B { ... }
```

```
try {
    metodo();
} catch (B b) {
    ... captura excepciones de clase B
} catch (A a) {
    ... captura excepciones de clase A
} catch (Exception e) {
    ... captura otras excepciones
}
```

```
try {
    metodo();
} catch (A a) {
    ... captura excepciones de clase A o B
} catch (B b) {
    ... no captura nada
}
```

Si ninguno de los ***catch*** presentes puede capturar la excepción, esta salta fuera del bloque ***try***.

## 27. ***char*** (palabra reservada)

Tipo que representa caracteres simples.

Un carácter es lo que usted se imagina:

```
'a', 'M', '0', '=', '$', ...
```

En el código fuente los caracteres se escriben entre comillas simples.

Algunos caracteres requieren una notación específica para que el código quede muy claro:

<b>caracteres especiales</b>		
'\b'	atrás ( <i>backspace</i> )	se suele usar para borrar el último carácter introducido
'\n'	nueva línea ( <i>new line</i> )	en sistemas como Unix, Linux, etc. se utiliza para separar líneas de texto
'\r'	retorno del carro ( <i>return</i> )	en sistemas como Windows y en Internet, la combinación “\r\n” se utiliza para separar líneas de texto
'\t'	tabulador	salta a la siguiente posición tabulada; normalmente al primer múltiplo de 8
'\\'	escape	el propio carácter “barra inclinada a la izquierda”
'\''	comilla simple	

Habitualmente podrá utilizar sin mayores complicaciones los caracteres del castellano e idiomas occidentales. Para ser más precisos, lo que se conoce como ISO-LATIN-1:

decimal	carácter	hexadecimal
32 - 47	! " # \$ % & ' ( ) * + , - . /	20 - 2F
48 - 63	0 1 2 3 4 5 6 7 8 9 : ; < = > ?	30 - 3F
64 - 79	@ A B C D E F G H I J K L M N O	40 - 4F
80 - 95	P Q R S T U V W X Y Z [ \ ] ^ _	50 - 5F
96 - 111	` a b c d e f g h i j k l m n o	60 - 6F
112 - 126	p q r s t u v w x y z {   } ~	70 - 7E
160 - 175	ı ç £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯	A0 - AF
176 - 191	° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿	B0 - BF
192 - 207	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï	C0 - CF
208 - 223	Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß	D0 - DF
224 - 239	à á â ã ä å æ ç è é ê ë ì í î ï	E0 - EF
240 - 255	ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ	F0 - FF

Un carácter en Java se representa internamente con 16 bits (2 bytes), capacitando a los programas para manejar 65.536 caracteres diferentes. Para codificar los caracteres se utiliza el convenio denominado Unicode, que es un acuerdo internacional para decidir qué combinación de 16 bits representa cada carácter. Con 65.536 posibilidades, realmente caben caracteres de muchos idiomas.

Ver “[Unicode](#)”.

Cuando necesite varios caracteres formando palabras y frases, utilice los “[String](#)”.

## 28. Clases (concepto)

Es el formalismo que permite definir un cierto tipo de [objetos](#). Todo objeto es de alguna clase y todos los objetos se comportan de acuerdo a la definición de su clase.

Las clases son un conjunto de [campos](#) y [métodos](#). Opcionalmente una clase puede definir otras clases; pero esto es menos frecuente.

Entre objetos de una clase, un objeto difiere de otro en el valor de sus [campos](#).

Las clases responden a la pregunta de “¿Cómo es (un objeto)?”

Cada clase tiene un nombre que debe serle propio y no repetirse dentro de su [ámbito](#) para evitar ambigüedades.

Por convenio, los nombres de las clases deben comenzar con una letra mayúscula

## operaciones con clases

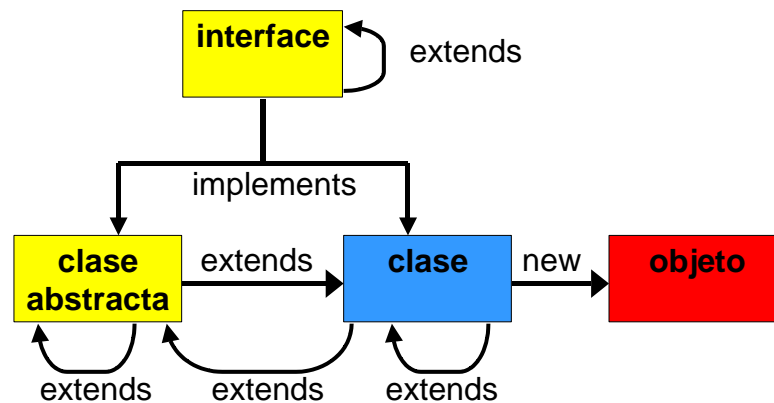
---

Las clases se pueden instanciar (**new**) para dar lugar a objetos.

Las clases se pueden extender (**extends**) para dar lugar a nuevas clases.

Las clases pueden implementar (**implements**) interfaces.

Gráficamente:



## clases abstractas

---

Son clases que indican qué se puede hacer con ellas; pero no indican cómo se hace algo.

Las clases abstractas tienen algún método abstracto: indica nombre, argumentos, resultado y excepciones, pero obvia el cuerpo del método.

### *ejemplo: series numéricas*

```
public abstract class Serie {
    private final int t0;

    protected Serie(int t0) { this.t0 = t0; }

    public int t0() { return t0; }

    public abstract int termino(int n);

    public int suma(int n) {
        int suma = 0;
        for (int i = 0; i < n; i++)
            suma+= termino(i);
        return suma;
    }
}
```

Ver "[abstract](#)".

## 29. **class** (palabra reservada)

Se usa para definir [clases](#).

## 30. **Código** [code] (concepto)

Se llama "código" a las instrucciones que se le dan a un ordenador para funcionar. El código es la representación del programa.

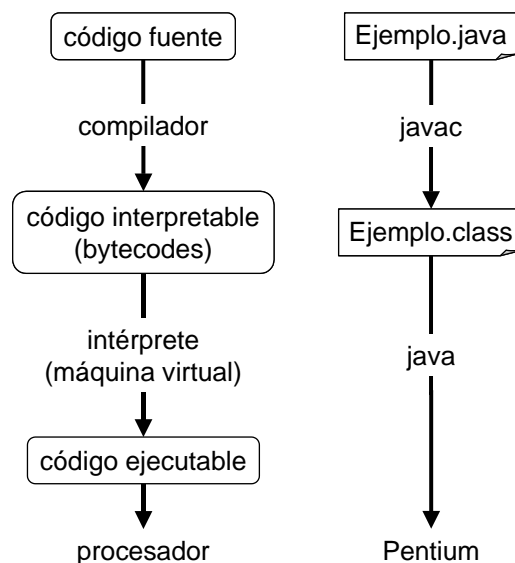
- **Código fuente.** Cuando el código está escrito para que lo lean las personas. El código fuente se puede editar para crearlo, ampliarlo o corregirlo. El código fuente se le pasa al compilador, que genera código interpretable.

El compilador de java produce código interpretable. En otros lenguajes, el compilador genera directamente código ejecutable.

- **Código interpretable.** Cuando el código está pensado para que lo lea el intérprete de java. Este intérprete, también llamado "máquina virtual de java" es un ordenador programado. El código interpretable se le pasa al intérprete, que genera código ejecutable.

En la jerga de java, al código interpretable se le denomina [bytecodes](#).

- **Código ejecutable.** Cuando el código está pensado para máquinas. Por ejemplo, código para ordenadores Pentium.



## 31. **clone** (método) *protected Object clone()*

Este método se declara en la clase Object y está disponible en todas las clases.

Básicamente, se trata de crear otro objeto similar a **this**, con sus campos propios.

Clone() copia los valores de los campos de tipo primitivo, y comparte referencias de los campos que son objetos o arrays. Por ello conviene ser cauto en el uso del método y lo más habitual es

que los objetos que van a utilizarlo lo redefinan, indicando claramente cuándo se comparten referencias y cuándo se hacen copias frescas.

Ver "[clone\(\)](#)".

Ver "[copia de arrays](#)".

## 32. Codificación de caracteres [encoding]

Java trabaja con caracteres usando 16 bits por carácter, según la norma internacional [unicode](#). Hay varios convenios para utilizar bytes (8 bits) para representar caracteres (16 bits). Los más habituales son:

### iso-8859-1

Sólo sirve para caracteres que usan 8 bits (es decir, de los 16 bits del carácter, los 8 primeros son ceros). Para transformar el carácter en un byte, simplemente se ignoran los 8 primeros bits. Para transformar un byte en carácter, se le añaden 8 bits delante.

Esta codificación es suficiente para los lenguajes del suroeste de Europa (lenguas latinas).

### UTF-8

Sirva para codificar cualquier carácter unicode.

El convenio se especifica en la norma [RFC2279](#).

Puede ver una descripción en la [wikipedia](#).

### UTF-16BE

### UTF-16LE

Estos formatos sirven para codificar cualquier carácter unicode. Se limitan a partir los 16 bits del carácter en 2 bytes de 8 bits cada uno. La diferencia entre los dos formatos es el orden en que se ponen los 2 bytes.

El convenio se especifica en la norma [RFC2781](#).

Puede ver una descripción en la [wikipedia](#).

Si no se dice nada, se utiliza el valor por defecto, que puede conocerse por medio del método:

```
Charset Charset.defaultCharset()
```

El conjunto de todas las codificaciones disponibles depende de cada plataforma. Para conocer los que tiene puede usar el método

```
SortedMap<String, Charset> Charset.availableCharsets()
```

## Ejemplos

---

Este pequeño programa le permitirá analizar cómo se transforma una String en bytes según una cierta codificación:

```
// private static final String LETRAS = "aeiou áéíóú";
private static final String LETRAS =
    "aeiou \u00E1\u00E9\u00ED\u00F3\u00FA";

public static void main(String[] args)
    throws UnsupportedOperationException {
    encode("ISO-8859-1"); // ISO Latin-1
    encode("Cp1252");    // Windows Latin-1
    encode("MacRoman");   // Macintosh Roman
    encode("UTF-8");
```



```

        encode("UTF-16BE");
        encode("UTF-16LE");
    }

    private static void encode(String name)
        throws UnsupportedEncodingException {
        byte[] bytes = LETRAS.getBytes(name);
        System.out.printf("%-12s:%d:", name, bytes.length);
        for (byte b : bytes)
            System.out.printf(" %02x", b);
        System.out.println();
    }

```

Resultado de ejecución:

```

carácter:      a  e  i  o  u      á  é  í  ó  ú
ISO-8859-1:11: 61 65 69 6f 75 20 e1 e9 ed f3 fa
Cp1252      :11: 61 65 69 6f 75 20 e1 e9 ed f3 fa
MacRoman    :11: 61 65 69 6f 75 20 87 8e 92 97 9c
UTF-8       :16: 61 65 69 6f 75 20 c3 a1 c3 a9 c3 ad c3 b3 c3 ba
UTF-16BE    :22:
    00 61 00 65 00 69 00 6f 00 75 00 20 00 e1 00 e9 00 ed 00 f3 00 fa
UTF-16LE    :22:
    61 00 65 00 69 00 6f 00 75 00 20 00 e1 00 e9 00 ed 00 f3 00 fa 00

```

### Utilización en ficheros (lectura y escritura)

---

Al leer y escribir ficheros java, podemos especificar el tipo de codificación en el fichero.

A continuación se proporcionan una serie de patrones típicos:

#### Para leer:

```

InputStream is= ...
Reader reader1= new InputStreamReader(is, "iso-8859-1");
Reader reader2= new InputStreamReader(is, "UTF-8");

```

#### Para escribir:

```

String filename = ...
OutputStream os= new FileOutputStream(filename);

Writer writer1= new OutputStreamWriter(os, "iso-8859-1");
Writer writer2= new PrintWriter(filename, "iso-8859-1");

Writer writer21= new OutputStreamWriter(os, "UTF-8");
Writer writer22= new PrintWriter(filename, "UTF-8");

```

Si no se dice nada, se utiliza el valor por defecto, que puede conocerse por medio del método:

```

static Charset Charset.defaultCharset()

```

### 33. Colas [queues] (estructura de datos)

Son listas de objetos que se caracterizan porque los nuevos objetos se añaden al final, mientras que van saliendo por el principio. De esta forma, resulta que el primero que entra es el primero que sale (en Inglés, FIFO: First In, First Out).

<i>Colas (listas FIFO)</i>
<pre>public interface Cola&lt;T&gt; {     // mete un objeto T al final de la cola     void mete(T t) throws ExcepcionColaLlena;      // retira el primer objeto de la cola     T sacaPrimero() throws ExcepcionColaVacía;      // mira, sin retirar, el primer objeto     T miraPrimero() throws ExcepcionColaVacía;      // objetos en la cola     int longitud(); }</pre>

Es fácil implementar las colas como listas:

<i>Cola implementada con una lista</i>
<pre>import java.util.*;  public class ColaLista&lt;T&gt; implements Cola&lt;T&gt; {     private List&lt;T&gt; lista = new ArrayList&lt;T&gt;();      public void mete(T t) {         lista.add(t);     }      public T miraPrimero() throws ExcepcionColaVacía {         if (lista.size() == 0) throw new ExcepcionColaVacía();         return lista.get(0);     }      public T sacaPrimero() throws ExcepcionColaVacía {         if (lista.size() == 0) throw new ExcepcionColaVacía();         return lista.remove(0);     }      public int longitud() {         return lista.size();     } }</pre>

### 34. Comentarios [comments] (concepto)

Texto que se escribe entremezclado con el código fuente pero que, a diferencia de éste, no está destinado a la máquina, sino a las personas que lean el código fuente. El objetivo de los

comentarios es facilitar a las personas la comprensión del código, así como aportar información adicional tal como el nombre del autor, fechas relevantes, propiedad intelectual, etc.

En java hay

- comentarios de una línea que comienzan con los caracteres "//" y terminan con la línea

```
// Comentario en una sola línea.
```

- comentarios de un bloque de líneas que comienzan con los caracteres "/\*" y terminan con los caracteres "\*/"

```
/*  
 * Comentario que ocupa varias líneas  
 * de forma que el autor se puede explayar.  
*/
```

- ✖ estos comentarios no se pueden anidar: comentarios dentro de comentarios

- comentarios para generación automática de documentación que comienzan con los caracteres "/\*" y terminan con los caracteres "\*/". En estos comentarios, algunas líneas tienen una interpretación especial a fin de generar la documentación pertinente.

```
/**  
 * Ejemplo: círculos.  
 * @author José A. Mañas <jmanas@dit.upm.es>  
 * @version 24.9.2004  
*/
```

estos comentarios no se pueden anidar: (es decir, no pueden haber comentarios dentro de otros comentarios)

véase "[documentación](#)"

### 35. *compareTo* (método)

Método que deben definir los objetos que implemental la interface Comparable<T>. Esta interface define clases que disfrutan de una relación de orden total, lo que se traduce en que las clases que implementen esta interface deben proporcionar un método

```
int compareTo(T x)
```

que compara THIS con x, devolviendo

- un número negativo si this < x
- cero si this = x
- un número positivo si this > x

Por medio de compareTo() establecemos una relación de orden total en los objetos de una clase. Recuerde que las relaciones de orden total, <=, tienen las siguientes propiedades:

- reflexividad  
(x <= x)
- antisimetría  
(x <= y) and (y <= x) => x = y
- transitividad  
(x <= y) and (y <= z) => (x <= z)

- total: todos los elementos están relacionados  
para todo x, y: (x <= y) or (y <= x)

Ver "[interface Comparable<T>](#)".

Ver "[equals\(\) y compareTo\(\)](#)".

### 36. **Compilación [compilation] (concepto)**

Paso de código fuente a código interpretable.

El compilador de java produce código interpretable. En otros lenguajes, el compilador genera directamente código ejecutable.

En el caso de java, se pasa de un fichero .JAVA a un fichero .CLASS

Ver "[código](#)".

Ver herramienta "[javac](#)".

### 37. **Command Line Interface (CLI)**

Ver "[Consola](#)".

### 38. **Composición [composition] (concepto)**

Se dice que se usa el principio de composición cuando para construir una nueva clase empleamos otras clases ya definidas.

```
class B { private A a = ...; }
```

se dice que B usa la clase A.

Ver "[delegación](#)".

#### **¿cuándo usar composición?**

---

- Cuando se necesita la funcionalidad de otra clase dentro de esta; pero no su interfaz (por eso la otra clase aparece como **private**)
- Cuando hay más de 1 objeto que incorporar

### 39. **Conjuntos [set] (estructuras de datos)**

Estructuras de datos donde no importa el orden, ni se admiten duplicados.

Ver "[Set](#)".

### 40. **Consola**

En Windows se denomina "Símbolo del sistema"; en los sistemas de tipos UNIX, se denominan "shell". El caso es que el usuario depende de una pantalla donde va dando órdenes textuales.

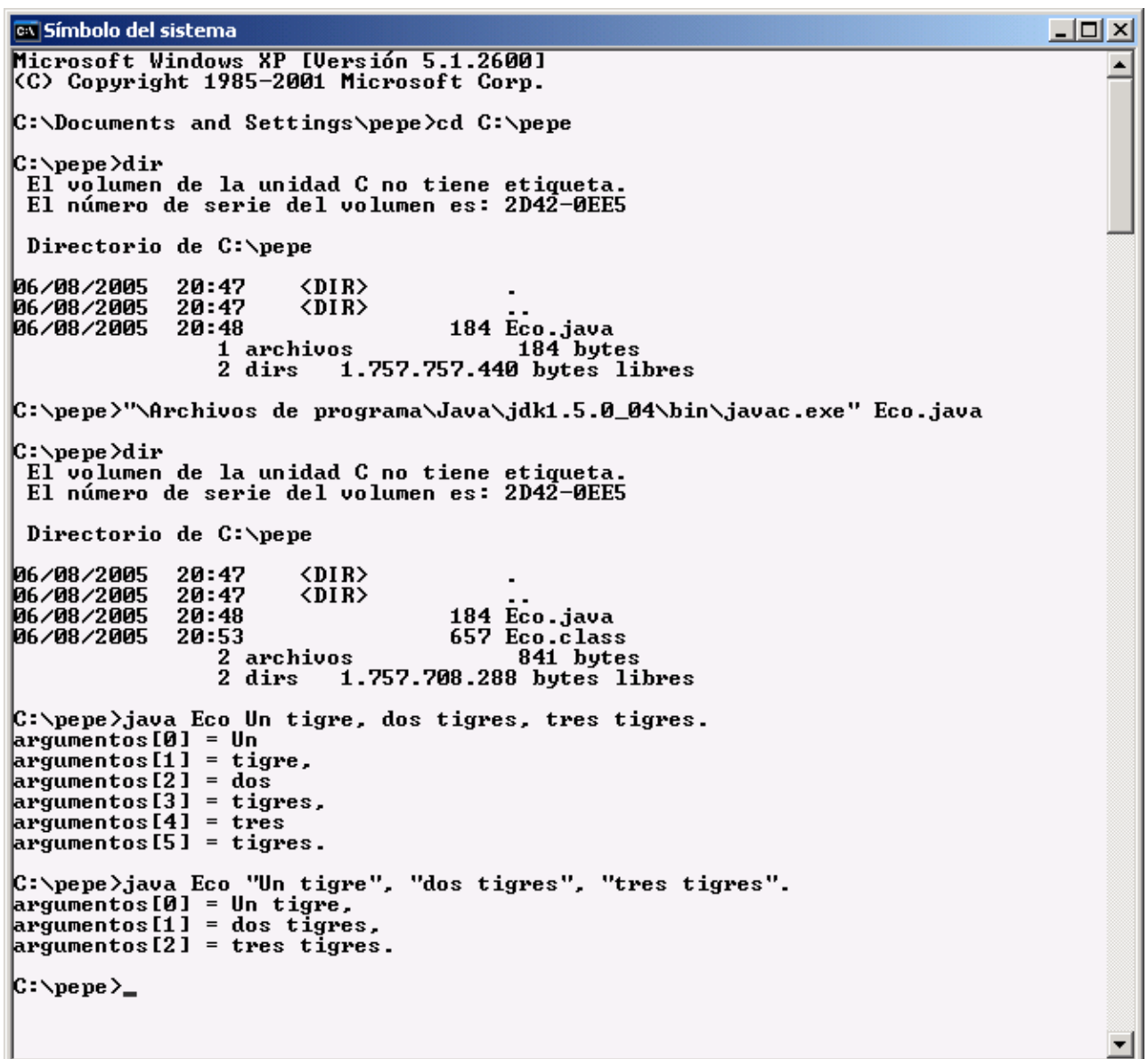
Veamos un ejemplo:

Paso a paso.

- vamos al directorio C:\pepe
- vemos que en el directorio sólo hay un fichero Eco.java ([código fuente](#))
- [compilamos](#) el fichero usando "javac"
- vemos que en el directorio aparece un fichero Eco.class ([código interpretable](#))
- [ejecutamos](#) "Eco.class" usando "java"
- vemos que aparecen los argumentos de la línea de "comandos", entendiendo que los argumentos son texto separado por espacio en blanco, salvo que usemos comillas para agrupar varias palabras en un solo argumento.

### *Eco.java*

```
public class Eco {  
    public static void main(String[] argumentos) {  
        for (int i = 0; i < argumentos.length; i++)  
            System.out.println("argumentos[" + i + "] = " + argumentos[i]);  
    }  
}
```



```
C:\> Símbolo del sistema  
Microsoft Windows XP [Versión 5.1.2600]  
<C> Copyright 1985-2001 Microsoft Corp.  
  
C:\Documents and Settings\pepe>cd C:\pepe  
  
C:\pepe>dir  
El volumen de la unidad C no tiene etiqueta.  
El número de serie del volumen es: 2D42-0EE5  
  
Directorio de C:\pepe  
06/08/2005  20:47    <DIR>          .  
06/08/2005  20:47    <DIR>          ..  
06/08/2005  20:48                184 Eco.java  
                1 archivos            184 bytes  
                2 dirs    1.757.757.440 bytes libres  
  
C:\pepe>"\Archivos de programa\Java\jdk1.5.0_04\bin\javac.exe" Eco.java  
  
C:\pepe>dir  
El volumen de la unidad C no tiene etiqueta.  
El número de serie del volumen es: 2D42-0EE5  
  
Directorio de C:\pepe  
06/08/2005  20:47    <DIR>          .  
06/08/2005  20:47    <DIR>          ..  
06/08/2005  20:48                184 Eco.java  
06/08/2005  20:53                657 Eco.class  
                2 archivos            841 bytes  
                2 dirs    1.757.708.288 bytes libres  
  
C:\pepe>java Eco Un tigre, dos tigres, tres tigres.  
argumentos[0] = Un  
argumentos[1] = tigre,  
argumentos[2] = dos  
argumentos[3] = tigres,  
argumentos[4] = tres  
argumentos[5] = tigres.  
  
C:\pepe>java Eco "Un tigre", "dos tigres", "tres tigres".  
argumentos[0] = Un tigre,  
argumentos[1] = dos tigres,  
argumentos[2] = tres tigres.  
  
C:\pepe>_
```

## 41. Constantes [constant] (concepto)

Se dice de los campos que se declaran “static final”.

Por convenio, se suelen escribir usando solamente letras mayúsculas y dígitos. Si constan de varias palabras, se separan con caracteres '\_’.

Ejemplos:

```
public static final double PI = 3.1416;
public static final int DESCUBRIMIENTO_AMERICA = 1492;
```

Ver "[variables invariantes](#)".

Ver "[identificadores](#)".

## 42. Constructores [constructors] (concepto)

Son métodos de una clase que sirven para crear objetos. Tiene una notación especial para diferenciarlos de los métodos normales:

- no devuelven nada, ni dicen "void"
- su nombre es idéntico al de la clase

Si no se escribe ningún constructor, java se inventa uno que no tiene ningún argumento e inicializa todos los campos a "cero". Java sólo inventa constructores si el programador no escribe ninguno. En cuanto se escribe uno, java se limita a lo escrito.

Ejemplo:

<i>class Punto</i>
<pre>class Punto {     private double x, y;     double getX() { return x; }     double getY() { return y; }</pre>
<i>uso</i>
<pre>// creamos un objeto usando el constructor por defecto Punto p= new Punto(); // vemos sus campos: p.getX()    // devuelve 0.0 p.getY()    // devuelve 0.0</pre>

**Se recomienda que todos los constructores inicialicen todos los campos del objeto.**

Ver "[métodos constructores](#)".

Ver "[fábricas](#)".

## 43. continue (palabra reservada)

Se emplea para forzar la terminación de una pasada en un [bucle](#). Es útil cuando queremos abortar limpiamente una ejecución del bucle; pero que éste siga ejecutándose:

```
for ( ; ; ) {
    String linea = entrada.readLine();
    if (linea.charAt(0) == '#')           // se ignoran las líneas que
        continue;                       // empiezan por '#'
    // se hace algo con la línea leída
}
```

Aunque es muy práctico, puede dar lugar a programas de difícil lectura.

**Se recomienda NO usar sentencias "continue" salvo que sea evidente su necesidad o mejore la legibilidad del código.**

<i>sin continue</i>	<i>con continue</i>
<pre>for (int i = 0; i &lt; 1000; i++) {     A;     if (c1) {         B;         if (c2) {             C;         }     } }</pre>	<pre>for (int i = 0; i &lt; 1000; i++) {     A;     if (! c1) continue;     B;     if (! c2) continue;     C; }</pre>

Ver "[break](#)".

#### **44. Cortocircuito [short-circuit] (concepto)**

Se dice del caso en el que para evaluar una expresión [booleana](#) sólo se evalúa uno de los operandos, el primero.

**x && y**

evalúa "x"; si es falso, devuelve FALSE; si no, evalúa "y"

Se basa en las reglas del álgebra de Boole que dicen que

$$\begin{aligned}\text{true} \wedge y &= y \\ \text{false} \wedge y &= \text{false}\end{aligned}$$

**x || y**

evalúa "x"; si es cierto, devuelve TRUE; si no, evalúa "y"

Se basa en las reglas del álgebra de Boole que dicen que

$$\begin{aligned}\text{false} \vee y &= y \\ \text{true} \vee y &= \text{true}\end{aligned}$$

Estos cortocircuitos sirven para acelerar la ejecución, evitando cálculos innecesarios o para evitar errores de ejecución proporcionando una escapatoria para casos particulares. Algunos ejemplos habituales:

```
if (Math.abs(x) < 0.01 || Math.sin(x) < 0.01) ...  
if (nombre == null || nombre.length() == 0) ...  
if (conjunto != null && conjunto.contains(elemento)) ...
```

NOTA. Si se quiere evaluar siempre el segundo operando, use los operadores

$$\begin{aligned}x \ \& \ y \\ x \ | \ y\end{aligned}$$

respectivamente.

## 45. Cuerpo (de un método) [method body] (concepto)

El fragmento de código fuente que indica qué hacer cuando se llama al [método](#); es decir, cómo se genera el resultado.

Sintácticamente, en java, lo que aparece entre llaves.

```
public class Punto {  
    private double x, y;  
  
    public double distancia(Punto q) {  
        double dx = q.getX() - x;  
        double dy = q.getY() - y;  
        return Math.sqrt(dx * dx + dy * dy);  
    }  
}
```

## 46. Debug (concepto)

Perseguir la ejecución de un programa detectando errores (bugs) y reparándolos.

Ver "[bugs](#)".

## 47. Declaración

Se llama declarar al acto por el que se anuncia la existencia de una [clase](#), un [método](#) o una [variable](#).

No se puede usar nada que no esté declarado en alguna parte.

La declaración debe dar suficientes indicaciones de cómo se puede usar lo declarado.

No pueden haber declaraciones ambiguas, ni dos cosas declaradas de tal forma que su uso pueda ser ambiguo.

Las declaraciones tienen un cierto [ámbito](#) o zona de [visibilidad](#).

### Declaración de variables

---

Las variables tienen un nombre que debe ser único en [ámbito](#) en que se declaran y se usan.

La declaración debe indicar el tipo de valores que puede contener, bien un [tipo primitivo](#), bien algún tipo o [clase](#) del programa.

<ámbito> [ static ] [ final ] <tipo> identificador [ = <valor\_inicial> ] ;

entre paréntesis aparece lo que es opcional

<[ámbito](#)> puede ser **public**, de paquete, **protected** o **private**

[static](#) aparece en campos, cuando son de clase

[final](#) aparece cuando a la variable se le asigna un valor inicial que no podrá ser alterado en el futuro

### Declaración de variables dentro de una clase: campos

---

Son variables declaradas dentro de una [clase](#), pero fuera de cualquier [método](#).

La declaración responde al esquema general:

<ámbito> [ static ] [ final ] <tipo> identificador [ = <valor\_inicial> ] ;



Su ámbito de visibilidad es al menos toda la clase en la que se declaran. Los modificadores de visibilidad pueden hacerlas visibles fuera de la clase.

Las variables de clase, [static](#), existen desde que arranca el programa hasta que termina.

Las variables de objeto existen desde que se crea el objeto (con [new](#)), hasta que el objeto deja de usarse.

### **Declaración de variables locales dentro de un método**

Son variables declaradas dentro del cuerpo de un [método](#).

A veces de llaman "variables automáticas".

Responden al esquema:

```
[ final ] <tipo> identificador [ = <valor_inicial> ] ;
```

Su [ámbito](#) de uso abarca desde que se declaran hasta que se cierra el método.

La variable se crea cuando la ejecución pasa por su declaración. La variable se destruye cuando el método termina.

### **Declaración de variables dentro de un bloque**

Son variables declaradas dentro de un [bloque de código](#): conjunto de sentencias encerradas entre llaves.

```
{ sentencias }
```

Responden al esquema:

```
[ final ] <tipo> identificador [ = <valor_inicial> ] ;
```

Su [ámbito](#) de uso abarca desde que se declaran hasta que se cierra el bloque.

La variable se crea cuando la ejecución pasa por su declaración. La variable se destruye cuando el método termina.

### **Declaración de argumentos de un método**

Son variables declaradas en la cabecera de un [método](#), entre paréntesis.

Responden al esquema:

```
[ final ] <tipo> identificador ;
```

Su [ámbito](#) de uso abarca todo el método.

### **Declaración de métodos**

Los métodos se declaran dentro de las [clases](#).

Responden al esquema:

```
<ámbito> [ static ] [ final ] <tipo> identificador ( [ <argumentos> ] )
```

Donde cada <argumento> responde al esquema de declaración de un argumento formal.

Su [ámbito](#) de uso es, al menos, toda la clase en la que se declaran. Los modificadores de visibilidad pueden hacerlos visibles fuera de la clase.

Los métodos [final](#) no pueden [redefinirse](#).

Los métodos de clase, [static](#), existen desde que arranca el programa hasta que termina.

Los métodos de objeto existen desde que se crea el objeto (con [new](#)), hasta que el objeto deja de usarse.

Ver "[signatura de un método](#)".

Ver "[métodos](#)".

## Declaración de clases

---

Las clases se declaran en ficheros. Los ficheros pueden estar en directorios diferentes, lo que corresponde en java a [paquetes](#) diferentes.

La declaración de clases responde al esquema:

<ámbito> [ final ] identificador

Su [ámbito](#) de visibilidad es al menos todo el paquete en el que se declaran. Los modificadores de visibilidad pueden hacerlos visibles fuera del paquete.

Las clases [final](#) no pueden [redefinirse](#).

Las clases existen desde que arranca el programa hasta que termina.

## 48. Delegación [delegation] (concepto)

Cuando un [método](#) de una clase B recurre a métodos de otra clase A para lograr sus objetivos, se dice que B delega en A.

<i>double distancia(Punto p, Punto q)</i>
<pre>double distancia(Punto p, Punto q) {     double dx = p.getX() - q.getX();     double dy = p.getY() - q.getY();     return Math.sqrt(dx*dx + dy*dy); }</pre>

"distancia" delega en el método "sqrt" de la clase "Math" para calcular la raíz cuadrada y en los métodos getX() y getY() de la clase Punto para acceder a las coordenadas.

La delegación es interesante en cuanto permite construir programas cada vez más complejos aprovechando métodos ya disponibles.

Lo contrario de la delegación es la "reinención de la rueda".

Ver "[composición](#)".

## 49. Desbordamiento [overflow] (concepto)

En general, se dice de las situaciones en que se intentan meter más datos en un contenedor de los que caben. Así es muy frecuente oír hablar de "*buffer overflow*" para indicar que no caben tantos datos en un *buffer*.

En aritmética se aplica cuando un valor excede el máximo previsto.

En números enteros el desbordamiento se puede producir por números excesivamente grandes, sean positivos o negativos. Java no lo detecta:

```
Integer.MAX_VALUE + 1 =  
2147483647 + 1 = -2147483648  
  
Integer.MIN_VALUE - 1 =  
-2147483648 - 1 = 2147483647  
  
1000000000 * 10 = 1410065408
```

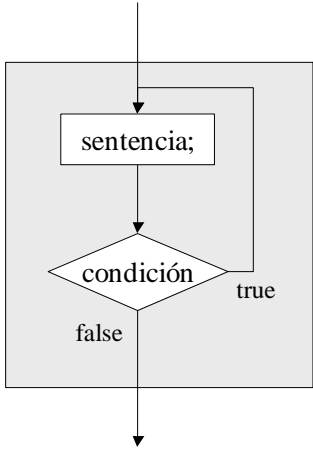
En números en coma flotante el desbordamiento se puede producir en el exponente cuando éste es positivo. Java lo interpreta como que nos hemos ido al infinito.

```
Double.MAX_VALUE * 10 =
1.79769e+308 * 10 = Infinity
```

Ver "[underflow](#)".

## 50. **do ... while (palabra reservada)**

Bucles que se ejecutan una o más veces. La condición de terminación se chequea al final.

<i>java</i>	<i>flujo</i>
<pre>do {     sentencia; } while (condición);  do {     sentencia 1;     sentencia 2;     ...     sentencia ...; } while (condición);</pre>	

```
double nota;
do {
    nota = examen();
} while (nota < 5.0);
```

Ver "[bucles](#)".

## 51. **Documentación [documentation] (concepto)**

En general se refiere a todo aquel material que complementa al código ayudando a su comprensión por las personas. Es básico para poder responsabilizarse de un programa pues permite entenderlo para reparar defectos o mejorarlo.

En particular se refiere a una forma especial de comentarios java, entremezclados con el código, y que se caracterizan por

- comenzar con la cadena `/**`
- terminar con la cadena `*/`

Estos comentarios no pueden anidarse, ni entre sí, ni con comentarios de tipo `/* ... */`. El compilador se limita a detectar la terminación, sin preocuparse de si hay varios comentarios empezados.

- incluir algunas claves `@etiqueta` que se comentan más abajo.

<b>comentarios de documentación</b>
<pre>/**  * Parte descriptiva.  * Que puede consistir de varias frases o párrafos.  * Se puede escribir con formato HTML.  *  * @etiqueta texto específico de la etiqueta  */</pre>

La forma estructurada de estos comentarios permiten un tratamiento especial

- los entornos de desarrollo resaltan en colores su empleo
- herramientas como [javadoc](#) permiten generar páginas HTML (para navegadores)

Estos comentarios se aplican a clases, campos y métodos.

<b>etiquetas habituales</b>
<b>clases</b>
<b>@author</b> nombre y/o filiación del autor
<b>@version</b> código y/o fecha
<b>@see</b> <i>clase   campo   método</i>
<b>campos</b>
<b>@see</b> <i>clase   campo   método</i>
<b>métodos</b>
<b>@param</b> <i>identificador</i> explicación del argumento
<b>@return</b> explicación del resultado
<b>@exception</b> <i>identificador</i> explicación de cuándo se lanza <b>@throws</b> <i>identificador</i> explicación de cuándo se lanza  El <i>identificador</i> identifica la clase de excepción que se puede lanzar.
<b>@deprecated</b>  Indica que, aunque el método existe, se recomienda no utilizarlo, típicamente porque hay otro método alternativo.
<b>@see</b> <i>clase   campo   método</i>

<b>ejemplo de documentación de clase</b>

```
/**
 * Ejemplo: círculos.
 *
 * @author José A. Mañas
 * @version 24.9.2008
 */
public class Circulo {
```

#### ***ejemplo de documentación de método***

```
/**
 * Intersección de dos rectas.
 *
 * @param recta1 una recta.
 * @param recta2 otra recta.
 * @return punto en el que las rectas se cruzan.
 * @exception Paralelas si las rectas son paralelas o coincidentes.
 * @see Punto
 */
public Punto interseccion(Recta recta1, Recta recta2)
    throws Paralelas {
    ...
}
```

## **52. double (palabra reservada)**

<b><i>double : números reales (coma flotante)</i></b>	
<b><i>ocupación</i></b>	8 bytes = 64 bits
<b><i>máximo (absoluto)</i></b>	$\text{Double.MAX\_VALUE} = (2 \cdot 2^{-52}) \cdot 2^{1023} = 1.8 \cdot 10^{308}$
<b><i>mínimo (distinto de cero)</i></b>	$\text{Double.MIN\_VALUE} = 2^{-1074} = 4.9 \cdot 10^{-324}$

Además se definen algunas constantes útiles

**Double.NaN**

Not a Number.

Se usa para representar errores. Por ejemplo, el resultado de dividir por 0.0.

**Double.NEGATIVE\_INFINITY**

**Double.POSITIVE\_INFINITY**

Infinito positivo y negativo.

Ver "[Números](#)".

### 53. Downcasting (concepto)

Se dice cuando el contenido de una variable de tipo A se asigna a una variable de tipo B, siendo B [subclase](#) de A.

```
class B extends A { ... }  
A a = ...;  
B b = (B) a;
```

No siempre es posible. Sólo funciona si “a” contiene un objeto de clase B. De lo contrario se lanza una `ClassCastException`.

Ver “[casting](#)”.

### 54. Edición [edition] (concepto)

La actividad en la que el programador crea o modifica el código fuente.

Se suele utilizar algún entorno integrado de desarrollo (IDE – Integrated Development Environment), aunque en realidad vale cualquier editor de texto.

### 55. Ejecución [execution] (concepto)

La actividad en la que se ejecutan las instrucciones del programa.

En el caso de java, se suele emplear un intérprete que trabaja sobre el código interpretable.

#### ficheros .class

---

Sea la clase

<i>Hola.java</i>
<pre>public class Hola {     public static void main(String[] argumentos) {         System.out.println("Hola.");     } }</pre>

Si el fichero “Hola.class” está en el directorio X, nos situaremos en dicho directorio y llamaremos al intérprete:

```
$ java Hola  
Hola.
```

Alternativamente, si estuviéramos en otro directorio, le indicáramos al intérprete dónde está nuestro código interpretable (el directorio X):

```
$ java -cp X Hola  
Hola.
```

#### con paquetes (o directorios)

---

Si empleamos paquetes, es muy conveniente alinear el nombre del paquete y los directorios. Sea

<i>Hola2.java</i>
<pre>package ejemplo.ejecucion;  public class Hola2 {     public static void main(String[] argumentos) {</pre>

<b>Hola2.java</b>
<pre> System.out.println("Hola.");     } }</pre>

Una vez colocado el fichero “Hola2.class” en su directorio

```
X/ejemplo/ejecucion/Hola2.class
```

nos situaremos en el directorio X y llamaremos al intérprete

```
$ java ejemplo.ejecucion.Hola2
Hola.
```

Alternativamente, si estuviéramos en otro directorio, le indicáramos al intérprete dónde está nuestro código interpretable (el directorio X):

```
$ java -cp X ejemplo.ejecucion.Hola2
Hola.
```

### ficheros .jar

Los ficheros *.class* pueden empaquetarse en ficheros [.jar](#). Esto es especialmente útil cuando hay muchos ficheros *.class* que, empaquetados en un solo fichero *.jar*, son más fáciles de manejar.

```
$ jar tvf Hola2.jar
  0 Thu Aug 18 10:46:34 CEST 2005 META-INF/
122 Thu Aug 18 10:46:34 CEST 2005 META-INF/MANIFEST.MF
427 Thu Aug 18 10:35:04 CEST 2005 ejemplo/ejecucion/Hola2.class
```

Para ejecutar métodos “*main*” en clases empaquetadas en el *jar* basta incorporar el fichero JAR al “*classpath*” e indicar la clase

```
$ java -cp Hola2.jar ejemplo.ejecucion.Hola2
Hola.
$ java -cp X/Hola2.jar ejemplo.ejecucion.Hola2
Hola.
```

### ficheros .jar con manifest

Los ficheros *.class* pueden empaquetarse en ficheros [.jar](#). Esto es especialmente útil cuando hay muchos ficheros *.class* que, empaquetados en un solo fichero *.jar*, son más fáciles de manejar.

Los ficheros *.jar* pueden contener un MANIFEST

```
$ jar tvf Hola2.jar
  0 Thu Aug 18 10:46:34 CEST 2005 META-INF/
122 Thu Aug 18 10:46:34 CEST 2005 META-INF/MANIFEST.MF
427 Thu Aug 18 10:35:04 CEST 2005 ejemplo/ejecucion/Hola2.class
```

Dicho MANIFEST puede indicar el objeto que dispone del método “*main*” para arrancar la ejecución

```

Manifest-Version: 1.0
Class-Path:
Main-Class: ejemplo.ejecucion.Hola2
```

De esta forma el intérprete, java, sabe cómo lanzar la aplicación:

```
$ java -jar Hola2.jar
```

```
Hola.
$ java -jar X/Hola2.jar
Hola.
```

Si su sistema operativo lo permite, pueden asociarse los ficheros con extensión *.jar* al intérprete de java, de forma que baste hacer "doble clic" en el fichero para que se ejecute.

## 56. Ejecución condicional [conditional execution] (concepto)

Podemos hacer que una serie de sentencias se ejecuten o no dependiendo de una condición (booleana).

Ver "[if](#)". Ver "[switch](#)".

## 57. Elección dinámica de método (concepto)

En inglés recibe varios nombres:

- dynamic method lookup
- dynamic method binding
- dynamic method dispatch

En todo caso se refiere al hecho de que cuando una variable es [polimórfica](#) y puede referirse a objetos de varias [subclases](#), el método que se ejecuta depende del tipo del objeto en tiempo de ejecución. Es decir, no depende del tipo de la variable, sino del tipo del objeto.

Se ve más fácilmente con un ejemplo.

- Sean dos clases, A y B
- sea B una extensión a A
- A define un método
- B [redefine](#) el mismo método
- Cuando una variable de tipo A se refiere a un objeto de tipo B, las llamadas al método que existe en ambas clases, elige el método de B

<b>class A</b>	
<pre>public class A {     public String getMe() { return "Soy A"; } }</pre>	
<b>class B extends A</b>	
<pre>public class A {     @Override     public String getMe() { return "Soy A"; } }</pre>	
<b>ejecución:</b>	
<pre>A a = new A(); B b = new B(); A ab= new B(); System.out.println(a.getMe()); System.out.println(b.getMe()); System.out.println(ab.getMe());</pre>	<pre>// upcasting Soy A Soy B Soy B    // sabe que es B</pre>



## 58. *else* (palabra reservada)

Componente sintáctico de sentencias condicionales de tipo "[if](#)".

## 59. Encapsulación [encapsulation] (concepto)

Criterio de diseño de clases que recomienda juntar en una misma clase las variables y los métodos que las gestionan.

Una buena encapsulación debe llevar a que todos los campos de los objetos sean **private** y sólo se puedan acceder o alterar a través de los métodos de la clase. La ventaja que se obtiene es un absoluto control sobre el acceso a dichos campos.

Un caso extremo de encapsulación es el denominado "Tipo Abstracto de Datos" (TAD). Se dice que una clase es un TAD cuando se puede cambiar completamente la representación interna del estado del objeto sin que otras clases que la usan lo perciban; es decir, cuando los métodos ocultan completamente la representación interna. La ventaja que se obtiene es la posibilidad de cambiar el código de la clase sin tocar otras clases del programa; por ejemplo, para optimizar el tiempo de ejecución.

En ingles se dice "ADT – Abstract Data Types".

## 60. *enum* (palabra reservada)

Sirve para declarar tipos [enumerados](#), bien en línea:

```
enum Vocal { A, E, I, O, U };
```

bien como clase en su fichero propio

<i>Vocal.java</i>
<pre>public enum Vocal {     A, E, I, O, U; }</pre>

En ambos casos disponemos de una nueva clase que se caracteriza por disponer de una serie finita y explícita de constantes. Una vez definidos los elementos de un tipo enumerado es imposible crear nuevos objetos.

Otros ejemplos:

```
enum Palo { OROS, ESPADAS, COPAS, BASTOS };  
enum Operacion { SUMA, RESTA, MULTIPLICACION, DIVISION };
```

Ver "[enumerados](#)".

## 61. Enumerados (concepto)

Son clases con un conjunto explícito, finito y fijo de constantes.

Ver "[enum](#)".

Se pueden emplear variables de los tipos enumerados definidos

```
Vocal vocal = Vocal.E;
```

---

### métodos estándar

Los objetos de tipos enumerados disfrutan de una serie de métodos estándar:

<b>código java</b>	<b>valor</b>
vocal	E
vocal.compareTo(Vocal.A)	1        // algo > 0
vocal.compareTo(Vocal.E)	0        // 0
vocal.compareTo(Vocal.U)	-3        // algo < 0
vocal.equals(Vocal.A)	false
vocal.equals(Vocal.E)	true
vocal.name()	"E"      // identificador de la constante
vocal.toString()	"E"      // puede reescribirse
vocal.ordinal()	1

Además, la clase “Vocal” proporciona un par de métodos útiles:

static Vocal valueOf(String)	que dado el identificador de una constante, devuelve el objeto correspondiente
static Vocal[] values()	devuelve un array con los valores constantes, array que puede recorrerse o iterar sobre él

### **campos y métodos**

Como un enumerado es una clase, puede disfrutar de campos y métodos propios, sin olvidar nunca que sólo existirán tantos objetos de esa clase como constantes se han definido.

<b><i>Billetes.java</i></b>
<pre> /**  * Billetes de euros.  */ public enum Billetes {     B5(5), B10(10), B20(20), B50(50), B100(100), B200(200), B500(500);      private final int valor;      Billetes(int valor) {         this.valor = valor;     }      public int getValor() {         return valor;     }      public String toString() {         return valor + "euros";     } } </pre>

También es posible definir métodos específicos para cada objeto constante:

### ***Operation.java***

```
public enum Operation {
    PLUS    { double eval(double x, double y) { return x + y; } },
    MINUS   { double eval(double x, double y) { return x - y; } },
    TIMES   { double eval(double x, double y) { return x * y; } },
    DIVIDE  { double eval(double x, double y) { return x / y; } };

    abstract double eval(double x, double y);

    public static void main(String[] argumentos) {
        double x = Double.parseDouble(argumentos[0]);
        Operation operation = Operation.valueOf(argumentos[1]);
        double y = Double.parseDouble(argumentos[2]);
        System.out.println(operation.eval(x, y));
    }
}
```

```
$ java Operation 2 PLUS 2
4.0
```

### **iteración**

La forma más directa de iterar sobre los miembros de un tipo enumerado es usar el array de constantes

```
for (Iterator vocal: Vocal.values()) { ... }
```

Ver “[EnumSet](#)”.

## **62. Envoltorios [wrappers] (concepto)**

Java diferencia entre tipos primitivos y objetos. Los envoltorios son objetos que rodean un tipo primitivo y proporcionan métodos. El valor (primitivo) contenido en un envoltorio se establece en el constructor, y no se puede modificar nunca.

<b><i>primitivo</i></b>	<b><i>envoltorio</i></b>
<a href="#">int</a>	<a href="#">Integer</a>
<a href="#">byte</a>	Byte
<a href="#">short</a>	Short
<a href="#">long</a>	Long
<a href="#">double</a>	<a href="#">Double</a>
<a href="#">float</a>	Float
<a href="#">boolean</a>	<a href="#">Boolean</a>

Ver “[autoboxing](#)”.

Los envoltorios proporcionan funciones de conveniencia tales como

- cargar datos de una cadena (de representación textual a formato interno)
- generar una cadena que representa el valor

Algunos muy frecuentemente utilizados:

```
int Integer.parseInt(String cadena)
double Double.parseDouble(String cadena)
boolean Boolean.parseBoolean(String cadena)
```

### 63. equals (método) public boolean equals(Object)

Es método sirve para indicar si un objeto es igual a otro.

Sería más propio decir “*si es equivalente a otro*” pues el programador puede definir la relación de equivalencia que guste.

Es un método heredado de la clase madre “[java.lang.Object](#)” y por tanto disponible en absolutamente todos los objetos que se creen; pero hay que tener en cuenta que el método heredado se limita a decir que un objeto es igual a sí mismo y diferente de todos los demás:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Como consecuencia, cuando un objeto no redefine equals(), es lo mismo igualdad que equivalencia

```
a.equals(b) <=> a == b
```

Ver “[hashCode\(\)](#)”.

Ver “[Igualdad \(==\)](#)”.

#### Tipos primitivos

---

Los tipos primitivos carecen de método equals(). Esto ocurre con enteros, reales, booleans y caracteres que simplemente o son iguales (==) o no lo son.

#### equals() y hashCode()

---

El método equals() va ligado al método [hashCode\(\)](#) que devuelve un entero que caracteriza al objeto, cumpliendo la propiedad de que dos objetos que sean equals() deben devolver el mismo hashCode(). Es decir

```
a.equals(b) => a.hashCode() == b.hashCode()
```

Esto permite que se pueda escribir código para buscar objetos iguales que primero miran el hashCode() y luego, si el hashCode() es igual, se mira el equals():

```
if (a.hashCode() == b.hashCode() && a.equals(b))
```

Las clases que hacen esta discriminación previa por hashCode(), simplemente funcionan mal si se redefine equals() pero no se redefine hashCode(). Esto ocurre, por ejemplo, en las clases [HashMap](#) y [HashSet](#) de la librería estándar de java.

Regla: si redefine equals(), debe redefinir [hashCode\(\)](#).

#### equals() y compareTo()

---

equals() define una relación de equivalencia, mientras que [compareTo\(\)](#) define una relación de orden. Es muy conveniente que sean coherentes

```
a.equals(b) <=> a.compareTo(b) == 0
```

El algunas ocasiones se puede usar otro criterio. Vease, por ejemplo, la clase BigDecimal de la librería estándar de java, que usa la relación de orden normal entre números Reales; pero tiene una peculiar interpretación de lo que es igualdad.

### Patrón básico

---

El método equals define una relación de equivalencia sobre objetos diferentes de null:

- reflexiva  
`x.equals(x) == true`
- simétrica  
`x.equals(y) <=> y.equals(x)`
- transitiva  
`x.equals(y) and y.equals(z) => x.equals(z)`
- y  
`x.equals(null) == false`

Por ello es muy típico esta forma de empezar cualquier método equals() que escribamos:

```
class C {
    @Override
    public boolean equals(Object x) {
        if (x == this)
            return true;
        if (x == null)
            return false;
        if (x instanceof C == false)
            return false;
```

... y se sigue con lo que es específico de la igualdad de objetos de clase C.

### Patrón: objetos iguales si son iguales algunos campos

---

A menudo la equivalencia no tiene más ciencia que decir que dos objetos son equivalentes si lo son algunos de sus campos.

Ejemplo, sean objetos que son coches

```
public class Coche {
    private String marca;
    private String modelo;
    private int puertas;
    private int km;
    private Color color;
```

Si queremos definir como iguales dos coches con igual marca, modelo y número de puertas:

```
@Override
public boolean equals(Object x) {
    if (x == this) return true;
    if (x == null) return false;
    if (!(x instanceof Coche)) return false;

    Coche otro = (Coche) x;
    if (puertas != otro.puertas) return false;
    if (!marca.equals(otro.marca)) return false;
    if (!modelo.equals(otro.modelo)) return false;
```

```
        return true;
    }
}
```

Si queremos tener en cuenta que alguno de los campos que son objeto pueda ser nulo:

```
@Override
public boolean equals(Object x) {
    if (x == this) return true;
    if (x == null) return false;
    if (!(x instanceof Coche)) return false;

    Coche otro = (Coche) x;
    if (puertas != otro.puertas) return false;
    if (marca == null) {
        if (otro.marca != null) return false;
    } else {
        if (!otro.marca.equals(marca)) return false;
    }
    if (modelo == null) {
        if (otro.modelo != null) return false;
    } else {
        if (!otro.modelo.equals(modelo)) return false;
    }

    return true;
}
```

Regla: si redefine equals(), debe redefinir [hashCode\(\)](#).

### Ejemplo de otros tipos de equivalencia

La equivalencia puede depender de las propiedades algebraicas del tipo de objeto de que se trate. Por ejemplo:

<i><b>Fraccion.java</b></i>
<pre>public class Fraccion {     private int num;     private int den;      @Override     public boolean equals(Object x) {         if (x == this)             return true;         if (x == null)             return false;         if (x.getClass() != this.getClass())             return false;         Fraccion fraccion = (Fraccion)x;         return this.num * fraccion.den == this.den * fraccion.num;     }      @Override     public int hashCode() {         return num ^ den;     } }</pre>

Regla: si redefine equals(), debe redefinir [hashCode\(\)](#).

## **64. Error (clase) java.lang.Error**

Es un tipo de excepciones que recoge situaciones francamente anormales tales como defectos en el propio sistema de ejecución de programas java. Son situaciones excepcionales que superan al programador y que por tanto, no se espera que este intente capturarlas, sino más bien ponerse en contacto con el fabricante para la reparación del producto.

Ver "[Exception](#)".

## **65. Errores (concepto)**

Son defectos en un programa que impiden alcanzar un resultado correcto.

Los errores pueden detectarse bien cuando intentamos compilar, bien durante la ejecución, bien al intentar verificar si el resultado es correcto.

### **de compilación (compile time)**

---

Son los errores que detecta el compilador ([javac](#)).

Suelen ser los errores más fáciles de comprender y reparar. Por ello todos los lenguajes de programación intentan que el compilador detecte cuantos más errores sea posible antes de intentar ejecutar un programa defectuoso. Pero no todo es detectable en compilación.

Un programa con errores de compilación no puede ejecutarse. De hecho, el compilador se niega a generar código interpretable.

Errores típicos de compilación

- errores en la escritura de las palabras reservadas
- errores en la sintaxis
- referencias a variables que no se han declarado previamente
- referencias a métodos que no se han declarado
- asignación a una variable de un valor para el que no está declarada
- etc.

### **de ejecución (run time)**

---

Son los errores que detecta el intérprete ([java](#)).

Estos errores suelen detener la ejecución. En el caso de java, el intérprete lanza una excepción que pudiera ser capturada por el propio programa.

Errores típicos de ejecución:

- objeto no creado (referencia a [NULL](#))
- acceso a un [array](#) fuera de su tamaño

Los errores de ejecución son fáciles de detectar cuando se producen; pero es difícil garantizar que no se van a producir. La forma habitual de prevenir estos errores consiste en probar el programa exhaustivamente de forma que los errores aparezcan en el laboratorio y no en producción.

### **semánticos**

---

Son aquellos que no respetan la semántica del lenguaje.

Errores típicos:

- intentar asignar a una variable un valor para el que no ha sido declarada
- llamar a un método con argumentos para los que no ha sido programado
- dar el mismo nombre a dos variables en el mismo [ámbito](#)

## **sintácticos**

---

Son aquellos que no respetan la sintaxis del lenguaje.

Errores típicos:

- no casar paréntesis de apertura y cierre
- no casar llaves de apertura y cierre
- olvidar ';' de fin de sentencia

## **66. Estilo de codificación [coding style]**

Es costumbre muy extendida establecer unas normas de estilo para escribir programas. Las normas de estilo son arbitrarias, pero permiten leer (y entender) los programas con facilidad.

Los siguientes párrafos presentan algunas reglas que son ampliamente utilizadas en java.

### **66.1. Nombres**

Nombres de clases, de métodos y de variables (campos incluidos):

- utilice nombres significativos: evite nombres crípticos, abreviaciones que nadie entiende y nombres genéricos que no significan nada.
- use `getX()` para métodos [getters](#)
- use `void setX(valor)` para métodos [setters](#)
- con campos booleanos, use `isX()` en métodos [getters](#)
- los nombres de clases deben empezar por letra mayúscula
- los nombres de métodos deben empezar por letra minúscula
- los nombres de variables y campos deben empezar por letra minúscula
- si el nombre consta de varias palabras, empiece cada una con letra mayúscula; por ejemplo la variable "diaDelMes" o la clase "AgendaDeTelefonos"
- los nombres de constantes deben usar letras mayúsculas; si consta de varias palabras, sepárelas con caracteres de subrayado; por ejemplo `ALTURA_DEL_TRIANGULO`

### **66.2. Sangrado y espacio en blanco [layout]**

Elija un tamaño de sangrado y úselo de forma sistemática en todo el programa. Es habitual usar 2, 3 o 4 espacios en blanco, o un carácter TABULADO como elemento de sangrado.

- cuando entre en un bloque (sentencias entre llaves {}), incremente en una unidad de sangrado todas las sentencias dentro del bloque
- la llave que cierra un bloque debe estar en una línea independiente y alineada con el primer carácter de la línea en la que está la llave de apertura correspondiente



<pre> ..if (condicion) { ....sentencias; ..} else { ....sentencias; ..} </pre>
<pre> ..if (condicion) { ....sentencias; ..} else if (condicion) { ....sentencias; ..} else { ....sentencias; ..} </pre>
<pre> ..for (int i= 0; i &lt; array.length; i++) { ....sentencias; ..} </pre>
<pre> ..for (int i= 0; i &lt; array.length; i++) ..{ ....sentencias; ..} </pre>
<pre> ..for (int Object x: lista) { ....sentencias; ..} </pre>
<pre> ..while (condicion) { ....sentencias; ..} </pre>
<pre> ..do { ....sentencias; ..} while (condicion); </pre>
<pre> ..public tipo metodo(argumentos) { ....sentencias; ..} </pre>

- use siempre llaves en las estructuras de control

<i><b>mejor</b></i>	<i><b>peor</b></i>
<pre> if (condicion) {     sentencias; } else if (condicion) {     sentencias; } else {     sentencias; } </pre>	<pre> if (condicion)     sentencias; else if (condicion)     sentencias; else     sentencias; </pre>

- deje un espacio en blanco entre cierra paréntesis y abre llave

<i><b>mejor</b></i>	<i><b>peor</b></i>
<pre> if (condicion) {     sentencias; } else if (condicion) {     sentencias; } </pre>	<pre> if (condicion){     sentencias; } else if (condicion){     sentencias; } </pre>

<pre> } else {     sentencias; } </pre>	<pre> } else {     sentencias; } </pre>
---	---

- deje espacio alrededor de los operadores

<i>mejor</i>	<i>peor</i>
2 * 3 + 4 * 5	2*3+4*5

- deje una línea en blanco entre un método y el siguiente

## 67. Etiquetas [labels] (concepto)

En Java sólo se usan para identificar bucles y forzar sentencias “[break](#)” y “[continue](#)”.

**En general, dan pie a código difícilmente inteligible, por lo que no se recomienda su uso salvo en ocasiones muy justificadas.**

Lo normal es que una sentencia “break” fuerce la salida del bucle más interno en ejecución. Si queremos salir de otro bucle más externo, lo marcaremos con una etiqueta X: y saldremos de él escribiendo

```
break X;
```

Sirva el siguiente método de ejemplo, aunque parece evidente que se podría programar con ayuda de algún método auxiliar y, probablemente, quedaría más claro:

<i>ejemplo break etiquetado</i>
<pre> public class TrianguloPitagorico {     public static void main(String[] argumentos) {         int a = -1, b = -1, c = -1;         <b>busqueda:</b>         for (int ijk = 1; ijk &lt; 100; ijk++)             for (int i = 1; i &lt; ijk; i++)                 for (int j = 1; i + j &lt; ijk; j++)                     for (int k = 1; i + j + k &lt; ijk; k++)                         if (i * i + j * j == k * k) {                             a = i; b = j; c = k;                             <b>break busqueda;</b>                         }         System.out.println("a= " + a + "; b= " + b + "; c= " + c + ";");     } } </pre>

Lo normal es que una sentencia “continue” fuerce el salto a la siguiente vuelta del bucle más interno en ejecución. Si queremos retomar otro bucle más externo, lo marcaremos con una etiqueta X: y regresaremos a él escribiendo

```
continue X;
```

Sirva el siguiente método de ejemplo, aunque parece evidente que se podría programar con ayuda de algún método auxiliar y, probablemente, quedaría más claro:

### ***ejemplo continue etiquetado***

```
/**
 * Encuentra la primera línea tal que los datos de las columnas
 * están ordenados (valores estrictamente crecientes).
 *
 * @param matriz no necesariamente rectangular.
 * @return fila con datos ordenados, o -1 si no hay ninguna.
 */
static int fileOrdenada(int[][] matriz) {
    filas:
    for (int fila = 0; fila < matriz.length; fila++) {
        for (int columna = 1;
             columna < matriz[fila].length;
             columna++) {
            int v1 = matriz[fila][columna - 1];
            int v2 = matriz[fila][columna];
            if (v1 >= v2)
                continue filas;
        }
        return fila;
    }
    return -1;
}
```

## **68. Excepciones [exceptions] (concepto)**

Constituyen un mecanismo para propagar rápidamente situaciones excepcionales (típicamente errores que se detectan) saltándose el flujo normal de ejecución del programa.

Materialmente, una excepción es un objeto que ...

- debe ser de una subclase de ***java.lang.Throwable***

```
class Throwable
class Exception extends Throwable
class MisErrores extends Exception
```

- se crea con **new** (o sea, usando un constructor)

```
MisErrores error = new MisErrores("clave errónea", clave);
```

- entre sus campos suele llevar información de por qué se crea (o sea, de qué circunstancia excepcional es mensajero)

```
class MisErrores extends Exception {
    private Clave clave;
    MisErrores(String msg, Clave clave) {
        super(msg);
        this.clave = clave;
    }
}
```

- lleva información acerca de en qué punto del código se crea y cual es la serie de llamadas a métodos que han llevado a ese punto del código (a esta información se la llama "traza"); la traza se puede imprimir en consola:

```
error.printStackTrace();
```

- se lanza con **throw**

```
throw error;
```

- el lanzamiento interrumpe la ejecución normal del código, es decir que la siguiente instrucción no se ejecuta; por ello el lanzamiento de la excepción suele estar en un bloque condicional

```
if (isOk(clave) == false)
    throw new MisErrores("clave incorrecta", clave);
... usamos la clave ...
```

- se pueden capturar con bloques catch si

1. se lanzan dentro de un bloque **try** y
2. existe un bloque **catch** que captura excepciones de la clase de la excepción o de alguna [superclase](#)

```
try {
    ...
    throw new MisErrores(...);
    ...
} catch (MisErrores e) {
    System.err.println(e);    // informa del error producido
    e.printStackTrace();    // describe dónde se produjo
}
```

- si la excepción se lanza en un método que no desea (o no puede, o no conviene) capturarla, el método puede propagarla, para lo cual debe indicarlo en su cabecera

```
void miMetodo(argumentos) throws MisErrores {
    ...
    throw ...;
    ...
}
```

Si en la traza (cadena de llamadas a métodos) encontramos un bloque **try**, la excepción puede ser capturada por un bloque **catch**, siempre y cuando el bloque **catch** tenga previsto capturar excepciones de esta clase o de alguna [superclase](#).

Si en la traza no hubiera ningún bloque **try** con un **catch** adecuada, la excepción sale del programa, pasando al soporte de ejecución que

1. informa en consola del problema (es decir, imprime la traza)
2. detiene la ejecución

### ¿Pueden haber 2 excepciones "volando" a la vez?

No.

Cuando se lanza una excepción, esta va saltando hasta que la pilla el primer catch que encuentra, momento en que deja de volar.

El catch que pilla una excepción puede hacer varias cosas:

1. puede absorberla y se acaba el asunto

```
try {  
    ...  
    throw new Exception("error x");  
    ...  
} catch (Exception e) {  
    System.err.println("aquí se captura la excepción " + e);  
}
```

2. puede relanzarla y la excepción sigue volando hasta que encuentre otro catch

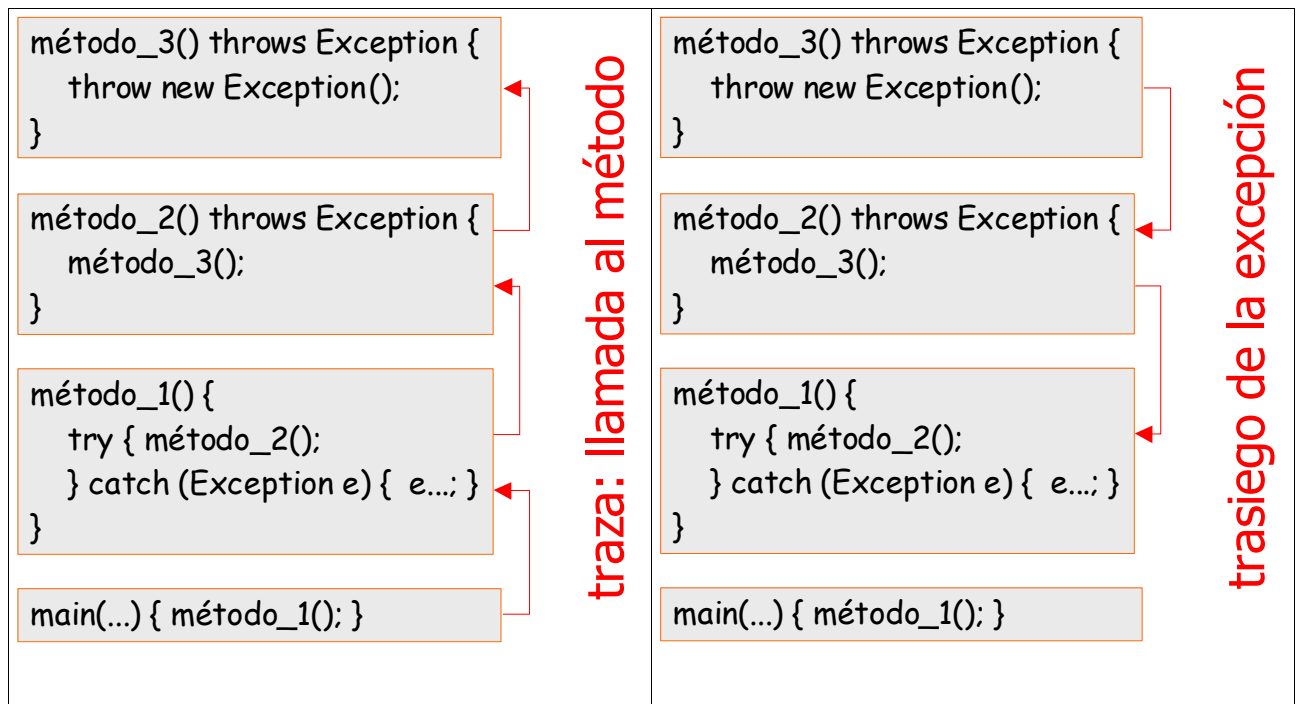
```
try {  
    ...  
    throw new Exception("error x");  
    ...  
} catch (Exception e) {  
    System.err.println("por aquí pasa la excepción " + e);  
    throw e;  
}
```

3. puede lanzar otra excepción, perdiéndose la primera

```
try {  
    ...  
    throw new Exception("error x");  
    ...  
} catch (Exception e) {  
    System.err.println("aquí se captura la excepción " + e);  
    throw new Exception("error y");  
}
```

## **69. Excepciones y métodos**

Si una [excepción](#) se lanza en dentro de un [método](#), puede que este la capture o que no. Si no la captura debe informar en su cabecera de que puede ser lanzada. Si un método propaga una excepción, esta deberá ser considerada en el código llamante que, a su vez, puede capturarla o retransmitirla



Se debe informar explícitamente en la cabecera del método, salvo que la excepción sea subclase de [java.lang.Error](#) o de [java.lang.RuntimeException](#), en cuyo caso se puede informar o no decir nada.

Ver "[Exception](#)".

## 70. *Exception (clase) java.lang.Exception*

Es una clase como fundamental dentro de la jerarquía de clases que definen objetos que se pueden lanzar.

```

java.lang.Throwable
    Error
        VirtualMachineError
        ...
    Exception
        ClassNotFoundException
        DataFormatException
        IOException
        NoSuchFieldException
        NoSuchMethodException
        ...
    RuntimeException
        ArithmeticException
        ClassCastException
        IllegalArgumentException
        IllegalStateException
        IndexOutOfBoundsException
        NegativeArraySizeException
        NoSuchElementException
        NullPointerException
        UnsupportedOperationException
        ...

```

Ver "[class java.lang.Exception](#)".

La relación completa de clases es interminable. Las anteriores son algunas de las más comunes, invitándose al programador a desarrollar sus propias clases, subclases de las anteriores, sabiendo que

- las clases que hereden de [RuntimeException](#) no necesitan declararse en la cabecera de los [métodos](#): se utilizan para errores imprevistos tales como errores del programador
- las clases que hereden de **Exception** sí deben declararse en la cabecera de los [métodos](#) que deseen propagarlas (la única alternativa es capturarlas dentro del método). Se utilizan para errores previstos por la aplicación: datos erróneos para los que se sabe qué hacer cuando se presentan

La clase Exception proporciona algunos métodos interesantes:

**void printStackTrace()**

imprime la traza: serie de métodos llamados sucesivamente para llegar al punto donde se crea. Lo normal es crear el objeto cuando se produce la circunstancia que provoca su lanzamiento, de forma que la traza indica la serie de llamadas a métodos que llevan a la ocurrencia de la incidencia reportada.

**Exception()**

**constructor**

crea un objeto sin mensaje asociado

**Exception(String mensaje)**

**constructor**

crea un objeto con un mensaje asociado

**String getMessage()**

devuelve el mensaje introducido en el constructor

Es muy habitual que las excepciones propias sean [subclases](#) de las Exception:

<b>class MisErrores</b>
<pre>public class MisErrores extends Exception {     private Dato dato;      public MisErrores() { super(); }      public MisErrores(String msg) { super(msg); }      public MisErrores(String msg, Dato dato) {         super(msg);         this.dato = dato;     }      public Dato getDato() { return dato; }</pre>

### **¿cuándo tener excepciones propias?**

Aunque frecuentemente se pueden usar las clases proporcionadas por el entorno java, las excepciones propias

- permiten documentar mejor el programa
- permiten discriminar por medio de varios bloques [catch](#) consecutivos

- permiten transportar datos adicionales desde donde se crean a donde se capturan

---

### ¿cuándo lanzar `Error` o subclases de ella?

---

Las excepciones [java.lang.Error](#) son competencia del entorno de ejecución de programas java. Es muy raro que sean lanzadas por un programa de usuario.

---

### ¿cuándo lanzar `Exception` o subclases de ella?

---

Es lo normal cuando el programa se encuentra con situaciones en las que el tratamiento normal de los datos no puede proseguir.

Las excepciones de tipo `Exception` deben aparecer explícitamente en la cabecera del método que puede lanzarlas.

Se usan excepciones de tipo `Exception` cuando

- no es responsabilidad del que llama al método, saber si los argumentos son correctos o dan pie a un error
- es responsabilidad del que llama gestionar la excepción cuando se produzca
- a veces se dice que estas excepciones se utilizan para tratar errores recuperables, pues el método llamante debe estar preparado para recibirlas y actuar en consecuencia

Se dice que son excepciones *informativas*.

---

### ¿cuándo lanzar `RuntimeException` o subclases de ella?

---

Es lo normal cuando el programa se encuentra con una situación inmanejable que no debería haberse producido; pero impide seguir.

Lo más habitual es que este tipo de excepciones las lance el propio entorno de ejecución ante errores del programador tales como

- indexación de un [array](#) fuera de rango
- referencia a ningún objeto (**null**)
- errores de formato, operaciones matemáticas, etc.

Las excepciones de tipo `RuntimeException` no es necesario declararlas en la cabecera de los métodos que pueden lanzarlas, aunque puede documentarse. Como regla puede decirse que es conveniente poner explícitas aquellas excepciones que se lanzan porque el programador lo ha escrito en su código.

Se usan excepciones de tipo `RuntimeException` cuando

- es responsabilidad del que llama al método, saber si los argumentos son correctos o darían pie a un error
- el que llama no prevé ninguna excepción, ni tratamiento explícito alguno; por ello a veces se dice que son excepciones para errores irrecuperables
- el método llamado se protege de fallos del programador lanzando una excepción de ejecución

Se dice que son excepciones *defensivas*.

## 71. Expresiones [expressions] (concepto)

Con [números](#) y variables numéricas se pueden escribir expresiones aritméticas.

Con [booleanos](#) y variables booleanas se pueden construir expresiones lógicas.



Las expresiones aritméticas y lógicas se pueden combinar para crear expresiones más complejas.

## aritméticas

<i><b>expresiones aritméticas</b></i>		
suma	$a + b$	$2 + 2 = 4$
resta	$a - b$	$5 - 6 = -1$
producto	$a * b$	$2 * 10 = 20$
cociente	$a / b$	$10 / 3 = 3$
		$10.0 / 3.0 = 3.333$
resto	$a \% b$	$10 \% 3 = 1$
		$10.0 \% 3.0 = 1.0$

Si ambos operandos son enteros, el resultado es entero. Si alguno de los operandos es real, el resultado es real. Si uno de los operandos es entero y el otro es real, en entero se pasa a real y se operan como reales. Ver "[promoción](#)".

Si se dividen enteros, el cociente es el resultado de la división de enteros. Si alguno de los operandos es un número real, el resultado es otro número real. Ver "[promoción](#)".

El resto sobre reales se calcula como sobre enteros; pero devuelve un valor real.

## lógicas

<i><b>expresiones lógicas</b></i>		
menor	$a < b$	$0 < 7 = \text{true}$
menor o igual	$a \leq b$	$7 \leq 7 = \text{true}$
igual	$a == b$	$0 == 1 = \text{false}$
mayor	$a > b$	$0 > 7 = \text{false}$
mayor o igual	$a \geq b$	$7 \geq 7 = \text{true}$

```
boolean bisiestro(int año) {
    boolean multiplo4 = año % 4 == 0;
    boolean multiplo100 = año % 100 == 0;
    boolean multiplo400 = año % 400 == 0;
    return (multiplo4 && (! multiplo100)) || multiplo400;
}
```

## expresiones condicionales

Forma compacta de decidir entre dos valores

condición ? valor\_1 : valor\_2

si es cierta la condición se toma el prime valor; si no el segundo. Ambos valores deben ser del mismo tipo o tipos compatibles (vía *casting*).

<b>expresiones condicionales</b>	
<b>forma compacta</b>	<b>forma clásica</b>
Tipo variable = condicion ? v1 : v2;	Tipo variable; if (condicion) variable = v1; else variable = v2;
return condicion ? v1 : v2;	if (condicion) return v1; return v2;

### **precedencia de operadores**

Con las expresiones elementales descritas se pueden escribir expresiones más complejas

$$2 * 3 + 4 * 5 \rightarrow 26$$

Cuando se combina expresiones hay unas reglas de precedencia. Si no las recuerda o prefiere no arriesgarse a recordarlas mal, use paréntesis para indicar exactamente en qué orden se van evaluando los resultados intermedios:

$$(2 * 3) + (4 * 5) \rightarrow 26$$

$$2 * (3 + 4) * 5 \rightarrow 70$$

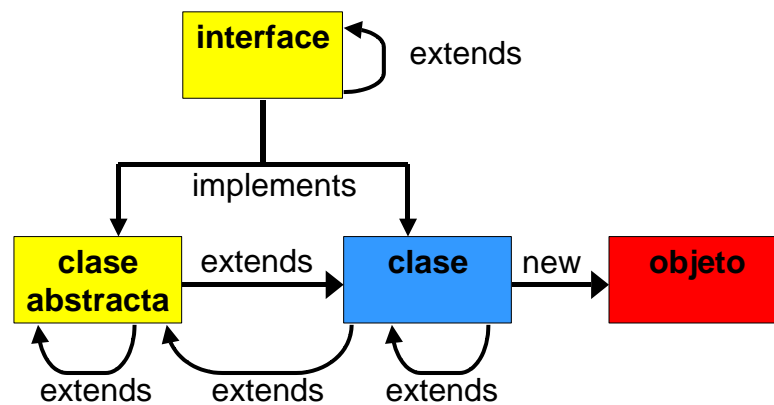
$$1 \leq \text{día} \ \&\& \ \text{día} \leq 31 \rightarrow \text{true si "día" está en el rango [1, 31]}$$

Las reglas de precedencia en java son las siguientes, ordenadas por orden de precedencia (los operadores se aplican en el orden en que aparecen en esta lista):

<b>reglas de precedencia</b>
<ol style="list-style-type: none"> <li>1. ++ -- !</li> <li>2. * / %</li> <li>3. + -</li> <li>4. &lt; &lt;= &gt; &gt;=</li> <li>5. == !=</li> <li>6. &amp;</li> <li>7. ^</li> <li>8.  </li> <li>9. &amp;&amp;</li> <li>10.   </li> </ol>

## **72. extends (palabra reservada)**

Permite crear clases que son extensión de otras (subclases).



Una interfaz de puede extender añadiendo nuevas constantes y/o métodos cuyo cuerpo no se explicita.

Una clase (normal o abstracta) se puede extender añadiendo nuevos métodos concretos o abstractos. Si como resultado de la extensión no queda ningún método abstracto, el resultado será una clase normal. Si como resultado de la extensión quedan métodos abstractos, la clase será abstracta.

### 73. Extensión (concepto)

Cuando

```
class B extends A { ... }
```

se dice que A es [superclase](#) de B y que B es [subclase](#) de A.

La nueva clase B dispone automáticamente de todos los miembros **public**, **protected** y “**de paquete**” de la clase A, miembros que puede usar o [redefinir](#).

Ver “[herencia](#)”.

En Java, una clase sólo puede heredar de otra única clase. No existe la posibilidad de herencia múltiple. Esto es incorrecto:

```
class B extends A1, A2 { ... }
```

### 74. Fábricas [factories] (concepto)

Son [métodos](#) de clase ([static](#)) que crean objetos.

A diferencia de los métodos constructores

- pueden tener cualquier nombre
- pueden haber varios con el mismo número y tipo de argumentos, siempre y cuando difieran en el nombre
- pueden no llegar a crear ningún objeto (típicamente, devolviendo “**null**”)

<i>class Punto2D</i>
<pre>public class Punto2D {     // representación interna: coordenadas cartesianas     private double x, y;      // constructor interno: coordenadas cartesianas</pre>

<i><b>class Punto2D</b></i>
<pre> private Punto2D(double x, double y) {     this.x = x;     this.y = y; }  // fábrica: coordenadas cartesianas public static Punto2D cartesianas(double x, double y) {     return new Punto2D(x, y); }  // fábrica: coordenadas polares public static Punto2D polares(double modulo, double angulo) {     double x = modulo * Math.cos(angulo);     double y = modulo * Math.sin(angulo);     return new Punto2D(x, y); } </pre>

Ver “[constructores](#)”.

## 75. Fichero fuente [source code file]

Dícese de los ficheros que contienen programas java.

Ver “[ficheros .java](#)”.

## 76. Ficheros .java

Las clases e interfaces se editan en ficheros que se caracterizan por tener la extensión .java.

Un fichero .java

- puede contener una o más clases o interfaces
- sólo puede contener una clase o interface calificada como **public**
- si hay una clase o interface **public**, el nombre del fichero coincidirá con el de dicha clase

Si la clase o interface pertenece a un paquete, debe estar en una estructura de directorios que refleje la estructura de paquetes.

Esta regla NO es obligatoria; pero si se la salta probablemente se encuentre en confusas situaciones para localizar las clases deseadas.

<i><b>paquetes y directorios</b></i>	
si es del paquete ...	debe estar en ...
package a;	.../a
package a.b;	.../a/b

Para compilar un fichero, use la herramienta “javac”. El resultado es un fichero .class que también se atendrá a la regla que relaciona directorios con paquetes (pero, en este caso, se encarga la propia herramienta “java”). También puede usar cualquiera de los entornos de desarrollo disponibles.

## 77. *final* (palabra reservada)

Es un calificativo que se puede aplicar a variables, métodos o clases y quiere decir que es invariable.

### **final class**

Cuando una clase se califica como **final**, no puede ser extendida. Todos los miembros de una clase **final** pasan a ser **finales**.

### **final método**

Cuando un método se califica como **final**, no puede ser [redefinido](#) si la clase a la que pertenece fuera [extendida](#) por otra clase.

La ventaja que aporta el calificativo **final** es que impide el [polimorfismo](#) pues al no poder ser redefinidos los métodos, nunca habrá sorpresas en tiempo de ejecución.

### **final variable de clase**

La variable se convierte en constante. Debe ser inicializada o bien junto a la propia definición, o en la zona **static** de inicialización de variables.

### **final variable de objeto**

La variable se convierte en constante. Debe ser inicializada o bien junto a la propia definición, o bien en todos y cada uno de los constructores de la clase.

### **final argumento formal de un método**

El argumento se convierte en constante, dentro del método; es decir, su valor no puede ser modificado.

### **final variable local (o automática)**

La variable se convierte en constante; es decir, su valor no puede modificarse.

La identificación de elementos finales puede proporcionar alguna mejora en tiempo de ejecución pues se evitan los algoritmos de resolución de polimorfismo y el compilador puede optimizar el código. No obstante, hay que advertir que estas mejoras suelen ser muy modestas.

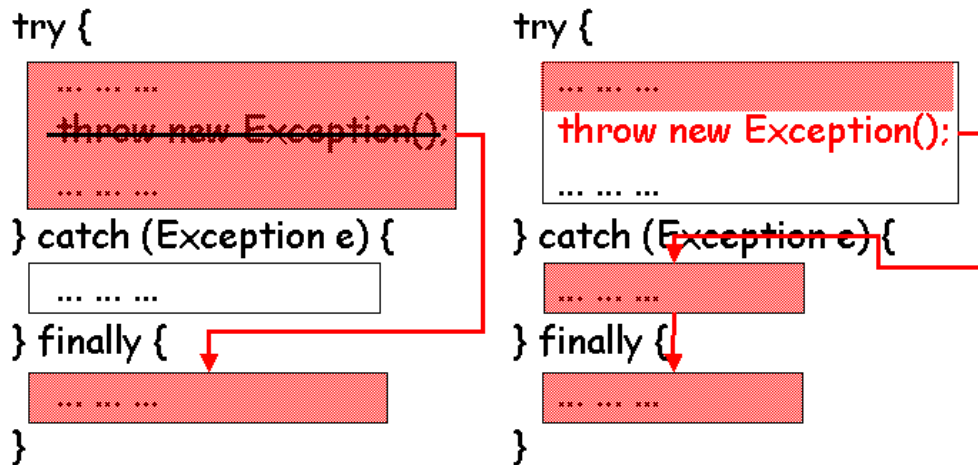
NOTA. Un error frecuente consiste en declarar como final una variable que es referencia a un objeto y luego pensar que el objeto no puede ser modificado. Final sólo garantiza que la variable siempre referenciará el mismo objeto, independientemente de que el objeto en sí evolucione.

```
Set<String> diccionario(String idioma) {  
    final Set<String> frases = new HashSet<String>();  
    if (idioma.equalsIgnoreCase("español")) {  
        frases.add("hola");  
        frases.add("adiós");  
    }  
    if (idioma.equalsIgnoreCase("english")) {  
        frases.add("hi");  
        frases.add("bye");  
    }  
    // frases = ...;    provocaría un error de re-asignación  
    return frases;  
}
```

## 78. *finally* (palabra reservada)

Componente sintáctico de sentencias "try".

El bloque finally se ejecuta siempre al acabar el bloque try, tanto si acaba bien como si acaba mal, sea cual sea el significado de acabar bien o mal.



El ejemplo siguiente muestra como el bloque puede acabar bien (calculando la raíz cuadrada de un real positivo) o mal (por diferentes motivos de formato o números negativos); pero siempre incrementa el número de pruebas realizadas:

<i><b>Inversos.java</b></i>
<pre> public class Inversos {     public static void main(String[] args) {         Scanner scanner = new Scanner(System.in);         int pruebas = 0;         while (true) {             double inverso = 0;             try {                 String x = scanner.next();                 if (x.equals("fin"))                     break;                 int valor = Integer.parseInt(x);                 inverso = 100 / valor;             } catch (Exception e) {                 System.out.println("  -&gt; " + e);                 inverso = 0;             } finally {                 pruebas++;                 System.out.println("  -&gt; " + pruebas + ": " + inverso);             }         }     } } </pre>
<i><b>ejecución</b></i>
<pre> \$ java Inversos 2   -&gt; 1: 50.0 dos   -&gt; java.lang.NumberFormatException: For input string: "dos"   -&gt; 2: 0.0 </pre>

### ***Inversos.java***

```
0
-> java.lang.ArithmeticException: / by zero
-> 3: 0.0
fin
-> 4: 0.0
```

El bloque **finally** puede emplearse incluso sin bloques **catch**, aprovechando su característica de que siempre se ejecuta al salir del bloque **try**.

El siguiente ejemplo muestra como medir el tiempo que tarda un algoritmo independientemente de que termine bien o mal:

Concretamente, en el ejemplo se busca un cero de una función en un intervalo dado. Como algoritmo se usa el consistente en ir dividiendo el intervalo en mitades hasta acotarlo. Hay varias razones por las que puede no funcionar: la función no está definida en algún punto, o presenta una discontinuidad, o simplemente no pasa por cero. Todas estas causas se traducen en el lanzamiento de una excepción informativa.

### ***Funcion.java***

```
public class Funcion {
    private static final double ERROR = 1E-12;

    public double y(double x) {
        return Math.tan(x);           // discontinuidad en x= PI/2
    }

    public double cero(double a, double z)
        throws Exception {
        if (a > z) throw new Exception("a > z");
        double fa = y(a);
        double fz = y(z);
        if (Math.abs(fa) < ERROR) return a;
        if (Math.abs(fz) < ERROR) return z;
        if (fa * fz > 0)
            throw new Exception("no pasa por cero en el intervalo");
        double m = (a + z) / 2;
        double fm = y(m);
        if (Math.abs(fm) < ERROR) return m;
        if (Math.abs(a - z) < ERROR)
            throw new Exception("función discontinua");
        if (fa * fm < 0)
            return cero(a, m);
        else
            return cero(m, z);
    }

    public static void main(String[] args)
        throws Exception {           // nadie captura la excepción
        long t0 = System.currentTimeMillis();
        try {
            Funcion f = new Test();
            System.out.println("raíz= " + f.cero(1, 3));
        } finally {
```

<i>Funcion.java</i>
<pre>         long t2 = System.currentTimeMillis();         System.out.println((t2 - t0) + "ms");     } } </pre>

### ¿Qué hace finally con una excepción lanzada y no capturada dentro del try?

No hace nada: la excepción sale fuera del bloque [try](#).

### ¿Puede hacerse "return" desde dentro del finally?

Sí.

El efecto es que el método en donde se encuentre termina inmediatamente.

Tanto si hemos llegado al "finally" por un "return" o un "throw" dentro del "try-catch", el efecto es el mismo: el "return" del "finally" es el único que tiene efecto.

### ¿Puede lanzarse una excepción desde dentro del finally?

Sí.

El efecto es que la ejecución lineal termina inmediatamente y sale volando la excepción a la búsqueda de un "catch" que la atrape..

Tanto si hemos llegado al "finally" por un "return" o un "throw" dentro del "try-catch", el efecto es el mismo: la excepción lanzada por el "finally" sale volando y java se olvida de lo demás.

## **79. float (palabra reservada)**

<i>float : números reales (coma flotante)</i>	
<b>ocupación</b>	4 bytes = 32 bits
<b>máximo (absoluto)</b>	$\text{Float.MAX\_VALUE} = (2 \cdot 2^{23}) \cdot 2^{127} = 3.4 \cdot 10^{38}$
<b>mínimo (distinto de cero)</b>	$\text{Float.MIN\_VALUE} = 2^{-149} = 1.4 \cdot 10^{-45}$

Ver "[Números](#)".

## **80. for (palabra reservada)**

Los bucles "for" se ejecutan un número determinado de veces.



<i>java</i>	<i>flujo</i>
<pre> for (inicialización;     condición;     actualización)     acción;  for (inicialización;     condición;     actualización) {     acción 1;     acción 2;     ...     acción ...; } </pre>	<pre> graph TD     Start(( )) --&gt; Init[inicialización;]     Init --&gt; Cond{condición}     Cond -- true --&gt; Act[acción;]     Act --&gt; Upd[actualización;]     Upd --&gt; Cond     Cond -- false --&gt; Exit(( )) </pre>

### **bucles "for" con contador**

Los bucles **"for"** tradicionalmente emplean un contador que es una variable (local al bucle).

*Local al bucle quiere decir que su [ámbito](#) es el propio bucle, no pudiendo utilizarse dicha variable fuera del bucle.*

<i>Ejemplos de bucles "for" con contador</i>
<pre> for (int i = 0; i &lt; 10; i++)     System.out.print(i); </pre>
<pre> Clase[] array= ...;  for (int i = 0; i &lt; array.length; i++) {     Clase x = array[i];     procesa(x); } </pre>
<pre> int factorial(int m) {     int fact = 1;     for (n = m; n &gt; 0; n--)         fact*= n; } </pre>

### **bucles "for" con iterador (for each)**

Alternativamente, podemos ir iterando sobre los elementos de un conjunto de valores sin variable contador. Esto se puede hacer

- sobre [arrays](#)
- sobre clases que implementen la "[interface Iterable<T>](#)":  
por ejemplo, listas y conjuntos

### ***Ejemplos de bucles “for” con iterador***

```
Clase[] array = ...;
```

```
for (Clase x: array)
    procesa(x);
```

```
List<Clase> lista = ...;
```

```
for (Clase x: lista)
    procesa(x);
```

```
Set<Clase> conjunto = ...;
```

```
for (Clase x: conjunto)
    procesa(x);
```

```
Collection <Clase> coleccion = ...;
```

```
for (Clase x: coleccion)
    procesa(x);
```

```
enum Color { ROJO, NARANJA, AMARILLO, VERDE, AZUL, AÑIL, VIOLETA };
```

```
for (Color color: Color.values()) {
    procesa(color);
}
```

```
for (Color color: EnumSet.range(Color.NARANJA, Color.AZUL)) {
    procesa(color);
}
```

Si tenemos una clase que proporciona un iterador:

```
class Almacen implements Iterable<Dato> {
    public Iteratos<Dato> iterator() { ... }
}
```

entonces podemos recorrer directamente

```
Almacen almacen;
for (Dato dato: almacen) { ... }
```

Ver “[Iterable<T>](#)”.

### **recorrido de arrays**

Los [arrays](#) pueden recorrerse indistintamente con un contado o barriendo con un iterador.

```
int[] datos = { 1, 2, 3, 4 };
```

```
for (int i = 0;
     i < datos.length;
     i++) {
    System.out.print(datos[i] + " ");
}
```

```
for (int dato: datos) {
    System.out.print(dato + " ");
}
```

La segunda forma (derecha) es más compacta y debería preferirse a la primera (izquierda) salvo cuando el contador “i” sea necesario para algo más.

## bucles "for" con varios contadores

Se pueden escribir bucles con varias variables que se van actualizando simultáneamente:

```
for (int i = 0, j = 10; i < 9 && j > 0; i++, j--)  
    acción;
```

## bucles "for" degenerados

Los componentes sintácticos de un bucle son opcionales, pudiendo hacerse "cosas raras"

<b>Bucles "for" degenerados</b>	
for ( ; condición ; actualización)	Se usa una variable externa.
for (inicialización ; ; actualización)	La condición de terminación es interna; típicamente con un "break" o un "return".
for (inicialización ; condición ; )	La condición de actualización es interna.
for ( ; ; )	Bucle que se ejecuta infinitas veces o hasta que una condición de terminación interna lo aborta.

Ver "[bucles](#)".

## 81. *format (método) void format(String, Object ...)*

Método de las clases [PrintStream](#) y [PrintWriter](#).

Sinónimo de printf().

Imprime una serie de valores siguiendo el formato proporcionado como primer argumento.

Ver "[Formatter](#)" donde se explica la notación usando en la descripción de formato.

## 82. *Friendly*

Ver "[visibilidad](#)": ámbito de código en el que un elemento de Java puede referenciarse por su nombre.

Se dice que un elemento es "*friendly*" cuando no es ni "**public**", ni "**protected**", ni "**private**". En este vademecum lo denominaremos "*de paquete*".

Los elementos "*de paquete*" pueden referenciarse desde cualquier punto del código dentro del mismo paquete en el que se define.

### **class X**

La clase X puede referenciarse desde cualquier punto del código dentro del mismo paquete en el que se define .

Se pueden definir clases "*de paquete*" dentro de otras clases o en ficheros que contienen alguna otra clase "pública".

### **resultado método (argumentos)**

El método puede referenciarse (ejecutarse) desde cualquier punto del código dentro del mismo paquete en el que se define .

Se pueden definir métodos “*de paquete*” dentro de cualquier clase.

#### variable

La variable puede referenciarse (leer o escribir) desde cualquier punto del código dentro del mismo paquete en el que se define .

Se pueden definir variables “*de paquete*” como campos de cualquier clase. Pero no se recomienda; es preferible definir las variables como “**private**” y establecer métodos de acceso para lectura (*getX()* o *isX()*) y escritura (*setX()*).

### 83. Genéricos [generics] (concepto)

Bajo la expresión “soporte de genéricos” java proporciona dos facilidades de programación bastante diferentes:

- tipos genéricos; que son clases parametrizadas por uno o más tipos que deben ser facilitados por el programador cuando quiera usar la clase creando objetos
- métodos genéricos, que son métodos en los que los argumentos y/o el resultado incluyen referencias a tipos que no se conocerán hasta que vayamos a usar el método

Probablemente el principal motivo para el uso de genéricos en java sea la necesidad de disponer de colecciones homogéneas de objetos (listas, conjuntos, etc. facilitados en el paquete `java.util`). Así el ejemplo más habitual del uso de genéricos es la clase lista genérica, definida como

```
public interface List<E>
public class ArrayList<E> implements List<E>
```

donde E queda a definir. Se dice que E es un tipo formal.

En inglés se suele emplear la expresión “*type parameter*” para referirse a esos parámetros formales que no se refieren a valores, sino a tipos de valores. En la traducción al español, “parámetros de tipo” suena muy extraño, por lo que emplearemos la forma más técnica “tipo formal” indicando que cuando se vaya a utilizar la clase hay que proporcionar un “tipo real”.

Con esta clase genérica podemos crear objetos de diferentes tipos

- `new ArrayList<String>` es una lista de String
- `new ArrayList<Integer>` es una lista de Integer
- `new ArrayList<Punto>` es una lista de objetos de clase Punto

donde todos ellos se caracterizan por crear listas homogéneas (todos los elementos son del mismo tipo), resultando programas limpios y, probablemente, con menos errores.

El objetivo de los genéricos con java es

- desplazar a tiempo de compilación los tradicionales errores de ejecución que ocurrían en programas con fuerte uso de [downcasting](#)
- si el compilador no se queja, se puede asegurar que no habrá errores de tipo de datos ([casting](#)) en ejecución

“Quejarse” quiere decir que el compilador protesta como se puede ver en el siguiente ejemplo, al compilar la clase `Queue_Array` que se usa un poco más adelante:

```
$ javac Queue_Array.java
```

Note: Queue\_Array.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.

```
jam@toshiba-a10 /c/home/doc/PET/exs/genericos
$ javac -Xlint:unchecked Queue_Array.java
Queue_Array.java:23: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: T
    T x = (T) data[0];
           ^
1 warning
```

## clases genéricas en el package java.util

---

El paquete java.util proporciona varias clases genéricas. La siguiente relación no pretende ser exhaustiva, sino aquellas que el autor, personalmente, considera de uso más frecuente:

```
interface Collection<E> extends Iterable<E>
interface Comparator<E>
interface Enumeration<E>
interface List<E> extends Collection<E>
interface Map<K, V>
interface Queue<E> extends Collection<E>
interface Set<E> extends Collection<E>
interface SortedMap<K, V> extends Map<K, V>
interface SortedSet<E> extends Set<E>

class ArrayList<E> ...
class EnumSet<E> ...
class HashMap<K, V> ...
class HashSet<E>
class LinkedList<E> ...
class PriorityQueue<E> ...
class Stack<E> ...
```

## creación de clases genéricas

---

El siguiente ejemplo muestra como crear un objeto que es una asociación de dos datos de tipos formales S y T:

<i><b>class Pareja</b></i>
<pre>public class Pareja&lt;S, T&gt; {     private S primero;     private T segundo;      public Pareja(S primero, T segundo) {         this.primero = primero;         this.segundo = segundo;     }      public S getPrimero() { return primero; }      public T getSegundo() { return segundo; }      public void setPrimero(S primero) { this.primero = primero; }</pre>

<i><b>class Pareja</b></i>
<pre>public void setSegundo(T segundo) { this.segundo = segundo; }  }</pre>

Cuando se crean objetos de un tipo genérico, hay que proporcionar tipos concretos que determinen qué son realmente los tipos formales de la definición.

<i><b>uso de Pareja&lt;S, T&gt;</b></i>
<pre>Pareja&lt;String, String&gt; textos =     new Pareja&lt;String, String&gt;("hola", "adios"); System.out.println(textos.getPrimero()); System.out.println(textos.getSegundo());  Pareja&lt;String, Number&gt; tabla =     new Pareja&lt;String, Number&gt;("pi", 3.1416); System.out.println(tabla.getPrimero()); System.out.println(tabla.getSegundo());</pre>

### **limitaciones de las clases genéricas en java**

La introducción de clases genéricas en java es tardía y adolece de algunas importantes limitaciones derivadas de la necesidad de hacer compatible el código creado con anterioridad. La siguiente relación no pretende explicar por qué, sólo constata las limitaciones.

<i><b>problema</b></i>	<i><b>ejemplo</b></i>
No se pueden crear objetos de un tipo formal.	<code>new T();</code>
No se pueden crear <a href="#">arrays</a> de tipos formales.	<code>new T[100]</code>
No se pueden sobrecargar métodos usando tipos formales.	<pre>class X&lt;T, S&gt; {     void metodo(T t) { ... }     void metodo(S t) { ... } }</pre>
No se puede usar el operador <b>instanceof</b>	<code>if (x instanceof T)</code>
Los campos y métodos static (de clase) no pueden referirse a un tipo formal. (*)	<pre>static T t;  static T metodo(T t) { ... }</pre>
Los tipos enumerados no admiten tipos formales.	<code>enum Z&lt;T&gt; { A, B, C }</code>
Las excepciones no admiten tipos formales.	<code>class E&lt;T&gt; extends Exception { }</code>
Aunque T2 sea un subtipo de T1, <T2> no es un subtipo de <T1>.	

- (\*) Los tipos genéricos pueden tener campos y métodos **static**, con la única salvedad de que no se refieran al tipo formal de la clase. Si un método estático se desea que sea genérico, hay que recurrir a métodos genéricos.

El asunto de los [arrays](#) de tipo formal es especialmente insidioso: no se puede declarar un array de un tipo formal; hay que declararlo de tipo Object y aplicar [downcasting](#).

```
Object[] datos = new Object[...];
T x = (T) datos[...];
```

o

```
T[] datos = (T[])new Object[...];
T x = datos[...];
```

## métodos genéricos

Los métodos, individualmente, también pueden ser genéricos. Se dice que un método es genérico cuando sus argumentos y/o su resultado se refieren a un tipo formal que se deja como parámetro.

Los siguientes ejemplos muestran un uso típico:

<i><b>métodos genéricos</b></i>	
<pre>&lt;T&gt; T primero(T[] datos) {     for (T t: datos)         if (t != null)             return t;     return null; }</pre>	<pre>static &lt;T&gt; List&lt;T&gt; pareja(T t) {     List&lt;T&gt; resultado =         new ArrayList&lt;T&gt;();     resultado.add(t);     resultado.add(t);     return resultado; }</pre>
Dado un array de datos de tipo T, devuelve el primero que no es null.	Dado un dato de tipo T, crea una lista con dos elementos, que son dicho dato.
<pre>String[] datos =     { null, "uno", "dos" }; System.out.println(primero(datos));</pre>	<pre>List&lt;String&gt; bora_bora =     pareja("bora");</pre>

Para usar un método genérico basta llamarlo con parámetros que casen con su definición. El propio compilador infiere el tipo (concreto) que debe asignar a T para que funcione correctamente.

Al igual que las clases genéricas, los métodos genéricos funcionan bastante deficientemente cuando hay que crear [arrays](#).

## tipos formales acotados

Existiendo tipos y métodos genéricos, los métodos pueden recibir o devolver objetos de dichos tipos.

```
String ultima(List<String> lista) {
    return lista.get(lista.size() - 1);
}
```

donde el compilador puede chequear que llamamos al método con una lista de String.

Curiosamente, java no admite en estos casos las tradicionales relaciones de herencia, sino que es necesario ajustarse exactamente a lo dicho. En el ejemplo anterior, String significa única y exclusivamente String, no siendo aceptable subtipos de dicha clase.

NOTA: El uso tradicional de la herencia dice que donde se admite un tipo T es posible emplear cualquier subtipo de T.

A fin de generalizar, java introduce una notación específica para aliviar dicha rigidez:

**<T>**

acepta el tipo T y nada más que el tipo T

**<? extends T>**

acepta el tipo T o cualquier subtipo de T. En particular, **<?>** acepta cualquier tipo.

**<? super T>**

acepta el tipo T o cualquier supertipo de T.

Esta flexibilidad se puede usar allá donde hay tipos formales o argumentos formales:

**class X<? extends T> { ... }**

es un tipo genérico que sólo puede instanciarse con subtipos de T.

**void metodo(List<? extends T> lista) { ... }**

es un método que sólo puede llamarse con listas de objetos que son subtipos de T.

**void metodo(Pareja<?, ?> pareja)**

es un método que admite cualquier pareja de datos de cualquier tipo.

## 84. *getClass (método) public Class getClass()*

Método estándar disponible en todo los objetos. Devuelve un objeto de tipo *Class*, que permite saber cosas sobre la clase de objeto que estamos manejando. Aunque la clase *Class* proporciona muchísimos métodos, se recogen a continuación solamente algunos de los más habitualmente utilizados:

<i>Class getClass()</i>
<pre>package PET;  public class Ejemplo {     public static void main(String[] args) {         Ejemplo x = new Ejemplo();         Class tipo = x.getClass();         System.out.println("toString() = " +                            tipo.toString());         System.out.println("getSimpleName() = " +                            tipo.getSimpleName());         System.out.println("getName() = " +                            tipo.getName());         System.out.println("getCanonicalName() = " +                            tipo.getCanonicalName());         System.out.println("getPackage() = " +                            tipo.getPackage());         System.out.println("getSuperclass() = " +                            tipo.getSuperclass());     } }</pre>
<pre>toString() = class PET.Ejemplo</pre>



```
getSimpleName() = Ejemplo
getName() = PET.Ejemplo
getCanonicalName() = PET.Ejemplo
getPackage() = package PET
getSuperclass() = class java.lang.Object
```

## 85. Getter (concepto)

Ver "[métodos / métodos de acceso \(getters\)](#)".

## 86. hashCode (método) public int hashCode()

Método estándar disponible en todos los objetos.

Devuelve un número entero propio que caracteriza el objeto.

El método base proporcionado en java.lang.Object es heredado por todos los objetos, devolviendo un valor único, exclusivo e inmutable para cada objeto creado.

Las clases creadas por el programador pueden devolver otro valor con la única condición de que

- si dos objetos A y B son iguales según el método "[equals\(\)](#)", entonces ambos deben devolver el mismo hashCode

```
a.equals(b) => a.hashCode() == b.hashCode()
```

es decir, que el valor "hash" de un objeto no tiene porque ser ni único, ni exclusivo, ni inmutable.

El valor "hash" sirve para discriminar rápidamente si dos objetos son diferentes antes de llamar a equals():

```
if (a.hashCode() == b.hashCode() && a.equals(b))
```

Regla: si redefine equals(), debe redefinir hashCode().

Ver "[equals\(\)](#)".

### Patrón básico

---

Hay una amplísima libertad para definir una función que devuelva un entero usando como parámetros los mismos campos que se consideran para la determinación de equals().

```
hashCode = función(campo1, campo2, ...);
```

Es frecuente utilizar una función líneas que suma los diferentes campos involucrados combinados con números primos.

### Patrón: objetos iguales si son iguales algunos campos

---

A menudo la equivalencia no tiene más ciencia que decir que dos objetos son equivalentes si lo son algunos de sus campos.

Ejemplo, sean objetos que son coches

```
public class Coche {
    private String marca;
    private String modelo;
    private int puertas;
    private int km;
    private Color color;
```

Si hemos definido como iguales dos coches con igual marca, modelo y número de puertas:

```
@Override
public int hashCode() {
    int hc = 17;
    hc = 31 * hc + marca.hashCode();
    hc = 31 * hc + modelo.hashCode();
    hc = 31 * hc + puertas;
    return hc;
}
```

Si queremos tener en cuenta que alguno de los campos que son objeto pueda ser nulo:

```
@Override
public int hashCode() {
    int hc = 17;
    if (marca != null)
        hc = 31 * hc + marca.hashCode();
    if (modelo != null)
        hc = 31 * hc + modelo.hashCode();
    hc = 31 * hc + puertas;
    return hc;
}
```

### Ejemplo de otros tipos de equivalencia

---

La equivalencia puede depender de las propiedades algebraicas del tipo de objeto de que se trate. Por ejemplo:

<i><b>Fraccion.java</b></i>
<pre>public class Fraccion {     private int num;     private int den;      @Override     public boolean equals(Object x) {         if (x == this)             return true;         if (x == null)             return false;         if (x.getClass() != this.getClass())             return false;         Fraccion fraccion = (Fraccion)x;         return this.num * fraccion.den == this.den * fraccion.num;     }      @Override     public int hashCode() {         return num ^ den;     } }</pre>

### Si hashCode() está mal hecho ...

---

Si hashCode() devuelve valores diferentes para objetos que son equivalentes, según equals(), las clases del tipo HashMap y HashSet fallan.

Si hashCode() devuelve valores que, aún estando correctos, no discriminan entre objetos que no son equivalentes, las clases HashMap y HashSet se ralentizan.

Como ejemplo extremo,

```
int hashCode(){ return 0; }
```

es código correcto, pero extremadamente ineficiente.

Sea

```
public class Cadena {
    private String texto;

    private static Random random = new Random();

    public Cadena(String texto) {
        this.texto = texto;
    }

    @Override
    public boolean equals(Object x) {
        if (x == this) return true;
        if (x == null) return false;
        if (!(x instanceof Cadena)) return false;
        Cadena otro = (Cadena) x;
        return texto.equals(otro.texto);
    }

    @Override
    public int hashCode() {
        return texto.hashCode();
    }

    public static void main(String[] args) {
        Set<Cadena> set = new HashSet<Cadena>();
        long t0 = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            Cadena ejemplo = new Cadena(getAlgo());
            set.add(ejemplo);
            if (set.contains(ejemplo) == false)
                System.out.println("falla");
        }
        long t2 = System.currentTimeMillis();
        System.out.printf("%dms%n", t2 - t0);
    }

    private static String getAlgo() {
        return Long.toString(random.nextLong(), 36);
    }
}
```

Si ejecutamos, obtenemos un tiempo de ejecución de

46ms

Si cambiamos a

```
int hashCode(){ return 0; }
```

el tiempo de ejecución pasa a

Como moraleja podemos decir que si va a redefinir el método hashCode() de una clase, es buena idea hacer algunos experimentos de eficiencia para elegir una función que, siendo correcta, no penalice la ejecución. No hay recetas mágicas; pero los patrones indicados en las secciones anteriores suelen funcionar razonablemente bien.

## 87. Herencia [inheritance] (concepto)

Cuando

```
class B extends A { ... }
```

la nueva clase B dispone automáticamente de todos los miembros **public**, **protected** y “**de paquete**” de la clase A, miembros que puede usar o [redefinir](#).

Ver “[extensión](#)”.

B hereda de A

- los miembros (campos y métodos)

B puede [ocultar](#)

- variables de A
- métodos estáticos de A (de clase)

B puede [redefinir](#)

- métodos de A (de objeto)

B puede aportar lo que quiera si no colisiona con lo definido en A

### **ocultación versus redefinición**

---

No es lo mismo ocultar que redefinir.

#### Ocultación

La subclase puede ocultar

- variables (de clase y de objeto)
- métodos estáticos

a base de definir

- variables con el mismo nombre
- métodos con la misma [signatura](#)

Se puede acceder a los elementos de la superclase vía [upcasting](#), creando una variable de la clase que necesitamos

#### Redefinición (ver [polimorfismo](#))

La subclase puede redefinir

- métodos de objetos

a base de definir métodos con la misma signatura

- mismo nombre
- mismo número y tipo de argumentos
- igual o menos excepciones

- igual o más visible  
"de paquete" < protected < public
- igual resultado

Sólo hay una forma de acceder al método de la superclase:

[super](#).metodo(argumentos)

y sólo puede accederse a los métodos de la superclase inmediatamente superior a donde nos encontramos.

Intentaremos mostrar mediante un ejemplo la sutil diferencia entre ocultar y redefinir.

<i><b>class Madre</b></i>
<pre>public class Madre {      public <b>static</b> String getClass() { return "clase Madre"; }      public String getObjeto() { return "objeto de clase Madre"; } }</pre>
<i><b>class Hija</b></i>
<pre>public class Hija <b>extends</b> Madre {      // oculta     public <b>static</b> String getClass() { return "clase Hija"; }      @Override     public String getObjeto() { return "objeto de clase Hija"; } }</pre>
<i><b>class Madres_Hijas</b></i>
<pre>public class Madres_Hijas {     public static void main(String[] argumentos) {         Madre madre = new Madre();         System.out.println("madre.getClass()      -&gt; " + madre.getClass());         System.out.println("madre.getObjeto()     -&gt; " + madre.getObjeto());          Hija hija = new Hija();         System.out.println("hija.getClass()       -&gt; " + hija.getClass());         System.out.println("hija.getObjeto()      -&gt; " + hija.getObjeto());          <b>Madre confusa = new Hija();</b>                // <b>upcasting</b>         System.out.println("confusa.getClass()    -&gt; " + <b>confusa.getClass()</b>);         System.out.println("confusa.getObjeto()   -&gt; " + <b>confusa.getObjeto()</b>);     } }</pre>
<i><b>ejecución</b></i>
<pre>\$ java Madres_Hijas madre.getClass()      -&gt; clase Madre madre.getObjeto()     -&gt; objeto de clase Madre</pre>

```
hija.getClase()      -> clase Hija
hija.getObjeto()     -> objeto de clase Hija

confusa.getClase()   -> clase Madre
confusa.getObjeto()  -> objeto de clase Hija
```

Nótese que java usa una variable, **confusa**, definida como **Madre**, pero referenciando un objeto de clase **Hija**. El [upcasting](#) resuelve diferentemente el caso de los métodos ocultos y el de los redefinidos.

## constructores

---

Para inicializar un objeto B que hereda de A

```
class B extends A { ... }
```

hay que inicializar antes la parte heredada: los constructores de B empiezan llamando a los constructores de A

- Bien explícitamente: El programador escribe el código de la llamada  

```
super(argumentos del constructor de la superclase);
```
- Bien implícitamente: Si el programado no lo escribe: el compilador lo inyecta  

```
super();
```

Java exige que cuando una clase invoca al constructor de su superclase, la llamada a dicho super-constructor debe ser exactamente la primera cosa que haga el constructor de la subclase.

Ver "[super](#)".

## final

---

Una clase **final** no se puede [extender](#).

Un método **final** no se puede [redefinir](#).

En una clase **final**, todos los métodos son **final**.

## ¿cuándo usar herencia?

---

Si el código y la interfaz de una clase nos gustan podemos aprovecharlo (herencia) en otra clase

- añadiendo más cosas
- [ocultando](#) variables (a base de definir otra que tape a la primera)
- [redefiniendo](#) métodos ([polimorfismo](#))

El gran inconveniente de la herencia es que no se pueden eliminar métodos de la superclase. Es imposible que el compilador no vea todo lo que hace público la superclase. Si necesita que desaparezca algo, tiene que usar [composición](#) + [delegación](#) (o sea, meter dentro de la nueva clase B una referencia a la primera clase A, y poner métodos en B para acceder a lo que se necesite de A por delegación). La composición + delegación es extremadamente flexible, siendo su única limitación que no podemos hacer [upcasting](#) y, por tanto, no podemos disfrutar de las ventajas del polimorfismo.

La gran ventaja de la herencia es que la nueva clase es como la anterior pero ampliada y mejorada.

## 88. Identificadores [identifiers] (concepto)

Son los nombres de las cosas de los programas: clases, objetos, métodos, variables, etc.

Los identificadores deben empezar con una letra o un símbolo "\_". A continuación pueden venir cuantas letras, dígitos o símbolos "\_" se desee, sin dejar espacio entre ellos. Si un identificador está formado por varias palabras, estas se concatenan poniendo en mayúsculas la primera letra de cada palabra.

No se pueden usar como identificadores las palabras reservadas de java.

Ejemplos

- i, j, m, alfa, beta, Manuel, coche, Europa
- JoséAntonio
- añoBisiesto
- mil24
- año0, año1, año2, ... año10
- manual\_de\_protección\_de\_datos

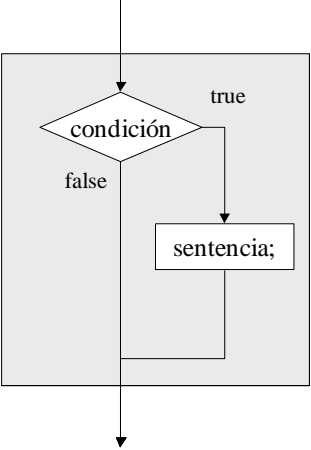
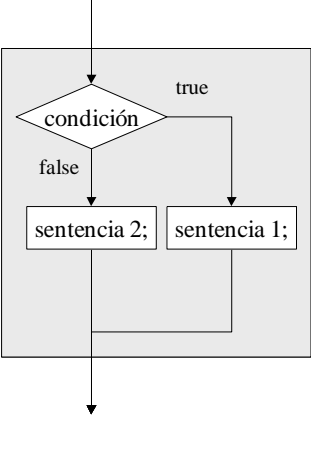
Por convenio, se recomienda que

- los nombres sean significativos
- la longitud: la necesaria para que el nombre no resulte críptico
- los nombres de las clases e interfaces empiecen con una letra mayúscula
- los nombres de los métodos empiecen con una letra minúscula
- los nombres de campos y variables empiecen con una letra minúscula
- los nombres de las constantes sean todas letras mayúsculas, usando el carácter '\_' si constaran de varias palabras

## 89. if (palabra reservada)

Ejecución condicional: algunas sentencias se ejecutan o no dependiendo de una condición [booleana](#):

<i>java</i>	<i>flujo</i>
-------------	--------------

<i>java</i>	<i>flujo</i>
<pre> if (condición)     sentencia;  if (condición) {     sentencia 1;     sentencia 2;     ...     sentencia ...; } </pre>	 <pre> graph TD     Entry(( )) --&gt; Cond{condición}     Cond -- true --&gt; Sent[<code>sentencia;</code>]     Sent --&gt; Exit(( ))     Cond -- false --&gt; Exit </pre>
<pre> if (condición)     sentencia 1; else     sentencia 2;  if (condición) {     sentencia 11;     sentencia 12;     ...     sentencia 1...; } else {     sentencia 21;     sentencia 22;     ...     sentencia 2...; } </pre>	 <pre> graph TD     Entry(( )) --&gt; Cond{condición}     Cond -- true --&gt; Sent1[<code>sentencia 1;</code>]     Cond -- false --&gt; Sent2[<code>sentencia 2;</code>]     Sent1 --&gt; Exit(( ))     Sent2 --&gt; Exit </pre>

### if encadenados

```

if (mes == 1)
    print("enero");
else if (mes == 2)
    print("febrero");
else if (mes == 3)
    print("marzo");
...
else
    print("no se");

```

### situaciones ambiguas

A veces no es evidente con qué "if" casa un "else":

```
if (a) if (b) S1; else S2;
```

java lo interpreta como

```
if (a) { if (b) S1; else S2; }
```



## errores típicos

<pre>if (condicion)     sentencia1;     sentencia2;     sentencia3;</pre>	El programador cree que las 3 sentencias dependen de la condición; pero, faltando las llaves, solo la primera sentencia está sujeta a la condición.
<pre>if (condicion) ;     sentencia;</pre>	Parece que la sentencia depende de la condición; pero el punto y coma tras la condición implica que hay una sentencia vacía. Efectivamente, la sentencia explícita no está sujeta a la condición.
<pre>if (a)     if (b)         S1; else     S2;</pre>	Parece que S2 se ejecutará siempre que la condición "a" falle; pero java entiende este código como <pre>if (a) {     if (b) {         S1;     } else {         S2;     } }</pre>

## 90. Igualdad (==)

Java define el operador de igualdad sobre todo tipo de datos

```
a == b
```

### Tipos primitivos

devuelve TRUE si los valores de a y b son idénticos.

OJO: La igualdad está muy bien definida para enteros, booleanos y caracteres; pero deja que desear en el caso de valores reales. Comparar dos reales es chequear que son idénticos hasta el último decimal, lo que puede dar resultados extraños si hay errores de redondeo. En el caso de números reales es mejor escribir código del tipo

```
Math.abs(a - b) < ERROR
```

donde ERROR especifica la precisión de la comparación.

### Objetos, incluidos arrays

devuelve TRUE si se refieren al mismo objeto.

Si interesa comparar el contenido de los objetos, conviene recurrir a un método específico que, por convenio, se suele denominar "[equals](#)".

En el siguiente ejemplo, dos cartas se consideran "equals" si son del mismo palo y del mismo valor.

<b><i>class Palo</i></b>
<pre>public enum Palo { Bastos, Copas, Espadas, Oros }</pre>

<i><b>class Carta</b></i>
<pre> public class Carta {     private Palo palo;     private int valor;      public Carta(Palo palo, int valor) {         this.palo = palo;         this.valor = valor;     }      @Override     public boolean equals(Object x) {         if (x == this) return true;         if (x == null) return true;         if (x.getClass() != this.getClass()) return false;         Carta carta= (Carta)x;         return this.palo == carta.palo &amp;&amp; this.valor == carta.valor;     }      @Override     public int hashCode() {         return palo.ordinal() ^ valor;     } } </pre>
<i><b>class TestCarta</b></i>
<pre> public class TestCarta {      public static void main(String[] args) {         Carta c1 = new Carta(Palo.Copas, 1);         Carta c2 = new Carta(Palo.Copas, 1);         Carta c3 = new Carta(Palo.Copas, 2);          System.out.println("c1 == c1 -&gt; " + (c1 == c1));         System.out.println("c1 == c2 -&gt; " + (c1 == c2));         System.out.println("c1.equals(c2) -&gt; " + c1.equals(c2));         System.out.println("c1 == c3 -&gt; " + (c1 == c3));         System.out.println("c1.equals(c3) -&gt; " + c1.equals(c3));     } } </pre>
<i><b>ejecución de las pruebas</b></i>
<pre> c1 == c1 -&gt; true c1 == c2 -&gt; false c1.equals(c2) -&gt; true c1 == c3 -&gt; false c1.equals(c3) -&gt; false </pre>

Ver “[equals](#)”.

## 91. Implementación (concepto)

Se dice que una clase implementa una interfaz cuando proporciona código para concreto para los métodos definidos en la interfaz.

<i>interface Coordenada.java</i>
<pre>public interface Coordenada {      double x();      double y();      double distancia(Coordenada q);  }</pre>
<i>class Punto implements Coordenada</i>
<pre>public class Punto     implements Coordenada {     private double x, y;      public Punto(double x, double y) {         this.x = x;         this.y = y;     }      public double x() {         return x;     }      public double y() {         return y;     }      public double distancia(Coordenada q) {         double dx = q.x() - x;         double dy = q.y() - y;         return Math.sqrt(dx * dx + dy * dy);     }  }</pre>

Se dice que los métodos en la clase concreta implementan a sus homónimos en la interface, debiendo cumplir las siguientes condiciones

- mismos argumentos (número, clase y orden)  
Misma clase implica que no vale recurrir a argumentos de tipos que sean superclase o subclase del prometido en la interface.
- misma clase de resultado (vale que la implementación devuelva una subclase)
- igual o mayor visibilidad  
"de paquete" < protected < public
- igual o menos capacidad de lanzar excepciones; el método que implementa puede
  - no lanzar excepción alguna

- lanzar excepciones que sean subclases de la del método implementado
- lanzar las mismas excepciones que el método implementado

De una misma interfaz pueden derivarse múltiples implementaciones.

<b>interface Funcion</b>	
<pre>public interface Funcion {     double y(double x)         throws Exception; }</pre>	
<b>class Seno</b>	<b>class Coseno</b>
<pre>class Seno     implements Funcion {     public double y(double x) {         return Math.sin(x);     } }</pre>	<pre>class Coseno     implements Funcion {     public double y(double x) {         return Math.cos(x);     } }</pre>
<b>class Tangente</b>	<b>class Suma</b>
<pre>class Tangente     implements Funcion {     public double y(double x)         throws Exception {         double seno =             Math.sin(x);         double coseno =             Math.cos(x);         if (Math.abs(coseno)             &lt; 1e-6)             throw new Exception();         return seno / coseno;     } }</pre>	<pre>class Suma     implements Funcion {     private final Funcion f1;     private final Funcion f2;      Suma(Funcion f1, Funcion f2) {         this.f1 = f1;         this.f2 = f2;     }      public double y(double x)         throws Exception {         return f1.y(x) + f2.y(x);     } }</pre>

Y se puede usar el tipo de la interfaz para referenciar a cualquier implementación

```
Funcion f1 = new Seno();
Funcion f2 = new Coseno();
Funcion f3 = new Suma(f1, f2);
```

### implementación múltiple

Una misma clase puede implementar varias interfaces.

<b>interface Trazo</b>	<b>interface Area</b>
<pre>public interface Trazo {     double longitud(); }</pre>	<pre>public interface Area {     double superficie(); }</pre>
<b>class Rectangulo implements Trazo, Area</b>	
<pre>public class Rectangulo</pre>	

<i>interface Trazo</i>	<i>interface Area</i>
<pre> <b>implements Trazo, Area</b> {     private double base, altura;      public Rectangulo(double base, double altura) {         this.base = base;         this.altura = altura;     }      public double <b>longitud</b>() {         return 2 * base + 2 * altura;     }      public double <b>superficie</b>() {         return base * altura;     } } </pre>	

### **implementación parcial**

Si una clase no implementa todos los métodos definidos en una interfaz, sino sólo parte de ellos, el resultado es una clase abstracta.

<i>interface Serie</i>
<pre> public interface Serie {     // término i-ésimo de la serie     public int termino(int i);      // suma de los n primeros términos     public int suma(int n); } </pre>
<i>abstract class SerieConSuma</i>
<pre> public abstract class SerieConSuma     implements Serie {     public int suma(int n) {         int suma = 0;         for (int i = 0; i &lt; n; i++)             suma += termino(i);         return suma;     } } </pre>

## **92. *implements* (palabra reservada)**

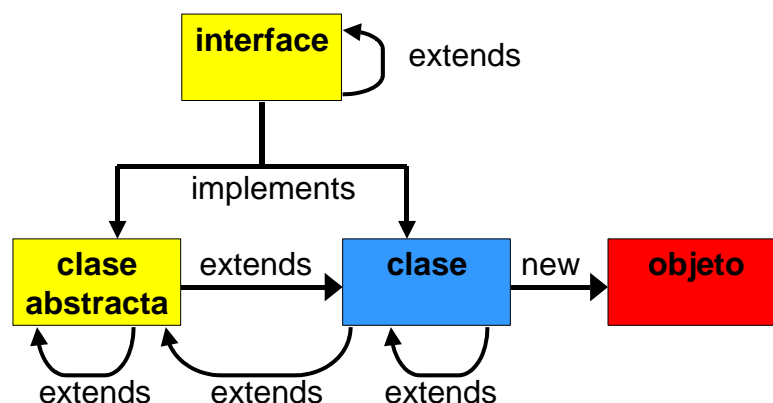
Sirve para indicar que una clase implementa una (o más) [interfaces](#).

En Java se pueden implementar varias interfaces simultáneamente:

```
class A implements I1, I2, ... { ... }
```

También se puede [extender](#) una clase (y sólo una) al tiempo que se implementan varias interfaces:

```
class B extends A implements I1, I2, ... { ... }
```



### 93. *import* (palabra reservada)

Permite referirse a las clases de un [paquete](#) sin explicitar su nombre canónico completo.

A continuación se muestra cómo el uso de "import" puede hacer menos farragoso el código de una clase:

<i>sin importación</i>
<pre> public class Ejemplo {     public static void main(String[] x) {         java.util.List&lt;String&gt; argumentos =             new java.util.ArrayList&lt;String&gt;();         for (String a : x)             argumentos.add(a);         java.util.Collections.reverse(argumentos);         for (String a2 : argumentos)             System.out.print(a2 + ", ");         System.out.println();     } } </pre>
<i>con importación de miembros individuales de un paquete</i>
<pre> import java.util.ArrayList; import java.util.Collections; import java.util.List;  public class Ejemplo {     public static void main(String[] x) {         List&lt;String&gt; argumentos = new ArrayList&lt;String&gt;();         for (String a : x)             argumentos.add(a);         Collections.reverse(argumentos);         for (String a2 : argumentos)             System.out.print(a2 + ", ");         System.out.println();     } } </pre>
<i>importación en bloque de todas las clases de un paquete</i>

```
import java.util.*;

public class Ejemplo {
    public static void main(String[] x) {
        List<String> argumentos = new ArrayList<String>();
        for (String a : x)
            argumentos.add(a);
        Collections.reverse(argumentos);
        for (String a2 : argumentos)
            System.out.print(a2 + ", ");
        System.out.println();
    }
}
```

También se pueden importar miembros estáticos de una clase:

#### ***sin importación***

```
public class Punto {
    private double x;
    private double y;

    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public static Punto polares(double modulo, double phy) {
        double x = modulo * Math.cos(phy);
        double y = modulo * Math.sin(phy);
        return new Punto(x, y);
    }

    public double getModulo() {
        return Math.sqrt((x * x) + (y * y));
    }

    public double getAngulo() {
        return Math.atan2(y, x);
    }

    public double getDistancia(Punto p) {
        double dx = p.x - this.x;
        double dy = p.y - this.y;

        return Math.sqrt((dx * dx) + (dy * dy));
    }
}
```

#### ***importando los miembros de la clase Math***

```
import static java.lang.Math.*;

public class Punto {
    private double x;
```

```

private double y;

public Punto(double x, double y) {
    this.x = x;
    this.y = y;
}

public static Punto polares(double modulo, double phy) {
    double x = modulo * cos(phy);
    double y = modulo * sin(phy);
    return new Punto(x, y);
}

public double getModulo() {
    return sqrt((x * x) + (y * y));
}

public double getAngulo() {
    return atan2(y, x);
}

public double getDistancia(Punto p) {
    double dx = p.x - this.x;
    double dy = p.y - this.y;

    return sqrt((dx * dx) + (dy * dy));
}
}

```

Ver "[paquetes](#)".

## 94. Inheritance (concepto)

Ver "[herencia](#)".

## 95. Inicialización (concepto)

Dícese de la asignación de un primer valor (valor inicial) a una [variable](#).

### campos (de clase o de objeto)

Pueden inicializarse explícitamente junto a la declaración o en el constructor. De lo contrario, reciben un valor por defecto que depende de su tipo:

- enteros: valor 0
- reales: valor 0.0
- booleanos: valor **false**
- caracteres: valor **(char)0**
- objetos: valor **null**

### variables locales

Deben ser inicializadas explícitamente por el programa, bien junto a la declaración, bien en alguna sentencia de asignación posterior; pero siempre antes de ser utilizadas. De lo contrario, el compilador emite un mensaje de error.



Las variables que recogen los argumentos de llamada a un método son inicializadas con los valores de llamada.

## 96. *instanceof* (palabra reservada)

Es un operador booleano que devuelve TRUE si el objeto de la izquierda es de la clase indicada a la derecha o de una subclase de ella.

<i>instanceof</i>	
<pre>class A { ... } class B extends A { ...}</pre>	
<pre>A a = new A(); B b = new B(); A ab = new B();</pre>	
<pre>a instanceof A a instanceof B b instanceof A b instanceof B ab instanceof A ab instanceof B</pre>	<pre>true false true true true true</pre>

**instanceof** también se puede emplear con [arrays](#)

**x instanceof int[]**

devuelve TRUE si se trata de un array de enteros.

**null instanceof X**

siempre devuelve TRUE.

## 97. *int* (palabra reservada)

<i>int</i> : números enteros	
<b>ocupación</b>	4 bytes = 32 bits
<b>mínimo</b>	Integer.MIN_VALUE = $-2^{31}$ = -2.147.483.648
<b>máximo</b>	Integer.MAX_VALUE = $2^{31} - 1$ = +2.147.483.647

Ver "[Números](#)".

## 98. *Interfaz de programación (concepto)*

Relación de clases, variables y métodos proporcionados por el suministrador del sistema de programación y que pueden ser empleados directamente por los programadores.

Por ejemplo, el paquete Math proporciona métodos para cálculos trigonométricos.

Ver

<http://java.sun.com/j2se/1.5.0/docs/api/>

## 99. *interface* (palabra reservada)

Es un elemento java que dice lo que se ofrece; pero no dice cómo se hace.

Una interface proporciona:

- valores constantes, que son variables “**public static final**” (no hace falta escribir estos modificadores: se sobreentienden)
- métodos, que son “**public**” (no hace falta escribir el modificador: se sobreentiende)
- NO incluye constructores

Las interfaces

- pueden extenderse con nuevas constantes y/o métodos
- pueden implementarse totalmente dando lugar a una clase que debe proporcionar código para todos y cada uno de los métodos definidos
- pueden implementarse parcialmente dando lugar a una [clase abstracta](#)

<i>ejemplos</i>
<pre>interface A {     /* public static final */ int CONSTANTE = 99;     /* public */ void metodoA(); }</pre>
<pre>interface B extends A {     /* public */ void metodoB(); }</pre>
<pre>class M implements A {     public void metodoA() { código } }</pre>
<pre>abstract class N implements B {     public void metodoA() { código } }</pre>

### ¿cuándo usar interfaces?

Cuando sabemos qué queremos; pero

- no sabemos (aún) cómo hacerlo
- lo hará otro
- lo haremos de varias maneras

Las interfaces son muy útiles para hacer [upcasting](#) (decir que una variable es de la clase interface y asignarle referencias a valores de clases que la implementan).

## 100. *Interfases* (concepto)

Ver palabras reservadas “[interface](#)” e “[implements](#)”.

## 101. Interpretación (concepto)

Es lo que hace la máquina virtual de java: lee código interpretable y ordena qué debe hacer la máquina real (por ejemplo, le indica al pentium qué debe ir haciendo para que ocurra lo que ha escrito el programador).

Ver "[ejecución](#)".

## 102. jar

Acrónimo: java archive. Formato de ficheros que almacenan múltiples ficheros, comprimidos.

Se emplea rutinariamente para convertir un grupo de ficheros *class* en un único fichero *jar*. En esta agrupación se puede incluir un fichero MANIFEST que indique por donde empezar, resultando en un fichero capaz de ejecutarse con un "*doble clic*".

Ver "[ejecución](#)".

Ver "[jar files](#)".

Se denomina de igual modo una herramienta estándar que permite comprimir varios ficheros y/o directorios en un único fichero. El formato es compatible con ZIP.

Ver <http://java.sun.com/j2se/1.5.0/docs/tooldocs/index.html#basic>

## 103. java (herramienta estándar)

Intérprete de código interpretable (ficheros *.class*). Se usa en consolas.

Ver "[ejecuciónEjecución \[execution\] \(concepto\)](#)".

Ver "[jar tool](#)".

## 104. javac (herramienta estándar)

Compilador. Lee código fuente (ficheros *.java*) y genera código interpretable (ficheros *.class*).

Sea la clase

<i>Hola.java</i>
<pre>public class Hola {     public static void main(String[] argumentos) {         System.out.println("Hola.");     } }</pre>

Si el fichero "Hola.java" está en el directorio X, nos situaremos en dicho directorio y llamaremos al compilador:

```
$ javac Hola.java
```

Si hay que compilar varios ficheros *.java*, indíquelos uno tras otro separados los nombres por espacio en blanco. El resultado es un fichero "Hola.class" en el mismo directorio.

Si deseamos separar los ficheros *.java* de los ficheros *.class*

```
$ javac -d Y Hola.java
```

El resultado es un fichero "Hola.class" en el directorio Y.

## con paquetes (o directorios)

---

Si empleamos paquetes, hay que alinear los nombres del paquete y los directorios. Sea

<i>Hola2.java</i>
<pre>package ejemplo.ejecucion;  public class Hola2 {     public static void main(String[] argumentos) {         System.out.println("Hola.");     } }</pre>

Una vez colocado el fichero “Hola2.java” en su directorio

```
X/ejemplo/ejecucion/Hola2.java
```

nos situaremos en el directorio X y llamaremos al compilador

```
$ javac ejemplo.ejecucion.Hola2.java
```

El resultado es el fichero “X/ejemplo/ejecucion/Hola.class”.

Si deseamos separar ficheros .java de ficheros .class, se lo indicaremos al compilador

```
$ javac -d Y ejemplo.ejecucion.Hola2.java
```

El resultado es el fichero “Y/ejemplo/ejecucion/Hola.class”.

## incorporando datos de otro proyecto

---

Si en la compilación necesitamos código de otro proyecto java que se encuentra en el directorio Z (y los directorios que correspondan a su estructura de paquetes), se lo indicaremos al compilador

```
$javac -classpath Z ... etc ...
```

Si el otro proyecto está compactado en un fichero Z.jar, se lo indicaremos al compilador

```
$ javac -classpath Z.jar ... etc ...
```

Si hay que recurrir a varios otros proyectos Z1, Z2.jar, ..., los indicaremos todos ellos

```
$javac -classpath Z1:Z2.jar:... ... etc ...      (en Unix)
$javac -classpath Z1;Z2.jar;... ... etc ...      (en Windows)
```

En lugar de escribir “-classpath” cada vez que se llama al compilador, se puede definir una variable de entorno CLASSPATH. El efecto es equivalente.

Ver <http://java.sun.com/j2se/1.5.0/docs/tooldocs/index.html#basic>

## 105. javadoc (herramienta estándar)

Herramienta que permite generar páginas HTML de documentación de programas. Para ello utiliza los comentarios de documentación insertados en el texto.

Ver “[documentación](#)”.

Ver <http://java.sun.com/j2se/javadoc/>

## 106. JDK (acrónimo)

Java Development Kit. Entorno de desarrollo java.

Paquete software que contiene herramientas para desarrollar y ejecutar programas java.

Incluye:

compilador: [javac](#)

intérprete ([runtime](#)): java

### **107. JRE (acrónimo)**

Java Runtime Environment. Entorno de ejecución java.

Paquete software que contiene herramientas para ejecutar programas java, previamente compilados. Incluye un intérprete ([java](#)) y bibliotecas de apoyo.

### **108. JVM (acrónimo)**

Java Virtual Machine. Máquina virtual de java.

También llamada “[intérprete de java](#)”. Ver “[java \(herramienta estándar\)](#)”.

Lee código interpretable ([bytecode](#)) y lo ejecuta.

### **109. Keywords (palabras reservadas)**

Ver “[palabras reservadas](#)”.

### **110. Listas (estructura de datos)**

Secuencias de datos en las que importa el orden relativo de los mismos.

Ver “[List](#)”.

Ver “[listas encadenadas](#)”.

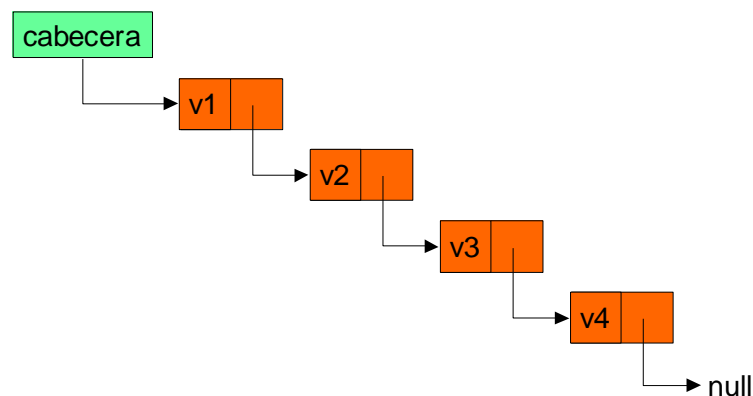
#### **¿listas o arrays?**

---

Ver “[arrays - ¿arrays o listas?](#)”

### **111. Listas encadenadas [linked lists] (estructura de datos)**

Aquellas en las que cada elemento de la lista referencia al siguiente.



La principal ventaja de las listas encadenadas es que ocupan tanto espacio como nodos tienen en cada momento (a diferencia de los [arrays](#) que ocupan tanto espacio como elementos pueden llegar a contener). Además, es muy fácil insertar o eliminar nodos en medio de la lista.

El principal inconveniente de las listas encadenadas es que hay que tener una referencia por nodo, ocupando más espacio.

## **nodos**

Los elementos de una lista suelen nodos con este aspecto

<b><i>class Nodo</i></b>
<pre> public class Nodo {     private Object valor;     private Nodo siguiente;      public Nodo(Object valor, Nodo siguiente) {         this.valor = valor;         this.siguiente = siguiente;     }      public Object getValor() { return valor; }      public void setValor(Object valor) { this.valor = valor; }      public Nodo getSiguiente() { return siguiente; }      public void setSiguiente(Nodo siguiente) {         this.siguiente = siguiente;     } } </pre>

## **algoritmos sobre listas**

Es habitual recorrer las listas en orden

<b><i>algoritmos iterativos</i></b>	<b><i>algoritmos recursivos</i></b>
<pre> Nodo nodo = cabecera; while (nodo != null) {     <i>haz algo con nodo</i>     nodo = nodo.getSiguiente(); } </pre>	<pre> haz(cabecera);  ... haz(Nodo nodo) {     if (nodo != null) {         <i>haz algo con nodo</i>         haz(nodo.getSiguiente());     } } </pre>
<pre> for (Nodo nodo = cabecera;      nodo != null;      nodo = nodo.getSiguiente()) {     <i>haz algo con nodo</i> } </pre>	<pre> haz(null, cabecera);  ... haz(Nodo anterior, Nodo nodo) { </pre>

<b>algoritmos iterativos</b>	<b>algoritmos recursivos</b>
<pre>Nodo anterior= null; for (Nodo nodo= cabecera;     nodo != null;     nodo = nodo.getSiguiente()) {     haz algo con anterior y nodo }</pre>	<pre>if (nodo != null) {     haz algo con anterior y nodo     haz(nodo,         nodo.getSiguiente()); }</pre>

### **listas doblemente encadenadas**

Son aquellas en las que los nodos referencian tanto al nodo siguiente como al anterior.

En estas listas es muy fácil avanzar o retroceder desde cualquier nodo.

### **listas circulares**

Son aquellas en las que el último nodo referencia de nuevo al primero de la lista.

## **112. long (palabra reservada)**

<b>long : números enteros</b>	
<b>ocupación</b>	8 bytes = 64 bits
<b>mínimo</b>	Long.MIN_VALUE = $-2^{63}$ = -9.223.372.036.854.775.808
<b>máximo</b>	Long.MAX_VALUE = $2^{63} - 1$ = +9.223.372.036.854.775.807

Ver "[Números](#)".

## **113. main (método) public static void main(String[])**

Si una clase es pública y tiene un método **main**, podemos lanzar la ejecución del método usando la herramienta "java".

Ver "[java](#)".

<b>Ejemplo.java</b>
<pre>public class Ejemplo {     public static void main(String[] argumentos) {         System.out.println("Hola. Soy yo.");     } }</pre>

El método

- debe ser **public** para ser accesible desde fuera
- debe ser **static** (de clase) porque aún no hemos creado ningún objeto
- es **void** por convenio: no devuelve nada
- se llama **main** por convenio

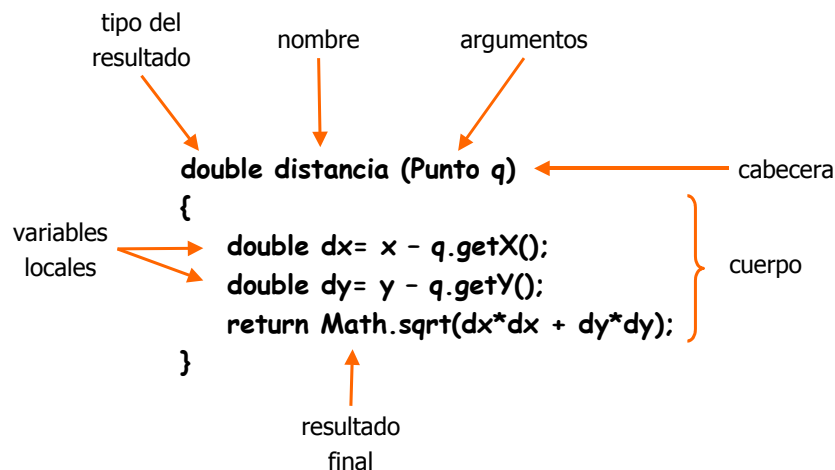
- el argumento es un [array](#) de cadenas de texto, [String](#), que se cargan con los datos que le pasemos al intérprete

## 114. Máquina virtual java (concepto)

Ver "[JVM](#) Java Virtual Machine".

## 115. Método [method] (concepto)

Un método es un poco de código con una misión. Típicamente utiliza una serie de argumentos de entrada para producir un resultado.



En una misma clase no pueden convivir dos métodos que tengan igual nombre e igual lista de argumentos de entrada. Simplemente porque java no sabría distinguir a qué método nos referimos en cada momento.

Cuando dos métodos se llaman igual pero difieren en el número, orden o tipo de los argumentos de entrada, se dice que el nombre del método está sobrecargado (en inglés, "[overloaded](#)"). Esto se puede hacer en java.

```

int suma (int a, int b) {
    return a + b;
}

char suma (char c, int n) {
    return c + n;
}
  
```

### variables

Un método accede a diferentes tipos de variables:

- los [campos](#) de la clase a la que pertenece, con el valor que tengan; existían antes y permanecen después de la terminación del método
- los [argumentos formales](#), con el valor dado por los datos de llamada al método; se crean con la llamada al método y desaparecen con su terminación
- las [variables locales](#) que defina para su propio uso; se crean cuando se declaran y desaparecen con la terminación del método



## signatura

---

Se llama signatura de un método al siguiente conjunto de información

- el nombre del método
- la serie de tipos de los argumentos de entrada

Así, la signatura del método "suma" del párrafo anterior es

`< suma, < int, int > >`

En una clase no pueden coexistir dos métodos con idéntica signatura: serían causa de ambigüedad.

## excepciones

---

Los métodos pueden declarar en la cabecera las excepciones que pueden lanzarse durante su ejecución, bien porque las lanza el propio método, bien porque las lanza algún otro método llamado desde este.

El que un método pueda lanzar una cierta excepción, no implica que la lance siempre; simplemente puede lanzarse o no.

La cabecera puede declarar las excepciones. Hay varios casos a considerar:

### excepciones no chequeadas

son las derivadas de [Error](#) o de [RuntimeException](#)

pueden lanzarse desde el método aunque no estén declaradas en la cabecera

aunque su declaración es opcional, conviene declararlas cuando son excepciones que lanza el programador, para que queden mejor documentadas

### excepciones chequeadas

las que no entran en el caso anterior; normalmente las derivadas de [Exception](#)

el método no puede lanzarlas si no están declaradas en la cabecera

su declaración es obligatoria si el método pretende lanzarlas

Ver "[Exception](#)" y "[Excepciones y métodos](#)".

## constructores

---

Se dice de aquellos métodos que sirven para crear un objeto. Se llaman igual que la clase del objeto que crean.

<b>Se recomienda que los constructores inicialicen todos los campos del objeto.</b>
---

```
class Circulo {  
    private double radio;  
  
    Circulo (double radio) { this.radio = radio; }  
}
```

En una misma clase puede haber varios constructores que deben diferir en el número o tipo de argumentos ([sobrecarga](#)).

Un constructor puede llamar a otro constructor:

```

class Rectangulo {
    private double ancho, alto;

    Rectangulo(double alto, double ancho) {
        this.alto= alto;
        this.ancho= ancho;
    }

    // construye un cuadrado
    Rectangulo(double lado) {
        this(lado, lado);
    }

    // construye un cuadrado de lado 1
    Rectangulo() {
        this(1);
    }
}

```

### **métodos de acceso (getters)**

---

Se dice de aquellos métodos que devuelven el valor de un [campo](#) del objeto.

Por convenio, se escriben comenzando con las letras "get" seguidas del nombre del campo. El tipo de retorno es el tipo del campo al que se accede.

```

public class Circulo {
    private double radio;

    public double getRadio() { return radio; }
}

```

Si el campo es booleano, se escriben con las letras "is" seguidas del nombre del campo.

```

public class Bombilla {
    private boolean encendida;

    public boolean isEncendida() { return encendida; }
}

```

### **métodos de carga (setters)**

---

Se dice de aquellos métodos que cargan el valor de un [campo](#) del objeto.

Por convenio, se escriben comenzando con las letras "set" seguidas del nombre del campo.

```

public class Circulo {
    private double radio;

    public void setRadio(double radio) { this.radio = radio; }
}

```

Si el campo es booleano, se puede optar por el mismo convenio:

```

public class Bombilla {
    private boolean encendida;
}

```

```

    public void setEncendida(boolean encendida) {
        this.encendida = encendida;
    }
}

```

Este planteamiento lleva a programas que dicen cosas así:

```

Bombilla bombilla = new Bombilla();
bombilla.setEncendida(true);    // para encender
bombilla.setEncendida(false);   // para apagar

```

Por ello, a veces es mejor usar nombres de métodos que sean significativos sin recurrir a un argumento booleano:

```

public class Bombilla {
    private boolean encendida;

    public void setEncendida() { this.encendida = true; }
    public void setApagada() { this.encendida = false; }
}

```

Y los programas quedan así:

```

Bombilla bombilla = new Bombilla();
bombilla.setEncendida();    // para encender
bombilla.setApagada();      // para apagar

```

### **paso de argumentos por valor (paso del valor)**

Dícese cuando al método se le pasa una COPIA del valor del dato. Al ser una copia, el dato original no se ve alterado si por alguna razón el método alterara su copia.

En java los tipos [primitivos](#) se pasan siempre por valor.

<i><b>paso por valor</b></i>	
void a(int n) {	// a recibe un valor en el argumento formal n
n = n+1;	// a modifica su copia
}	
void b() {	
int n = 0;	// n vale 0
a(n);	// pasamos en valor 0 como argumento real
System.out.println(n);	// n sigue valiendo 0
}	

### **paso de argumentos por referencia (paso de la referencia)**

Dícese cuando al método se le pasa una referencia al dato. Aunque el método no puede alterar la referencia propiamente dicha, sí puede alterar aquello a que se refiere la referencia.

Se copia la referencia; pero se comparte el objeto.

Java pasa por valor todo aquello que no sean tipos [primitivos](#): [arrays](#) y [objetos](#).

### ***modificación de un objeto pasado***

```
void a(int[] datos) {           // a recibe la referencia a un array
    datos[0] = 1;               // a modifica el array compartido
}
void b() {
    int[] datos = new int[2];
    datos[0] = 0;               // la primera posición contiene 0
    a(datos);                   // pasamos una referencia al array
    System.out.println(datos[0]); // la primera posición contiene 1
}
```

### ***copia local***

```
void a(int[] datos) {           // a recibe la referencia a un array
    datos = new int[2];         // a fabrica su propia copia
    datos[0] = 1;               // a modifica su array
}
void b() {
    int[] datos = new int[2];
    datos[0] = 0;               // la primera posición contiene 0
    a(datos);                   // pasamos a a una referencia al array
    System.out.println(datos[0]); // la primera posición contiene 0
}
```

## **valor devuelto (resultado)**

---

La declaración de un método indica el tipo del resultado que devuelve el método.

Si el método no devuelve nada, se dice "[void](#)".

En Java los métodos sólo pueden devolver un único valor.

Si necesitara devolver varios valores, hay múltiples opciones:

- use como retorno un tipo estructurado; es decir, una clase con varios campos para los diferentes valores del resultado.

El método crea un objeto resultado con los datos pertinentes.

Es la solución más elegante; pero pudiera ser costosa en ejecución.

- pase como argumento un tipo estructurado; es decir una clase con varios campos para los diferentes valores del resultado.

El método carga los resultados en al objeto pasado.

Es menos elegante; pero más eficaz en ejecución al evitar la creación de nuevos objetos.

- use los [campos](#) de la clase a la que pertenece el método.

El método carga en ellas los resultados.

Aunque depende mucho de cada caso, puede dar lugar a programas ininteligibles, debido a que se modifican campos inesperados. **Es (muy) poco recomendable.**

A esta forma de programar se la conoce como "por efecto colateral" (*side effect programming*).

$$a x^2 + b x + c = 0$$

```
public class EcuacionGrado2 {
    private double a, b, c;

    public EcuacionGrado2(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // resultado estructurado
    public Solucion raices() {
        double discriminante = b * b - 4 * a * c;
        if (discriminante >= 0.0) {
            double x1 = (-b + Math.sqrt(discriminante)) / (2 * a);
            double x2 = (-b - Math.sqrt(discriminante)) / (2 * a);
            return new Solucion(true, x1, x2);
        } else {
            double real = (-b + Math.sqrt(-discriminante)) / (2 * a);
            double imag = (-b - Math.sqrt(-discriminante)) / (2 * a);
            return new Solucion(false, real, imag);
        }
    }

    // modificación del argumento de llamada
    public void raices(Solucion solucion) {
        double discriminante = b * b - 4 * a * c;
        if (discriminante >= 0.0) {
            double x1 = (-b + Math.sqrt(discriminante)) / (2 * a);
            double x2 = (-b - Math.sqrt(discriminante)) / (2 * a);
            solucion.real = true;
            solucion.valor1 = x1;
            solucion.valor2 = x2;
        } else {
            double real = (-b + Math.sqrt(-discriminante)) / (2 * a);
            double imag = (-b - Math.sqrt(-discriminante)) / (2 * a);
            solucion.real = false;
            solucion.valor1 = real;
            solucion.valor2 = imag;
        }
    }

    public class Solucion {
        boolean real;
        double valor1;
        double valor2;

        public Solucion(boolean real, double valor1, double valor2) {
            this.real = real;
            this.valor1 = valor1;
            this.valor2 = valor2;
        }
    }
}
```

## número variable de argumentos (varargs)

---

Java permite pasar un número indefinido (0 o más) de argumentos a un método, con algunas limitaciones:

- sólo el último argumento puede ser de longitud indefinida
- todos los valores deben ser del mismo tipo, incluyendo [promoción](#), [upcasting](#) y [autoboxing](#).

```
public int maximo(int ... valores) {  
    int maximo = Integer.MIN_VALUE;  
    for (int n : valores)  
        if (n > maximo)  
            maximo = n;  
    return maximo;  
}
```

```
max(5, -7, 21) → 21  
max() → MIN_VALUE
```

Nótese que el número de argumentos puede ser cero (0).

Técnicamente, java trata estos argumentos como un [array](#) de datos del tipo indicado:

```
public boolean ordenados(String ... piezas) {  
    if (piezas.length < 2)  
        return true;  
    for (int i = 1; i < piezas.length; i++) {  
        String s1 = piezas[i - 1];  
        String s2 = piezas[i];  
        if (s1.compareTo(s2) > 0)  
            return false;  
    }  
    return true;  
}
```

```
ordenados("alfa", "beta", "gamma", "delta") → false  
ordenados("alfa", "beta", "delta", "gamma") → true
```

## métodos recursivos

---

Se dice que un método es recursivo cuando se llama a sí mismo

```
public int factorial(int n) {  
    if (n < 1)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

```
factorial( 5 ) = 5 * factorial( 4 )  
               = 5 * ( 4 * factorial( 3 ) )  
               = 5 * ( 4 * ( 3 * factorial( 2 ) ) )  
               = 5 * ( 4 * ( 3 * ( 2 * factorial( 1 ) ) ) )
```

```
public int factorial(int n) {
    if (n < 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
= 5 * ( 4 * ( 3 * ( 2 * ( 1 * factorial( 0 ) ) ) ) )
= 5 * ( 4 * ( 3 * ( 2 * ( 1 * 1 ) ) ) )
= 5 * 4 * 3 * 2 * 1 * 1
= 120
```

Este tipo de programas funciona porque cada método tiene sus propias variables locales (argumentos). En cada llamada se crea una variable propia y diferente de otras.

Los métodos recursivos se prestan a errores de programación si el creador no se asegura de que en ejecución el número de llamadas termina alguna vez. El código anterior podría estar mal en pequeños detalles que darían lugar a una recursión sin fin:

```
public int factorial(int n) {
    if (n < 1)
        return 1;
    else
        // modificamos la copia local, pasando el mismo n
        return n * factorial(n--);
}
```

```
public int factorial(int n) {
    if (n == 0) // falla si llamamos con n < 0
        return 1;
    else
        return n * factorial(n-1);
}
```

Es prácticamente obligatorio que los métodos recursivos incluyan código condicional. De hecho, en su diseño hay que tener en cuenta dos criterios:

1. en cada llamada recursiva debemos acercarnos a la solución: convergencia
2. debe haber un caso “cero”: terminación

Los métodos recursivos reflejan en programación la técnica de demostración por inducción de las matemáticas.

Los programas recursivos tienen cierta fama, no siempre merecida, de lentos. Aunque conviene medir tiempos antes de opinar, es cierto que en ocasiones la solución recursiva es muy elegante pero discutiblemente eficiente.

Existe una forma de plantear métodos recursivos que frecuentemente ayuda a mejorar el tiempo de ejecución. Se conoce como “*tail recursion*” y consiste en evitar que un método tenga que hacer algo tras conocer el resultado de la llamada recursiva; es decir, que lo último que haga un método sea llamarse recursivamente. El método anterior usando esta técnica quedaría así

```

public int factorial(int n) {
    return factorial2(n, 1);
}

private int factorial2(int n, int resultado) {
    if (n < 1)
        return resultado;
    else
        return factorial2(n-1, resultado*n);
}

```

No obstante, si es necesario evitar la recursión, siempre puede pasarse a una estructura de bucles que refleje el mismo razonamiento recursivo:

```

public int factorial(int n) {
    int resultado = 1;
    while (n >= 1) {
        resultado*= n;
        n-= 1;
    }
    return resultado;
}

```

## 116. Miembro [member] (concepto)

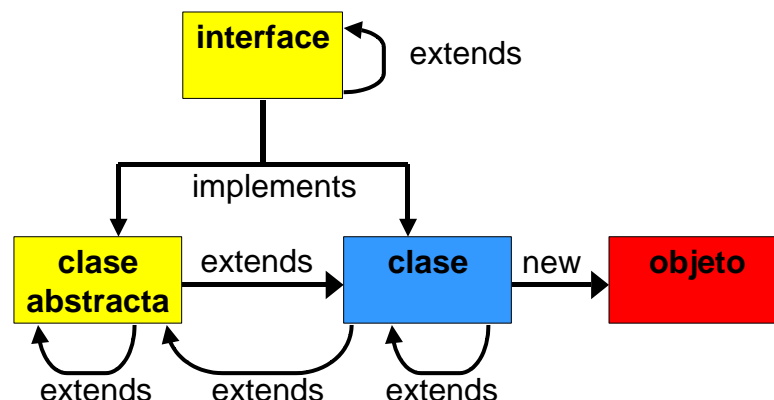
Una clase está formada por miembros:

- [campos](#) de clase ([static](#)): variables de la clase, compartidas por todos los objetos
- [campos](#) de objeto: variables de cada objeto
- [métodos](#): operaciones a realizar
- otras clases

## 117. new (palabra reservada)

Se usa para construir un [objeto](#) de una cierta [clase](#). Lo que se hace es ejecutar el [constructor](#), inicializando los [campos](#) del objeto según se crea.

No se puede usar con clases [abstractas](#), ni con [interfaces](#).





## 118. null (palabra reservada)

Es una referencia (a un [objeto](#)) especial.

Cuando una variable no apunta a ningún objeto, se dice que contiene la referencia “**null**” que es la referencia que no apunta a nada.

Cuando un campo que referencia a objetos se declara pero aún no se ha inicializado, contiene la referencia “**null**”.

## 119. Números (concepto)

Java maneja directamente números enteros y reales (de coma flotante; en inglés *floating point*).

### enteros

Java maneja diferentes rangos de valores: [byte](#), [short](#), [int](#) y [long](#). Lo normal, mientras no se indique lo contrario, es trabajar con enteros *int*.

### Notación

En código fuente los enteros se representan usando una serie de caracteres y una base de representación. Normalmente se usan números decimales (base 10) que emplean los caracteres '0', '1', '2', '3', '4', '5', '6', '7', '8' y '9'. El valor de un número se calcula multiplicando cada carácter por la potencia de 10 correspondiente a su posición.

Ejemplo:

$$2005 \rightarrow 2 \cdot 10^3 + 0 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0$$

En general:

$$c_n c_{n-1} \dots c_1 c_0 \rightarrow c_n 10^n + c_{n-1} 10^{n-1} + \dots + c_1 10^1 + c_0 10^0$$

Esta notación se puede generalizar para trabajar en cualquier base, siendo frecuentes los números en notación binaria ( $b = 2$ ), octal ( $b = 8$ ) y hexadecimal ( $b = 16$ ):

$$c_n c_{n-1} \dots c_1 c_0 \rightarrow c_n b^n + c_{n-1} b^{n-1} + \dots + c_1 b^1 + c_0 b^0$$

En notación binaria,  $b = 2$ , y se usan los dígitos '0' y '1'.

En notación octal,  $b = 8$ , y se usan los dígitos del '0' al '7'.

En notación hexadecimal,  $b = 16$ , y se usan los dígitos del '0' al '9' y las letras de la 'a' a la 'f'. De forma que 'a' (o 'A') vale 10; 'b' vale 11 y así sucesivamente hasta la 'f' que vale '15'. A veces se usan letras minúsculas, a veces mayúsculas: es indiferente.

El siguiente código permite imprimir un valor numérico en diferentes notaciones:

<b>void bases(int n)</b>	
<pre>public void bases(int n) {     System.out.println("decimal:      " + n);     System.out.println("binario:      " + Integer.toBinaryString(n));     System.out.println("octal:      " + Integer.toOctalString(n));     System.out.println("hexadecimal: " + Integer.toHexString(n));     System.out.println("base 3:      " + Integer.toString(n, 3)); }</pre>	
<b>bases(2001)</b>	
decimal:	2001
binario:	11111010001

### ***void bases(int n)***

```
octal:      3721
hexadecimal: 7d1
base 3:     2202010
```

Java permite escribir en el código fuente números:

- en notación decimal: como estamos acostumbrados (ej. 2001)
- en notación octal: empezando por '0' (ej. 03721)
- en notación hexadecimal: empezando por '0x' (ej. 0x7d1)

Si no se dice nada, el valor se interpreta con formato “**int**”. Si se quiere forzar la representación:

- para que el valor sea “**long**”, debe terminarse con la letra “l” o “L”:
  - ejemplo: 1L, que es la unidad en representación “**long**”

### **Conversión de valores en String**

Frecuentemente tendremos los datos metidos en una String (por ejemplo porque los hemos leído de un fichero de datos). La conversión a enteros de java se realiza de la siguiente manera:

```
int n = Integer.parseInt(cadena);      // notación decimal, b= 10
int n = Integer.parseInt(cadena, b);   // base b
```

### **Valores fuera de rango**

¿Qué pasa si nos salimos de rango? NADA BUENO.

Cada tipo de enteros, [byte](#), [short](#), [int](#) y [long](#), se caracterizan por un valor mínimo (el entero más negativo que se puede representar) y un valor máximo (el mayor entero positivo que se puede representar). Si una operación aritmética nos lleva fuera de ese rango el resultado es difícilmente predecible.

Por ejemplo, los *int* deben estar en el rango [-2.147.483.648, +2.147.483.647]. Pero ...

```
2147483647 + 1 = -2147483648
-2147483648 - 1 = 2147483647
1000000000 * 10 = 1410065408
```

Aunque el valor calculado está evidentemente mal, java no avisa de forma alguna.

Ver "[desbordamiento](#)".

### **¿Qué tipo de enteros debo usar?**

Java tiene un especial cariño por los "[int](#)". Todos los números java los interpreta como "int", salvo que se diga lo contrario.

Lo más normal es que todos los enteros sean de tipo "int"; salvo que haya una necesidad evidente de recurrir a otro de los tipos:

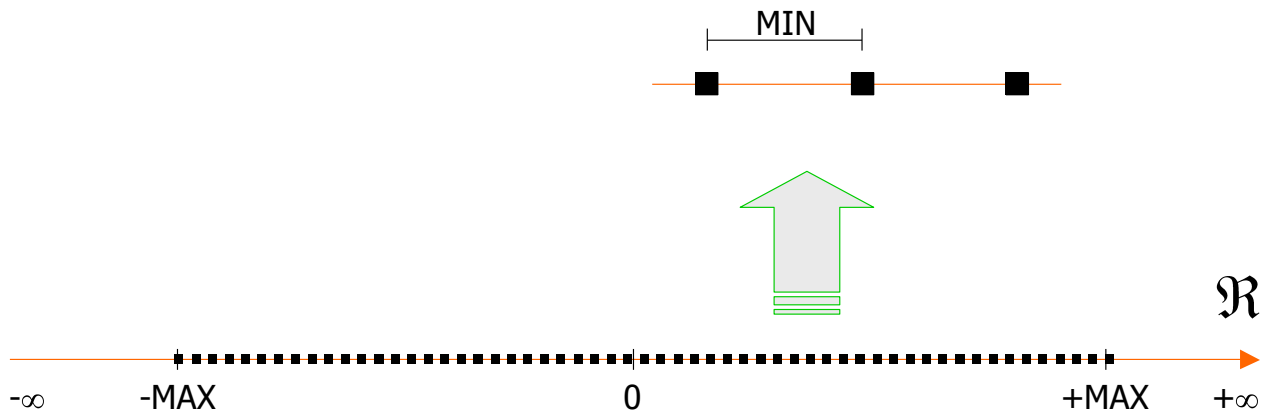
- byte o short, porque tenemos muchísimos enteros con valores muy pequeños
- long, porque tenemos enteros con valores muy grandes

## reales

Java maneja diferentes rangos de valores: *float*, y *double*. Lo normal, mientras no se indique lo contrario, es trabajar con reales *double*.

Los números reales se caracterizan por

- un valor máximo, que es el máximo valor absoluto que se puede representar
- un valor mínimo, que es la mínima diferencia entre dos números reales (precisión)



## Notación

Un valor real se indica por medio de

- un signo; si no se indica, se presume que el valor es positivo
- una mantisa; que es una serie de dígitos decimales, posiblemente con un punto para indicar la posición de la coma decimal
- un exponente, que es opcional (si no se indica se presume que el exponente es 1); si aparece viene precedido de la letra 'E' y será un valor entero en notación decimal

Ejemplos:

- 3.1416, pi
- 2.5E3, dos mil quinientos
- -1E-12, una millonésima

Si no se dice nada, el valor se interpreta con formato "**double**". Si se quiere forzar la representación:

- para que el valor sea "**float**", debe terminarse con la letra "f" o "F":
  - 1f, que es la unidad en representación "**float**"
- para que el valor sea "**double**", debe terminarse con la letra "d" o "D":
  - 1d, que es la unidad en representación "**double**"

## Valores fuera de rango

¿Qué pasa si nos salimos de rango?

<b>class Reales</b>
<pre>public class Reales {</pre>

```

public static void main(String[] args) {
    double n = Double.MAX_VALUE;
    System.out.println("n=    " + n);
    System.out.println("2n=   " + 2*n);    // overflow
    System.out.println();

    n = Double.MIN_VALUE;
    System.out.println("n=    " + n);
    System.out.println("n/2=  " + n/2);    // underflow
}
}

```

```

n=    1.7976931348623157E308
2n=   Infinity

```

```

n=    4.9E-324
n/2=  0.0

```

Ver "[desbordamiento](#)" y "[underflow](#)".

### ¿Qué tipo de reales debo usar?

Java tiene un especial cariño por los "double". Todos los números con cifras decimales los interpreta como "double", salvo que se le indique lo contrario.

Lo más normal es que todos los reales sean de tipo "double", salvo que haya una necesidad evidente de recurrir a otro tipo:

"float", porque tenemos muchísimos valores con valores pequeños

### paso de reales a enteros

Depende de lo que quiera hacer ...

número real	<code>double n = Math.PI</code>	3.141592653589793
parte entera	<code>int e = (int) n</code>	3
parte fraccionaria	<code>double f = n - e</code>	0.141592653589793
suelo	<code>Math.floor(n)</code>	3.0
techo	<code>Math.ceil(n)</code>	4.0
redondeo	<code>Math.round(4.45)</code>	4
	<code>Math.round(4.45)</code>	5

### notación local

Java utiliza normalmente notación anglosajona, donde la coma decimal es un punto '.' y el separador de miles es la coma ','.

Si deseamos utilizar notación local hay que recurrir a un formateador específico.

Ver "[Formatter](#)".

## 120. Objetos [objects] (concepto)

Son las diferentes materializaciones de una [clase](#).

A partir de una clase se crean objetos usando sentencias "[new](#)", bien directamente, bien indirectamente por medio de [fábricas](#).

```
Punto p = new Punto(3, 4);
Punto2D q = Punto2D.polares(1.0, 3.1416);
```

Cada objeto debe tener su nombre propio, que no debe repetirse dentro de su [ámbito](#) para evitar ambigüedades.

Por convenio, los nombres de los objetos deben comenzar por una letra minúscula.

## 121. Ocultación [hiding] (concepto)

Si tenemos un [ámbito](#) de visibilidad A1 y definimos un ámbito A2 dentro de A1, las variables definidas en A2 ocultan a las variables homónimas de A1.

Ejemplo:

<pre>1 class C { 2     private int x; 3     public void metodo(int x) { 5         this.x = x; 6     } 7 }</pre>	<p>Ámbito clase: líneas 1-7.</p> <p>Ámbito método: líneas 3-6.</p> <p>La x del ámbito clase (declarada en la línea 2) queda oculta por la x del ámbito método (declarada en la línea 3).</p> <p>El uso de <i>THIS</i> permite que en la línea 5 el valor de la x del método pase a la x de la clase.</p>
<pre>11 class C { 12     private int x1; 13     public void metodo(int x2) { 15         this.x1 = x2; 16     } 17 }</pre>	<p>Este bloque de código (11-17) hace exactamente lo mismo que el bloque (1-7) pero sin problemas de visibilidad.</p>
<pre>20 int m; 21 for (int m= 0; m &lt; ...; m++) { 22     metodo(m); 23 }</pre>	<p>La m declarada en la línea 20 queda oculta por la m declarada en la línea 21 en el ámbito del for (líneas 21-23).</p> <p>La m de las línea 21 y 23 se refieren a la declaración en el ámbito más estrecho: la de la línea 21.</p>

Ver "[ocultación vs redefinición](#)".

## 122. OO (acrónimo)

Object Oriented. Orientado a Objetos.

Ver "[OOP](#)".

## 123. OOP (acrónimo)

Object Oriented Programming. Programación orientada a objetos.

Es una forma de programar consistente en entender los programas como un conjunto de objetos que se relacionan entre sí por medio de los métodos. A menudo se dice "un paradigma de programación".

A menudo se dice que “los objetos intercambian mensajes” que no es otra cosa que decir que un objeto llama a los métodos de otro objeto pasándole de datos como argumentos y recibiendo datos como resultados.

La programación orientada a objetos resulta mejor estructurada que la programación clásica, disminuyendo el coste de desarrollo, mejorando la calidad de los programas y facilitando su mantenimiento posterior. Además, los objetos que constituyen un programa se prestan a ser reutilizados en otros, dando pie a bibliotecas de componentes que aceleran nuevos desarrollos.

A diferencia de la programación orientada a objetos, la programación clásica se decía procedural por cuanto se centraba en controlar el flujo de instrucciones (o sentencias) que iba ejecutando el ordenador. Aunque más simple conceptualmente, este modelo sobrevive malamente a la complejidad que supone la inmensa cantidad de instrucciones que realizan los programas de cierta envergadura.

## **124. Operadores (concepto)**

## **125. Overflow**

Ver "[desbordamiento](#)".

## **126. Overloading**

Ver "[sobrecarga de nombres](#)".

## **127. @Override**

Anotación que se puede poner justo antes de declarar un método. Le indica al compilador que el método que se va a declarar [redefine](#) a un método de alguna superclase. Si no fuera así, se produce un error en tiempo de compilación.

Es útil para prevenir errores causados por métodos que creemos que redefinen a otros pero que no es así por algún error.

Por ejemplo, es muy frecuente redefinir el método equals().

```
@Override
public boolean equals(Object x) { ... }
```

Un error frecuente es escribir mal el parámetro

```
@Override
public boolean equals(MiObjeto x) { ... }
```

Este error pasaría desapercibido sin la anotación @Override: el compilador interpretaría que el nuevo método no redefine a ningún otro, lo que es perfectamente posible.

Otro error frecuente es modificar la signatura de un método de una superclase y olvidar que hay que adaptar los métodos que lo redefinen. “@Override” detectaría el olvido.

Ver "[redefinición](#)".

## **128. package (palabra reservada)**

Ver "[paquetes](#)".

## **129. Palabras reservadas [keywords]**

Son una serie de palabras que forman parte del léxico de java y que no se pueden utilizar como identificadores de elementos del usuario:

<b>palabras reservadas</b>					
abstract	const	final	int	public	throw
assert	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp	try
case	else	if	null	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	synchronized	while
class	false	instanceof	protected	this	

### 130. Paquete [package] (concepto)

Los paquetes son una forma de agrupar varias clases. Es útil

- cuando el programa es muy grande, para estructurar las clases en grupos relacionados.
- para aprovechar la posibilidad de dotar a los miembros de una clase de visibilidad “*de paquete*”, de forma que sólo son visibles por otras clases del mismo paquete y no por clases de otros paquetes.

Aunque no es estrictamente necesario, se recomienda encarecidamente que la estructura de paquetes coincida con la estructura de directorios donde se encuentran los ficheros .java.

La declaración de paquete se hace por fichero .java. Es decir, si un fichero .java comienza diciendo

```
package x.y.z;
```

todas las clases que se definan en ese fichero se dirán del paquete `x.y.z`.

Y deben estar en el directorio

```
... / x / y / z /
```

#### nombres canónicos

Los nombres de las clases deben ser únicos dentro del mismo paquete. Si en un paquete `x` existe una clase `A` y en otro paquete `y` existe otra clase `A`, para romper la ambigüedad diremos que nos referimos a “`x.A`” o “`y.A`”, respectivamente.

A la concatenación del nombre del paquete y el nombre de la clase se le denomina “nombre canónico de la clase.”

En una aplicación Java es imprescindible que no haya dos clases con el mismo nombre canónico.

#### importación de paquetes

Siempre podemos referirnos a una clase empleando su nombre canónico completo. Por comodidad del programador y por alcanzar un código de más fácil lectura, podemos importar el nombre del paquete y luego referirnos a las clases del mismo por su identificador sin prefijo.

El paquete al que pertenece un fichero se importa directamente sin necesidad de indicarlo.

Si se importan dos paquetes con clases homónimas, el compilador exigirá que se indique el nombre canónico para que no haya dudas.

Ver "[import](#)".

### **nombre universales**

---

Es relativamente frecuente que las organizaciones usen su dominio de correo electrónico como denominación de los paquetes que ofrecen al público. Dado que las direcciones de correo son únicas, los paquetes también lo serán.

```
package es.upm.dit.fprg.evaluacion ;
package com.ibm.... ;
```

## **131. Parámetros (concepto)**

Ver "[argumentos](#)".

## **132. Pilas [stacks] (estructura de datos)**

Son listas de objetos que se caracterizan porque los nuevos objetos se añaden al final, y también salen por el final. De esta forma, resulta que el último que entra es el primero que sale (en Inglés, LIFO: Last In, First Out).

<b><i>Pilas (listas LIFO)</i></b>
<pre>public interface Pila&lt;T&gt; {     // mete un objeto T al final de la pila     void push(T t) throws ExcepcionPilaLlena;      // retira el último objeto de la pila     T pop() throws ExcepcionPilaVacía;      // mira, sin retirar, el último objeto     T top() throws ExcepcionPilaVacía;      // objetos en la pila     int longitud(); }</pre>

Es fácil implementar las pilas como listas:

<b><i>Pila implementada con una lista</i></b>
<pre>import java.util.*;  public class PilaLista&lt;T&gt; implements Pila&lt;T&gt; {     private List&lt;T&gt; lista = new ArrayList&lt;T&gt;();      public void push(T t) {         lista.add(t);     }      public T top() throws ExcepcionPilaVacía {         if (lista.size() == 0) throw new ExcepcionPilaVacía();         return lista.get(lista.size()-1);     } }</pre>



### ***Pila implementada con una lista***

```
public T pop() throws ExcepcionPilaVacía {
    if (lista.size() == 0) throw new ExcepcionPilaVacía();
    return lista.remove(lista.size()-1);
}

public int longitud() {
    return lista.size();
}
}
```

## **133. Polimorfismo [polimorphism] (concepto)**

Es la posibilidad de que una variable se refiera a objetos de diferentes clases. El comportamiento exacto depende de la clase exacta del objeto referido.

Tenemos polimorfismo cuando:

- diferentes clases [implementan](#) una misma [interface](#)
- unas clases son [subclases](#) de otras

### ***interface Punto***

```
public interface Punto {
    double getX();
    double getY();
    double getModulo();
    double getAngulo();
}
```

#### ***class Cartesianas***

```
public class Cartesianas
    implements Punto {
    private double x;
    private double y;

    public Cartesianas(double x,
                        double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getModulo() {
        return
            Math.sqrt(x*x + y*y);
    }
}
```

#### ***class Polares***

```
public class Polares
    implements Punto {
    private double modulo;
    private double angulo;

    public Polares(double modulo,
                    double angulo) {
        this.modulo = modulo;
        this.angulo = angulo;
    }

    public double getX() {
        return modulo *
            Math.cos(angulo);
    }

    public double getY() {
        return modulo *
            Math.sin(angulo);
    }

    public double getModulo() {

```

<i>interface Punto</i>	
<pre>     }      public double getAngulo() {         return Math.atan2(y, x);     } } </pre>	<pre>         return modulo;     }      public double getAngulo() {         return angulo;     } } </pre>

```

Punto p = new Cartesianas(1.0, 1.0);
Punto q = new Polares(1.0, 1.0);

```

```
p.getModulo()
```

utiliza Cartesianas.getModulo()

```
q.getModulo()
```

utiliza Polares.getModulo()

Ver "[elección dinámica de método](#)".

### **134. print (método) void print(...)**

Método de las clases [PrintStream](#) y [PrintWriter](#).

Imprime un valor. Está sobrecargado para tratar números, booleanos, caracteres y cualquier objeto. En este último caso imprime la cadena devuelta por el método [toString\(\)](#).

Ver [System.out](#) y [System.err](#).

### **135. printf (método) void printf(String, Object ...)**

Método de las clases [PrintStream](#) y [PrintWriter](#).

Sinónimo de [format\(\)](#).

Imprime una serie de valores siguiendo el formato proporcionado como primer argumento.

Ver "[Formatter](#)" donde se explica la notación usada en la descripción de formato.

Ver [System.out](#) y [System.err](#).

### **136. println (método) void println(...)**

Método de las clases [PrintStream](#) y [PrintWriter](#).

Imprime un valor seguido de un fin de línea. Está sobrecargado para tratar números, booleanos, caracteres y cualquier objeto. En este último caso imprime la cadena devuelta por el método [toString\(\)](#).

Ver [System.out](#) y [System.err](#).

### **137. private (palabra reservada)**

Ver "[visibilidad](#)": ámbito de código en el que un elemento de Java puede referenciarse por su nombre.

Los elementos "**private**" sólo pueden referenciarse dentro del mismo fichero en el que se definen.

### **private class X**

La clase X sólo pueden referenciarse dentro del mismo fichero en el que se define.

Se pueden definir clases “**private**” dentro de otras clases o junto con otra clase en el mismo fichero.

### **private resultado método (argumentos)**

El método sólo pueden referenciarse dentro del mismo fichero en el que se define.

Se pueden definir métodos “**private**” dentro de cualquier clase.

### **private variable**

La variable sólo pueden referenciarse dentro del mismo fichero en el que se define.

Se pueden definir variables “**private**” como campos de cualquier clase.

## **138. Programación orientada a objetos (concepto)**

Ver “[OOP](#)”.

## **139. Programación estructurada (concepto)**

Una forma de estructurar los programas para que no sean un laberinto, sino algo fácil de entender por uno mismo y por los demás.

La idea fundamental es que los programas son un conjunto de bloques que encajan entre sí sobre la premisa de que cada bloque tiene un sólo punto de entrada y un sólo punto de salida, y la salida de uno se enchufa a la entrada del siguiente.

O sea, un lego.

En java todas las estructuras sintácticas tienen un punto de entrada y un punto de salida.

## **140. Programa [program] (concepto)**

Un programa o aplicación está formado por una serie de clases con el objetivo de resolver un problema. Un programa aplica un [algoritmo](#) para transformar los valores iniciales (datos de entrada) en valores finales (resultados o datos de salida).

En Java un programa arranca ejecutando el método “[main](#)” de alguna clase. A partir de este método se crean los objetos pertinentes y se realizan las transformaciones sobre los valores.

## **141. Promoción [widening] (concepto)**

Se dice de la conversión automática de valores entre [tipos primitivos](#). Convierte de un formato “más pobre” a un formato “más rico”.

A veces se conoce como *upcasting* de tipos primitivos.

Las promociones toleradas por java son

```
byte → short → int → long → float → double
char → int → long → float → double
```

La promoción se produce automáticamente (el compilador la realiza sin quejarse ni avisar al programador) cuando:

- en una expresión se combinan valores de diferente tipo
- se intenta asignar a una variable de un cierto tipo un valor de otro tipo

Por ejemplo:

- 2.0 / 3  
se promociona el (int)3 a (double)3.0 y la expresión vale (double)1.666
- double x = 3;  
se promociona el (int)3 a (double)3.0 y la variable recibe el valor (double)3.0

La operación inversa de la promoción es la [reducción](#).

## 142. **protected** (palabra reservada)

Ver "[visibilidad](#)": ámbito de código en el que un elemento de Java puede referenciarse por su nombre.

Los elementos "**protected**" pueden referenciarse desde cualquier punto del código dentro del mismo paquete en el que se define o en [subclases](#) ("**extend**") de aquella en que se definen.

### **protected class X**

La clase X puede referenciarse desde cualquier punto del código dentro del mismo paquete en el que se define y, además, en cualquier subclase de aquella en la que se define.

Se pueden definir clases "**protected**" dentro de otras clases o junto con otra clase en el mismo fichero.

### **protected resultado método (argumentos)**

El método puede referenciarse (ejecutarse) desde cualquier punto del código dentro del mismo paquete en el que se define y, además, en cualquier subclase de aquella en la que se define siempre y cuando el objeto sea de la subclase.

Se pueden definir métodos "**protected**" dentro de cualquier clase.

### **protected variable**

La variable puede referenciarse (leer o escribir) desde cualquier punto del código dentro del mismo paquete en el que se define y, además, en cualquier subclase de aquella en la que se define siempre y cuando el objeto sea de la subclase.

Se pueden definir variables "**protected**" como campos de cualquier clase. Pero no se recomienda; es preferible definir las variables como "**private**" y establecer métodos de acceso para lectura (getX() o isX()) y escritura (setX()).

## 143. **public** (palabra reservada)

Ver "[visibilidad](#)": ámbito de código en el que un elemento de Java puede referenciarse por su nombre.

Los elementos "**public**" pueden referenciarse desde cualquier punto del código, incluyendo desde otros paquetes.

### **public class X**

La clase X puede referenciarse desde cualquier punto del código, incluyendo desde otros paquetes.

Se pueden definir cero o una clase "**public**" en cada fichero del proyecto, siendo el nombre del fichero igual al de la clase (más la extensión ".java").

### **public resultado método (argumentos)**

El método puede referenciarse (ejecutarse) desde cualquier punto del código, incluyendo desde otros paquetes.

Se pueden definir métodos “**private**” dentro de cualquier clase.

### public variable

La variable puede referenciarse (leer o escribir) desde cualquier punto del código, incluyendo desde otros paquetes.

Se pueden definir variables “**public**” como campos de cualquier clase. Pero no se recomienda; es preferible definir las variables como “**private**” y establecer métodos de acceso para lectura (getX() o isX()) y escritura (setX()).

## 144. Recursión (concepto)

Ver “[métodos recursivos](#)”.

## 145. Redefinición de métodos [method overriding] (concepto)

Cuando un método cambia su contenido sin cambiar su signature.

Ocurre cuando una clase B extiende a otra A y B tiene un método con la misma signature que un método de A.

class Operacion
<pre>public class Operacion {     public double aplica(double x) {         return x;     } }</pre>
class Duplica extends Operacion
<pre>public class Duplica     extends Operacion {     @Override     public double aplica(double x) {    // redefine el método         return 2 * x;     } }</pre>
class Suma extends Operacion
<pre>public class Suma     extends Operacion {     private double k;      public Suma(double k) {         this.k = k;     }      @Override     public double aplica(double x) {    // redefine el método         return x + k;     } }</pre>

La anotación “[@Override](#)” le pide al compilador que verifique que efectivamente estamos redefiniendo un método de la superclase. Si por error (por ejemplo, si nos equivocamos en los parámetros) el método no redefine nada, el compilador lanza un error. Otro error frecuente es modificar la signature de un método de una superclase y olvidar que hay que adaptar los métodos que lo redefinen. “@Override” detectaría el olvido.

Cuando se van a utilizar las clases derivadas, se aplica [polimorfismo](#):

```
Operacion opA = new Operacion();
Operacion opB = new Duplica();
Operacion opC = new Suma(3);
opA.aplica(7); // devuelve 7
opB.aplica(7); // devuelve 14
opC.aplica(7); // devuelve 10
```

Ver "[herencia](#)".

En la subclase se pueden usar los métodos de la subclase y los métodos de la superclase, teniendo en cuenta que cuando un método de la superclase ha sido redefinido, el acceso directo es a la redefinición en la subclase. No obstante, todos los métodos de la superclase son accesibles por medio de "[super](#)".

### **ejemlo: polígono**

Se redinen los métodos por razones de eficiencia (más rapidez).

<b><i>class Poligono</i></b>
<pre>public class Poligono {     private final Punto2D[] vertices;      public Poligono(Punto2D... vertices) {         this.vertices = vertices;     }      <b>public double getPerimetro() {</b>         double p = 0;         Punto2D p1 = vertices[vertices.length - 1];         for (Punto2D p2 : vertices)             p += p1.getDistancia(p2);         return p;     } }</pre>
<b><i>class Triangulo extends Poligono</i></b>
<pre>public class Triangulo     extends Poligono {     public Triangulo(Punto2D A, Punto2D B, Punto2D C) {         super(A, B, C);     } }</pre>
<b><i>class Cuadrilatero extends Poligono</i></b>
<pre>public class Cuadrilatero     extends Poligono {     public Cuadrilatero(Punto2D A, Punto2D B, Punto2D C, Punto2D D) {         super(A, B, C, D);     } }</pre>
<b><i>class Rectangulo extends Cuadrilatero</i></b>
<pre>public class Rectangulo     extends Cuadrilatero {     private final double base;     private final double altura;</pre>

<pre> public Rectangulo(Punto2D centro, double base, double altura) {     super(centro.desplazado(-base / 2, altura / 2),           centro.desplazado(base / 2, altura / 2),           centro.desplazado(base / 2, -altura / 2),           centro.desplazado(-base / 2, -altura / 2)     );     this.base = base;     this.altura = altura; }  @Override public double getPerimetro() {     return 2 * base + 2 * altura; } } </pre>
<b><i>class Rombo extends Cuadrilatero</i></b>
<pre> public class Rombo     extends Cuadrilatero {     private final double eje1;     private final double eje2;      public Rombo(Punto2D centro, double eje1, double eje2) {         super(centro.desplazado(0, eje2 / 2),               centro.desplazado(eje1 / 2, 0),               centro.desplazado(0, -eje2 / 2),               centro.desplazado(-eje1 / 2, 0)         );         this.eje1 = eje1;         this.eje2 = eje2;     }      @Override     public double getPerimetro() {         double sx = eje1 / 2;         double sy = eje2 / 2;         return 4 * Math.sqrt(sx * sx + sy * sy);     } } </pre>

### **ejemplo: colas de tamaño limitado**

Se redefinen los métodos por razones de funcionalidad (se cambia el funcionamiento).

<b><i>class Cola</i></b>
<pre> public class Cola {     private List&lt;String&gt; datos = new ArrayList&lt;String&gt;();      public Cola() {     }      // mete sin límite     public void mete(String s) {         datos.add(s);     } } </pre>

<pre>     }      public String saca() {         return datos.remove(0);     }      public int size() {         return datos.size();     } } </pre>
<b>class ColaMax1 extends Cola</b>
<pre> public class ColaMax1     extends Cola {     private final int max;      public ColaMax1(int max) {         this.max = max;     }      // mete si no hemos llegado al límite     @Override     public void mete(String s) {         if (size() &lt; max)             super.mete(s);     } } </pre>
<b>class ColaMax2 extends Cola</b>
<pre> public class ColaMax2     extends Cola {     private final int max;      public ColaMax2(int max) {         this.max = max;     }      // saca el primero si se va a superar el límite     @Override     public void mete(String s) {         if (size() &gt;= max)             super.saca();         super.mete(s);     } } </pre>

## 146. Reducción [narrowing] (concepto)

Se dice de la conversión forzada de valores de tipos primitivos.

Es lo contrario de la [promoción](#), convirtiendo de un formato “más rico” en un formato “más pobre”.

A veces se conoce como *downcasting* de tipos primitivos.

Las promociones toleradas por java son

```
byte ← short ← int ← long ← float ← double
```



char ← int ← long ← float ← double

Al contrario que la promoción, el compilador se niega a reducir valores salvo que se le ordene explícitamente:

```
int x = (int)3.5;    // x carga el valor 3
```

Cuando un valor real se reduce a un valor entero, nos quedamos exclusivamente con la parte entera, sin realizar ningún tipo de redondeo.

Si el formato destino de la conversión es incapaz de contener el valor, el resultado es imprevisible. Pero tenga en cuenta que java no se va a quejar (no se produce ningún error ni de compilación, ni de ejecución).

## 147. Refactoring (concepto)

Actividad por la que vamos modificando el código fuente, sin modificar su funcionamiento, pero buscando una mejor legibilidad o estructuración.

Algunas actividades de *refactoring* son evidentes:

- renombrar variables para que su nombre refleje intuitivamente su contenido.
- renombrar métodos para que su nombre refleje intuitivamente su cometido.
- renombrar clases para que su nombre refleje intuitivamente su esencia.
- reordenar las clases en paquetes para que la estructura agrupe funciones de detalle.

Ver "<http://www.refactoring.com/>".

### encapsulación

---

Consiste en sustituir campos públicos por campos privados con métodos de acceso.

<b>Refactoring: encapsulación</b>
<pre>public String nombre;</pre>
<pre>private String nombre;  public String getNombre() {     return nombre; }  public void setNombre(String nombre) {     this.nombre = nombre; }</pre>

### introducción de variables explicativas

---

Una expresión imbuida en una expresión más grande puede extraerse para aclarar el significado de la sub-expresión. Para ello se introducen variables auxiliares.

<b>Refactoring: variables explicativas</b>
<pre>public boolean bisiesto(int año) {     return (año % 4 == 0 &amp;&amp; (!(año % 100 == 0)))    año % 400 == 0; }</pre>

### ***Refactoring: variables explicativas***

Descomponemos la expresión booleana en expresiones más sencillas y explicativas:

```
public boolean bisiestro(int año) {
    boolean multiplo4    = año % 4 == 0;
    boolean multiplo100  = año % 100 == 0;
    boolean multiplo400  = año % 400 == 0;
    return (multiplo4 && (! multiplo100)) || multiplo400;
}
```

### **extracción / eliminación de variables**

En la extracción de variables se elige una expresión y se calcula asignando el resultado a una variable; a continuación, se emplea la variable en donde aparecía la expresión. La variable puede ser local o ser un campo del objeto, según convenga.

La extracción de variables permite

- dar un nombre significativo a una expresión
- simplificar las expresiones que usan la nueva variable
- acelerar la ejecución cuando la expresión reemplazada se evalúa una sola vez en vez de varias veces

A modo de ejemplo, se muestra en diferentes pasos como la extracción de variables ayuda a mejorar la legibilidad del código:

### ***Refactoring: extracción de variables***

#### ***Paso 0: código que funciona; pero de difícil lectura***

```
public int[][] mezcla(int[][] m1, int[][] m2) {
    int[][] resultado =
        new int[Math.max(m1.length, m2.length)]
            [Math.max(m1[0].length, m2[0].length)];
    for (int i = 0; i < Math.max(m1.length, m2.length); i++) {
        for (int j = 0;
            j < Math.max(m1[0].length, m2[0].length);
            j++) {
            if (i < m1.length && j < m1[0].length) {
                if (i < m2.length && j < m2[0].length) {
                    resultado[i][j] = (m1[i][j] + m2[i][j]) / 2;
                } else {
                    resultado[i][j] = m1[i][j];
                }
            } else {
                if (i < m2.length && j < m2[0].length) {
                    resultado[i][j] = m2[i][j];
                } else {
                    resultado[i][j] = 0;
                }
            }
        }
    }
    return resultado;
}
```

### ***Refactoring: extracción de variables***

```
}
```

#### ***Paso 1: extracción de las variables filasM1, columnasM1, filasM2 y columnasM2***

```
public int[][] mezcla(int[][] m1, int[][] m2) {
    int filasM1 = m1.length;
    int columnasM1 = m1[0].length;
    int filasM2 = m2.length;
    int columnasM2 = m2[0].length;
    int[][] resultado =
        new int[Math.max(filasM1, filasM2)]
            [Math.max(columnasM1, columnasM2)];
    for (int i = 0; i < Math.max(filasM1, filasM2); i++) {
        for (int j = 0; j < Math.max(columnasM1, columnasM2); j++) {
            if (i < filasM1 && j < columnasM1) {
                if (i < filasM2 && j < columnasM2) {
                    resultado[i][j] = (m1[i][j] + m2[i][j]) / 2;
                } else {
                    resultado[i][j] = m1[i][j];
                }
            } else {
                if (i < filasM2 && j < columnasM2) {
                    resultado[i][j] = m2[i][j];
                } else {
                    resultado[i][j] = 0;
                }
            }
        }
    }
    return resultado;
}
```

#### ***Paso 2: introducción de las variables filasR y columnasR***

```
public int[][] mezcla(int[][] m1, int[][] m2) {
    int filasM1 = m1.length;
    int columnasM1 = m1[0].length;
    int filasM2 = m2.length;
    int columnasM2 = m2[0].length;
    int filasR = Math.max(filasM1, filasM2);
    int columnasR = Math.max(columnasM1, columnasM2);
    int[][] resultado = new int[filasR][columnasR];
    for (int i = 0; i < filasR; i++) {
        for (int j = 0; j < columnasR; j++) {
            if (i < filasM1 && j < columnasM1) {
                if (i < filasM2 && j < columnasM2) {
                    resultado[i][j] = (m1[i][j] + m2[i][j]) / 2;
                } else {
                    resultado[i][j] = m1[i][j];
                }
            } else {
                if (i < filasM2 && j < columnasM2) {
                    resultado[i][j] = m2[i][j];
                } else {
                    resultado[i][j] = 0;
                }
            }
        }
    }
}
```

<b>Refactoring: extracción de variables</b>
<pre>         }     } } return resultado; } </pre>
<b>Paso 3: introducción de las variables <i>estaEnM1</i> y <i>estaEnM2</i></b>
<pre> public int[][] mezcla(int[][] m1, int[][] m2) {     int filasM1 = m1.length;     int columnasM1 = m1[0].length;     int filasM2 = m2.length;     int columnasM2 = m2[0].length;     int filasR = Math.max(filasM1, filasM2);     int columnasR = Math.max(columnasM1, columnasM2);     int[][] resultado = new int[filasR][columnasR];     for (int i = 0; i &lt; filasR; i++) {         for (int j = 0; j &lt; columnasR; j++) {             boolean estaEnM1 = i &lt; filasM1 &amp;&amp; j &lt; columnasM1;             boolean estaEnM2 = i &lt; filasM2 &amp;&amp; j &lt; columnasM2;             if (estaEnM1) {                 if (estaEnM2) {                     resultado[i][j] = (m1[i][j] + m2[i][j]) / 2;                 } else {                     resultado[i][j] = m1[i][j];                 }             } else {                 if (estaEnM2) {                     resultado[i][j] = m2[i][j];                 } else {                     resultado[i][j] = 0;                 }             }         }     }     return resultado; } </pre>

La operación inversa de la extracción de variables es la eliminación de variables (en inglés denominada *inlining*). Consiste en reemplazar el uso de una variable por la expresión en ella cargada. En el ejemplo anterior, consiste en ejecutar los pasos en orden inverso.

### **sustitución de condiciones anidadas por guardas**

A veces se acumulan varias condiciones anidando unas tras otras. Aunque correcto, el código puede llegar a ser ininteligible.

<b>condiciones anidadas</b>	<b>guardas</b>
<pre> public String texto(double nota) {     String txt;     if (nota &gt; 9.99)         txt = "Matrícula de Honor"; } </pre>	<pre> public String texto(double nota) {     if (nota &gt; 9.99)         return "Matrícula de Honor";     if (nota &gt; 8.99) } </pre>

<b>condiciones anidadas</b>	<b>guardas</b>
<pre> else if (nota &gt; 8.99)     txt = "Sobresaliente"; else if (nota &gt; 6.99)     txt = "Notable"; else if (nota &gt; 4.99)     txt = "Aprobado"; else     txt = "Suspendido"; return txt; </pre>	<pre> return "Sobresaliente"; if (nota &gt; 6.99)     return "Notable"; if (nota &gt; 4.99)     return "Aprobado"; return "Suspendido"; } </pre>

### **extracción de métodos**

Consiste en la identificación de una serie de líneas de código que se llevan a un método aparte, reemplazando las líneas originales por una llamada al nuevo método.

Es útil

- para extraer código común que se repite en varios sitios
- para hacer más legible un programa, dándole un nombre a unas líneas de código
- para evitar el uso de "[break](#)"

<b>Refactoring: extracción de métodos</b>
<pre> // cálculo de la diagonal mayor de un paralelepípedo rectangular public double getDiagonalMayor(double a, double b, double c) {     return Math.sqrt(Math.sqrt(a * a + b * b) *         Math.sqrt(a * a + b * b) + c * c); } </pre>
Extraemos un método que aplica Pitágoras:
<pre> // cálculo de la diagonal mayor de un paralelepípedo rectangular public double getDiagonalMayor(double a, double b, double c) {     return hipotenusa(hipotenusa(a, b), c); }  // teorema de Pitágoras private double hipotenusa(double x, double y) {     return Math.sqrt(x * x + y * y); } </pre>

El siguiente ejemplo es un programa que detecta si una String es un palíndromo

<b>Refactoring: extracción de métodos para evitar "break"</b>
<pre> public void testSimetria(String s) {     boolean esSimetrica = true;     for (int i = 0; i &lt; s.length(); i++) {         int j = s.length() - 1 - i;         if (j &lt; i)             break;         char c1 = s.charAt(i); </pre>

```

        char c2 = s.charAt(j);
        if (c1 != c2) {
            esSimetrica = false;
            break;
        }
    }
    System.out.println(esSimetrica);
}

```

Extraemos el bucle nuclear en un método auxiliar:

```

public void testSimetria2(String s) {
    System.out.println(isSimetrica(s));
}

private boolean isSimetrica(String s) {
    for (int i = 0; i < s.length(); i++) {
        int j = s.length() - 1 - i;
        if (j < i)
            return true;
        char c1 = s.charAt(i);
        char c2 = s.charAt(j);
        if (c1 != c2)
            return false;
    }
    return true;          // necesario para la cadena vacía ""
}

```

### **sustituir iteración por recursión (o viceversa)**

Los programas iterativos suelen ser más rápidos, mientras que los programas recursivos suelen ser más fáciles de entender. El programador puede elegir una u otra forma, según le convenga.

Los siguientes métodos calculan el máximo común divisor de dos números enteros positivos.

<i><b>iterativo</b></i>	<i><b>recursivo</b></i>
<pre> int mcd1(int a, int b) {     while (a != b) {         if (a &gt; b)             a -= b;         else if (b &gt; a)             b -= a;     }     return a; } </pre>	<pre> static int mcd2(int a, int b) {     if (a == b)         return a;     if (a &gt; b)         return mcd2(a - b, b);     else         return mcd2(a, b - a); } </pre>

## **148. Referencias [references] (concepto)**

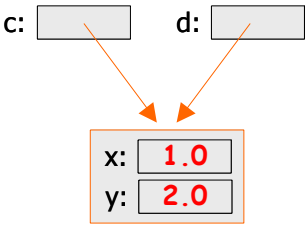
Cuando el programador crea un objeto, java lo identifica por su referencia.

Gráficamente, puede concebirse una referencia como un puntero al objeto.

Cuando a una variable se le asigna un objeto, exactamente se carga en la variable una copia del valor de dicho puntero. Cuando el valor de una variable se asigna a otra

```
x = y;
```

se carga en x otra copia del puntero almacenado en y. Se dice que ambas variables se refieren al mismo objeto o, en otras palabras, que el objeto es compartido por ambas variables. Al ser un objeto compartido, lo que se cambie a través de una de las variables que lo referencian se ve cambiado para la otra.

código java	variables y objetos
<pre>Punto c, d; c = new Punto(0, 0); d = c; c.x = 1; d.y = 2;</pre>	

Véase “paso de argumentos por referencia”.

Cuando una variable no apunta a ningún objeto, se dice que contiene la referencia “**null**” que es la referencia que no apunta a nada.

### 149. *return* (palabra reservada)

Sirve para terminar un método, devolviendo el resultado, si el método lo requiere.

Las sentencias **return** provocan la terminación del método en el que aparecen, incluso si estamos dentro de algún bucle.

```
public int buscaPunto(String s) {
    for (int p = 0; p < s.length(); p++) {
        char c = s.charAt(p);
        if (c == '.')
            return p;
    }
    return -1;
}
```

### 150. *RuntimeException* (clase) *java.lang.RuntimeException*

Un tipo de excepciones que se caracteriza porque no hay que explicitarlo en la cabecera de los métodos que, no deseando tratar la excepción ellos mismos, prefieren propagarla a quien los ha llamado.

Se utiliza para aquellos errores que son de programación; es decir, imputables al programador que se ha equivocado, que no a los datos de una ejecución concreta.

Ver “[Exception](#)” y “[Excepciones](#)”.

### 151. *SDK* (acrónimo)

Software Development Kit. Entorno de desarrollo.

Ver “[JDK](#)”.

### 152. *Setters* (concepto)

Ver “[métodos](#)”/ métodos de carga (setters”).

### 153. short (palabra reservada)

<b>short : números enteros</b>	
<b>ocupación</b>	2 bytes = 16 bits
<b>mínimo</b>	Short.MIN_VALUE = $-2^{15} = -32.768$
<b>máximo</b>	Short.MAX_VALUE = $2^{15} - 1 = +32.767$

Ver "[Números](#)".

### 154. Signatura (de un método) (concepto)

Se denomina signatura de un método a lo que lo caracteriza de forma inconfundible: su nombre y la serie de argumentos (número, tipo y orden).

Ver "[métodos](#)".

### 155. Sobrecarga de nombres [name overloading] (concepto)

En un mismo [ámbito](#) pueden coexistir dos o más métodos homónimos (mismo nombre o identificador) siempre y cuando difieran en el número, orden o tipo de sus argumentos.

No es suficiente diferenciación diferir en el tipo del valor devuelto o en las excepciones que pueden lanzar.

#### sobrecarga, promoción y upcasting

Dos métodos pueden diferir en tipos de argumentos que, sin ser idénticos, puedan admitir datos comunes, bien por promoción automática (caso de tipos primitivos) bien por *upcasting* (caso de objetos). En tiempo de compilación java opta por el método que no requiera ni promoción ni *upcasting*.

<b>sobrecarga</b>	
<b>con promoción</b>	<b>con upcasting</b>
<pre>void metodo(int x) { ... }  void metodo(long x) { ... }</pre>	<pre>class B extends A { ... }  void metodo(A a) { ... }  void metodo(B b) { ... }</pre>
<pre>int v = 5; metodo(v)      prefiere metodo(int);     si no existiera, se acogería a metodo(long)</pre>	<pre>B b = new B(); metodo(b)      prefiere metodo(B);     si no existiera, se acogería a metodo(A)</pre>

Esta decisión se realiza en tiempo de compilación. Además, en tiempo de ejecución, puede entrar en juego la existencia de métodos [redefinidos](#), en cuyo caso se aplicaría [polimorfismo](#) sobre el método seleccionado en tiempo de ejecución.



## 156. static (palabra reservada)

Se utiliza para referirse a los miembros de una clase.

### campos static

---

Ver "[variables de clase](#)".

### métodos static

---

Son [métodos](#) de clase:

- se pueden utilizar sin declarar objetos: basta referirse a la clase
- sólo pueden acceder a variables de clase  
No pueden acceder a los campos de los objetos porque no hay objetos (todavía).
- ver "[fábricas](#)"

<i>class Math</i>
<pre>public class Math {     public <b>static</b> double max(double a, double b) {         if (a &lt; b)             return a;         return b;     } }</pre>
<pre>int maximo = Math.max(x, y);</pre>

### bloques static

---

Son bloques entre llaves { ... } que sirven para inicializar campos de clase

<pre>public class Diccionario {     private static final String[][]diccionario;      <b>static</b> {         diccionario = new String[2][2];         diccionario[0] = new String[]{"hola", "hi"};         diccionario[1] = new String[]{"adios", "bye"};     } }</pre>
--

## 157. subclases (concepto)

Se dice que B es subclase de A cuando B es extensión directa o indirecta de A.

```
class B extends A { ... }
```

Ver "[extensión](#)".

## 158. subtipo (concepto)

En java, es sinónimo de "[subclass](#)".

Ver “[extensión](#)”.

## 159. *super* (palabra reservada)

Cuando una clase B [extiende](#) a otra A

```
class B extends A { ... }
```

los miembros (valores y métodos) no privados de la clase A son accesibles en B usando el prefijo “**super.**”, incluso si el método ha sido [redefinido](#) en B.

<i>class Punto2D</i>
<pre>public class Punto2D {     private double x, y;      public void <b>set</b>(double[] coordenadas) {         this.x = coordenadas[0];         this.y = coordenadas[1];     } }</pre>
<i>class Punto3D extends Punto2D</i>
<pre>public class Punto3D extends Punto2D {     private double z;      public void <b>set</b>(double[] coordenadas) {         <b>super</b>.set(coordenadas);         this.z = coordenadas[2];     } }</pre>

Una clase sólo puede referirse a su madre inmediata, no se pueden dar 2 saltos "hasta la abuela".

### **super en constructores**

Se puede usar "super(...)" como primera sentencia de un constructor de una [subclase](#) para ejecutar el constructor de su [superclase](#).

<i>class Poligono</i>
<pre>public class Poligono {     private Punto[] vertices;      public Poligono(Punto... vertices) {         this.vertices = new Punto[vertices.length];         for (int i = 0; i &lt; vertices.length; i++)             this.vertices[i] = vertices[i];     } }</pre>
<i>class Cuadrado extends Poligono</i>
<pre>class Cuadrado extends Poligono {     public Cuadrado(Punto centro, double lado) {         super(             new Punto(centro.getX() - lado/2, centro.getY() + lado/2),             new Punto(centro.getX() + lado/2, centro.getY() + lado/2),             new Punto(centro.getX() + lado/2, centro.getY() - lado/2),             new Punto(centro.getX() - lado/2, centro.getY() - lado/2));     } }</pre>

```

    }
}

```

Ver "[extensión](#)".

### 160. *superclases (concepto)*

Se dice que A es superclase de B cuando B es extensión directa o indirecta de A.

```
class B extends A { ... }
```

Ver "[extensión](#)".

### 161. *superipo (concepto)*

En java, es sinónimo de "superclase".

Ver "[extensión](#)".

### 162. *sustitución (concepto)*

Es la capacidad de utilizar un objeto de una [subclase](#) en una variable de una [superclase](#).

<i>class Poligono</i>
<pre>public class Poligono {     ...     ...     ... }</pre>
<i>class Cuadrado extends Poligono</i>
<pre>class Cuadrado extends Poligono {     ...     ...     ... }</pre>
<i>sustitución (upcasting)</i>
<pre>Poligono poligono = new Cuadrado();</pre>
<pre>ArrayList&lt;Poligono&gt; dibujo = new ArrayList&lt;Poligono&gt;;  dibujo.add(new Cuadrado());</pre>

Ver "[upcasting](#)".

### 163. *switch (palabra reservada)*

Construcción sintáctica muy compacta para representar la ejecución de una entre varias sentencias dependiendo de un valor:

<i>ejemplo</i>	<i>ejemplo</i>
<pre>switch (mes) { case 1:</pre>	<pre>switch (mes) { case 1: case 3: case 5: case 7:</pre>

<i><b>ejemplo</b></i>	<i><b>ejemplo</b></i>
<pre> print("enero"); break; case 2:     print("febrero");     break; ... default:     print("no se"); </pre>	<pre> case 8: case 10: case 12:     dias = 31;     break; case 4: case 6: case 9: case 11:     dias = 30;     break; case 2:     if (bisiesto)         dias = 29;     else         dias = 28;     break; default:     dias = 0; } </pre>

- Sólo funcionan sobre enteros, booleanos o caracteres
- Chequean que no hay duplicados
  - las condiciones tienen que ser excluyentes
- Mejor legibilidad
- Sin **"break"**, el programa no saltaría al final
- El **"default"** es opcional; si no aparece, no se ejecuta nada

Las sentencias **"break"** provocan la terminación de la sentencia condicional. Si no se aparece, el código siguiente se sigue ejecutando.

<i><b>Switcher.java</b></i>
<pre> public class Switcher {     public static void main(String[] argumentos) {         int m = Integer.parseInt(argumentos[0]);         int n = 0;         switch (m) {             case 0:                 n++;             case 1:                 n++;             case 2:                 n++;             default:                 n++;         }         System.out.println("n = " + n);     } } </pre>

```

$ java Switcher 0
n = 4

$ java Switcher 1
n = 3

$ java Switcher 2
n = 2

$ java Switcher 3
n = 1

$ java Switcher 100
n = 1

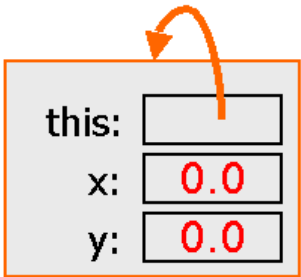
```

## 164. *this* (palabra reservada)

Todos los objetos disponen de un campo que se llama “**this**” y contiene una referencia al propio objeto.

### Uso de *this* para acceder a campos del objeto

Es frecuente utilizar “**this**” cuando hay ambigüedad entre los campos del objeto y otras variables de [ámbito](#) más cercano.

<i>java</i>	<i>variables</i>
<pre> public class Punto {     private double x;     private double y;      public Punto(double x, double y) {         <b>this</b>.x = x;         <b>this</b>.y = y;     } } </pre>	

### Uso de *this* para acceder a constructores de objetos

Cuando un queremos llamar a un constructor desde otro constructor, usaremos *this*:

```

class Elipse {
    private double ancho, alto;

    Elipse(double alto, double ancho) {
        this.alto= alto;
        this.ancho= ancho;
    }

    // construye un círculo
    Elipse(double diametro) {
        this(diametro, diametro);
    }

    // construye un círculo de diámetro 10

```

```
Ellipse() {  
    this(10);  
}
```

### **165. throw (palabra reservada)**

Para lanzar una excepción que pueda ser capturada por un bloque **catch** anexo a un bloque **try**.

Ver "[excepciones](#)".

### **166. throws (palabra reservada)**

Para indicar que un método puede lanzar una excepción, renunciando a capturarla internamente.

Ver "[excepciones](#)".

Es necesario indicar explícitamente cualquier excepción que pueda ser lanzada, excepto aquellas que sean [subclases](#) (**extends**) de

- *java.lang.Error*
- *java.lang.RuntimeException*

### **167. Tipos abstractos de datos (TAD) (concepto)**

Abstract Data Types (ADT).

Ver "[encapsulación](#)".

### **168. Tipos formales [type parameters]**

Dícese de los tipos que quedan como parámetros en la definición de tipos y métodos genéricos.

Ver "[genéricos](#)".

### **169. Tipos primitivos [primitive data types]**

Se dice de los siguientes

- enteros: [int](#), [byte](#), [short](#) y [long](#)
- reales: [double](#) y [float](#)
- caracteres: [char](#)
- valores lógicos: [boolean](#)

### **170. toString (método) public String toString()**

Todos los objetos java disponen de este método que devuelve una cadena representativa del objeto.

Lo más habitual es que el programador codifique el método para que la cadena devuelta sea cómoda y útil:

```
public class Punto {  
    private int x, y;  
  
    public String toString() {  
        return "<" + x + ", " + y + ">";  
    }  
}
```

Si no se programa el método en un cierto objeto, se hereda el de su superior en la jerarquía de herencia, en último caso, el definido en la clase `Object`, que imprime el nombre de la clase y el `hashCode` del objeto en hexadecimal:

<i><b>java.lang.Object</b></i>
<pre>public class Object {     public String toString() {         return getClass().getName() +             "@" + Integer.toHexString(hashCode());     } }</pre>
<i><b>Ejemplo.java</b></i>
<pre>public class Ejemplo {     public static void main(String[] argumentos) {         Ejemplo ejemplo = new Ejemplo();         System.out.println(ejemplo);     } }</pre>
<pre>\$ java Ejemplo Ejemplo@107077e</pre>

## 171. *try catch finally* (palabras reservadas)

Ver “[excepciones](#)”, “[Exception](#)”, “[catch](#)” y “[finally](#)”.

Un bloque **try** puede terminarse con

- cero o más bloques **catch**, de los que se ejecuta el primero cuya excepción casa con la lanzada dentro del bloque **try**
- un bloque **finally**, opcional, que se ejecuta siempre al salir del bloque **try**, háyase ejecutado o no alguno de los bloques **catch**

```
try {
    ....
} catch (ExceptionA a) {           // opcional: 0, 1, 2, ...
    ....
} catch (ExceptionB b) {
    ....
} finally {                       // opcional: 0 o 1
    ....
}
```

## 172. *Underflow* (concepto)

En general, se dice de las situaciones en que se intentan sacar datos en un contenedor que no los tiene. Así es muy frecuente oír hablar de “*buffer underflow*” para indicar que no hay tantos datos en un *buffer*.

En aritmética se aplica cuando un valor no llega al mínimo previsto.

En números en coma flotante el *underflow* se puede producir en el exponente cuando éste es negativo. Java lo interpreta como valor despreciable, indistinguible de cero.

Double. MIN\_VALUE / 10 =  
4.90000e-324 / 10 = 0.00000

Ver "[desbordamiento](#)".

### 173. Unicode (concepto)

Convenio internacional para representar caracteres de múltiples idiomas en 16 bits. Es el convenio que usa java. Los detalles pueden verse en

<http://www.unicode.org/charts/>

La siguiente tabla recoge los códigos que nos afectan en el suroeste de Europa:

<b>ISO-Latin-1</b>		
<b>decimal</b>	<b>carácter</b>	<b>hexadecimal</b>
32 a 47	! " # \$ % ' ( ) * + , - . /	20 a 2F
48 a 63	0 1 2 3 4 5 6 7 8 9 : ; < = > ?	30 a 3F
64 a 79	@ A B C D E F G H I J K L M N O	40 a 4F
80 a 95	P Q R S T U V W X Y Z [ \ ] ^ _	50 a 5F
96 a 111	` a b c d e f g h i j k l m n o	60 a 6F
112 a 126	p q r s t u v w x y z {   } ~	70 a 7F
160 a 175	ı ċ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯	A0 a AF
176 a 191	° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿	B0 a BF
192 a 207	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï	C0 a CF
208 a 223	Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß	D0 a DF
224 a 239	à á â ã ä å æ ç è é ê ë ì í î ï	E0 a EF
240 a 255	ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ	F0 a FF

No se "ve" en la tabla; pero el código 32 (20 en hexadecimal) corresponde al carácter "espacio en blanco".

No se ve en la tabla; pero el código hexadecimal 7F corresponde a la tecla DEL ("borrado").

Java permite expresar los caracteres usando el código Unicode hexadecimal. Por ejemplo, el carácter 'A' que tiene el código hexadecimal 41 se puede escribir

'A' → '\u0041'

lo que es perfectamente absurdo para caracteres normales; pero puede ser útil para los caracteres que no suelen aparecer en nuestros teclados.

Se pueden escribir cadenas de caracteres (String) con el mismo convenio.



"José" → "\u004a\u006f\u0073\u00e8"

Vea "[codificación de caracteres](#)" donde se explica cómo se utilizan bytes (8 bits) para guardar caracteres (16 bits).

## 174. Upcasting (concepto)

Se dice cuando el contenido de una variable de tipo B se asigna a una variable de tipo A, siendo B subclase de A.

```
class B extends A { ... }  
B b = ...;  
A a = b;
```

Siempre es posible.

Ver "[sustitución](#)".

Ver "[casting](#)".

## 175. Variables [variables] (concepto)

Son contenedores de valores, bien de [tipos primitivos](#) (números, caracteres o booleanos), bien objetos creados por el programador.

Una variable tiene un nombre que es un identificador que permite dirigirse a ella sin ambigüedad.

Si cada variable tiene un nombre exclusivo, no hay ambigüedad alguna. Pero si varias variables se denominan igual, hay que aclarar en cada momento a cual nos referimos. Ver "[ámbitos](#)". Java no permite variables homónimas en el mismo ámbito; pero si pueden existir en ámbitos diferentes.

Lo primero que hay que hacer con una variables es [declararla](#). En la declaración se indica

- el tipo de valores que puede contener (siempre)
- el identificador que la denomina (siempre)
- el valor inicial que va a contener (opcional)  
(si no se indica un valor inicial, este depende de tipo de la variable)
- el tipo de [visibilidad](#) de que va a disfrutar: **public**, **protected**, **private** (opcional)  
(si no se indica el tipo de accesibilidad, se considera "accesible en el paquete")
- si es modificable o no: [final](#) (opcional)  
(Una variable sólo es final si se indica explícitamente. Si no, será modificable.)

### actualización ([asignación](#))

---

El valor contenido en una variable se puede cambiar por medio de sentencias de asignación que eliminan el valor anterior y colocan uno nuevo:

<i><b>código .java</b></i>	<i><b>contenido</b></i>
<code>int x;</code>	0
<code>x = 5;</code>	5

<b>código .java</b>	<b>contenido</b>
<code>x = x + 1;</code>	6

En general, una sentencia de asignación tiene este formato

```
variable = expresión ;
```

La ejecución consiste en evaluar la expresión y almacenar el valor resultante en la variable.

Si la expresión se refiere a la misma variable, primero se extrae el valor de la variable, luego se calcula la expresión y, por último, se almacena el valor resultante en la variable.

Si se quiere asignar el mismo valor a varias variables, se pueden acumular sintácticamente:

```
v1 = v2 = v3 = expresión;
```

todas las variables citadas toman el mismo valor que es el de la expresión.

Las variables "[final](#)" no pueden ser actualizadas. Esto quiere decir que sólo se les puede asignar un valor una sólo vez, bien al declararlas, bien en el constructor. Es más, java impone que se les asigne un valor en uno u otro momento; de lo contrario se quejará de que la variable no está inicializada.

### **ámbito: accesibilidad**

---

Las variables se pueden utilizar dentro de un cierto [ámbito](#). El ámbito de accesibilidad depende del tipo de variable.

#### **variables de clase (static) y de objeto**

Siempre se pueden utilizar dentro del ámbito definido por las llaves { ... } que delimitan el cuerpo de la clase. Este ámbito puede verse ampliado por los calificativos **public** (se pueden usar desde cualquier sitio), "*de paquete*" (se pueden usar dentro del mismo paquete) y **protected** (se pueden usar en el mismo paquete y en [subclases](#)).

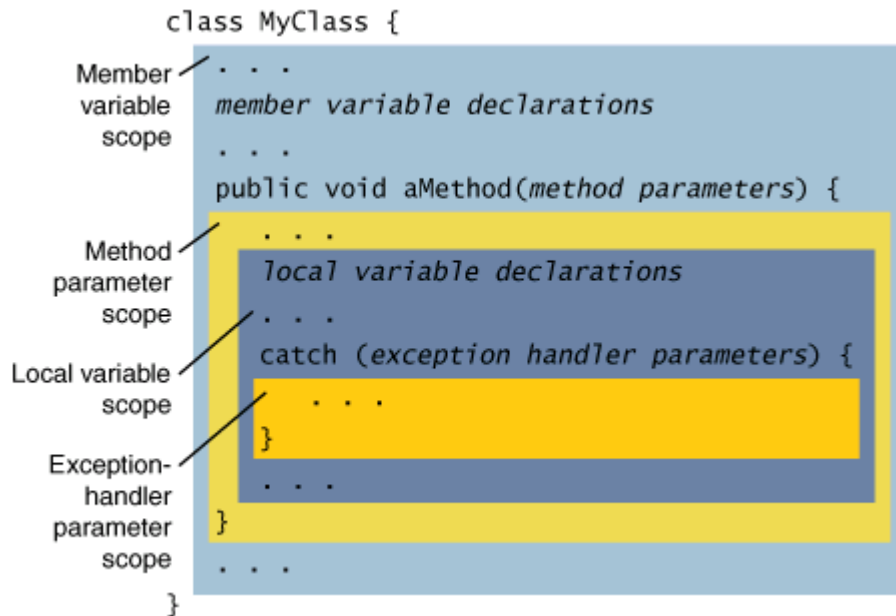
Un detalle: las variables de objeto no pueden usarse en métodos de clase (**static**).

#### **argumentos formales (de los métodos)**

Se pueden utilizar dentro del cuerpo del [método](#).

#### **variables locales (en un bloque)**

Se pueden utilizar dentro del bloque, entre el punto en el que se definen y la llave final que termina el bloque.



## ciclo de vida: creación y destrucción

---

Las variables se crean en un cierto momento y desaparecen en otro momento dado. A partir de la creación se pueden utilizar para almacenar valores. Cuando se destruyen, desaparece la posibilidad de acceder a los valores que contenían. Los detalles dependen de cada tipo de variable.

### variables de clase ([static](#))

Se crean al arrancar el programa y se destruyen cuando termina el programa. Es decir, siempre están disponibles.

### variables de objeto

Se crean al crear el objeto ([new](#)) y desaparecen cuando el objeto deja de utilizarse.

### argumentos formales (de los [métodos](#))

Se crean cada vez que se llama al método. Si hay 20 llamadas, se crean 20 veces. Si un método se llama a sí mismo (recursión), se generan nuevas variables en el nuevo método. Desaparecen cuando el método devuelve su resultado.

### variables locales (en un bloque)

Se crean cada vez que se ejecuta al bloque, pero sólo a partir del punto en el que se declaran. Desaparecen cuando el bloque termina.

Un caso particular, especialmente frecuente es el de las variables en el cuerpo de un [método](#), que siguen las mismas reglas. Se crean cada vez que se ejecuta al método, pero sólo a partir del punto en el que se declaran. Si hay 20 llamadas, se crean 20 veces. Si un método se llama a sí mismo (recursión), se generan nuevas variables en el nuevo método. Desaparecen cuando el método devuelve su resultado.

## de clase

---

Son las que se declaran [static](#) en el cuerpo de una clase.

Ver "[campos de clase](#)".

Ver "[ámbitos de visibilidad](#)".

Ver “[creación y destrucción](#)”.

Ver “[bloques static](#)”.

## invariantes

---

Se dice de las variables que se declaran “**final**”.

Pueden haber campos invariantes. Se caracterizan porque hay que darles un valor inicial durante la construcción del objeto, bien en la misma línea de declaración, bien en el constructor. Una vez asignado un valor, este no puede cambiarse nunca jamás.

```
public class Circulo {
    private final double radio;

    public Circulo (double radio) {
        this.radio = radio;           // queda fijo para siempre
    }
}
```

También pueden ser ‘**final**’ variables locales o parámetros cuyo valor no se puede modificar tras la asignación inicial.

```
public int getFilaLibre(final int columna) {
    for (int fila = FILAS - 1; fila >= 0; fila--) {
        if (array[columna][fila] == null) {
            return fila;
        }
    }
    return -1;
}
```

```
public Ficha getFicha(Posicion posicion) {
    final int columna = posicion.getColumna();
    final int fila = posicion.getFila();
    return array[columna][fila];
}
```

Ver “[Constantes](#)”.

## declaración

---

En la declaración de una constante

- se indica su nombre (identificador)
- su tipo (primitivo o clase). Este tipo no podrá ser modificado en el futuro. No obstante,
  - si el tipo es [primitivo](#): véase [promoción](#) y [reducción](#).
  - si el tipo es un objeto: véase “[casting](#)”.
- [opcionalmente] se carga un valor inicial
- se califica como **public**, “*de paquete*”, **protected** o **private**. Esta calificación no podrá modificarse en el futuro.
- [opcionalmente] se marca como **final**, indicando que su valor no podrá ser modificado (tras su carga inicial).

## finales

---

Se dice de aquellas variables marcadas como [final](#), indicando que su valor no podrá ser modificado (tras su carga inicial).

Ver “[invariantes](#)”.

## inicialización

---

Se dice de la carga del primer valor en una variable. Se puede hacer aprovechando su declaración o posteriormente. A veces el compilador puede detectar de que se intenta utilizar una variable no inicializada, lo que suele ser síntoma de error u olvido.

### campos (variables) de clase ([static](#))

Conviene inicializarlas junto a la declaración o en un [bloque static](#).

Si no se dice nada, se inicializan con valor 0 o null.

### campos (variables) de objeto

Conviene inicializarlas junto a la declaración o en el constructor.

Si no se dice nada, se inicializan con valor 0 o null.

### argumentos formales (de los [métodos](#))

Se inicializan con los argumentos reales aportados en la llamada al método.

### variables locales (en un bloque)

Conviene inicializarlas junto a su declaración. O, visto al revés, conviene no declararlas hasta que se conoce su valor inicial.

No se pueden utilizar si no se les asigna algún valor.

Ver “[inicialización](#)”.

## locales (en un bloque)

---

Cuando el bloque es el cuerpo de un [método](#), también se denominan “variables de método” o “variables automáticas”.

Son aquellas que se se declaran en un bloque.

Se crean cuando se declaran y se destruyen cuando el bloque termina. Cada vez que se ejecuta el bloque se crea una nueva variable.

Si existen simultáneamente varias llamadas al bloque, existirán simultáneamente varias variables, todas diferentes entre sí. Esto ocurre en métodos recursivos.

### EJEMPLO

#### Búsqueda binaria (de diccionario).

Dado un array ordenado de enteros, encontrar en qué posición se encuentra el número N.

Devuelve la posición de N en el array. Devuelve -1 si N no está en el array.

```
public int busca (int n, int[] datos) {  
    return busquedaBinaria(n, datos, 0, datos.length);  
}  
  
// busca en el rango a..z  
private int busquedaBinaria(int n, int[] datos, int a, int z) {
```

```

if (a > z)
    return -1;
int medio = (a + z) / 2;
int valor = datos[medio];
if (valor == n)
    return medio; // encontrado
if (valor < n)
    return busquedaBinaria(n, datos, a, medio-1); // a la izquierda
else
    return busquedaBinaria(n, datos, medio+1, z); // a la derecha
}

```

## de objeto

Ver “[campos de clase](#)”.

Ver “[ámbitos de visibilidad](#)”.

Ver “[creación y destrucción](#)”.

## 176. Visibilidad [scope]

**Ámbito** o zona del código en el que un elemento de Java puede referenciarse por su nombre.

Java define 4 modificadores fundamentales que califican a [clases](#), [métodos](#) y [campos](#):

### private

el elemento sólo es accesible dentro del fichero en el que se define

### de paquete

el elemento sólo es accesible dentro del paquete en el que se define (clases en el mismo directorio).

### protected

el elemento es accesible dentro del paquete en el que se define y, además, en las [subclases](#).

### public

el elemento es accesible desde cualquier sitio.

Las reglas básicas se pueden resumir en la siguiente tabla

modificador	<b>visibilidad</b>			
	fichero	paquete (directorio)	subclases (extends)	cuaquiera
<b>private</b>	SÍ	NO	NO	NO
<b>de paquete</b> (package local)	SÍ	SÍ	NO	NO
<b>protected</b>	SÍ	SÍ	SÍ	NO
<b>public</b>	SÍ	SÍ	SÍ	SÍ

Las clases pueden ser “**public**” (para todo el mundo), “de paquete” (restringidas a un paquete) o “**private**” (restringidas a un fichero).

Los campos se recomienda que sean siempre “**private**” (restringidos a un fichero).

Los métodos suelen aprovechar frecuentemente las facilidades java de control de visibilidad.

El siguiente ejemplo recoge toda la casuística y muestra qué es posible y que no:

<b>a/A.java</b>	<pre>package a;  public class A {      private void metodoPrivate() {     }      /* de paquete */ void metodoFriendly() {     }      protected void metodoProtected() {     }      public void metodoPublic() {     }      void acceso(A a) {         a.metodoPrivate();         a.metodoFriendly();         a.metodoProtected();         a.metodoPublic();     } }</pre>
<b>a/A2.java</b>	<pre>package a;  public class A2 {      void acceso(A a) {         a.metodoPrivate();    // mal         a.metodoFriendly();         a.metodoProtected();         a.metodoPublic();     } }</pre>
<b>b/B.java</b>	<pre>package b;  import a.A;  public class B {      void acceso(A a) {</pre>

	<pre> a.metodoPrivate();    // mal a.metodoFriendly();   // mal a.metodoProtected();  // mal a.metodoPublic();     } } </pre>
<b>c/C.java</b>	<pre> package c;  import a.A;  public class C extends A {      void acceso(A a) {         a.metodoPrivate();    // mal         a.metodoFriendly();   // mal         a.metodoProtected();  // mal         a.metodoPublic();     }      void acceso(C c) {         c.metodoPrivate();    // mal         c.metodoFriendly();   // mal         c.metodoProtected();         c.metodoPublic();     } } </pre>

Nótese la diferencia en la clase C que extiende a la clase A, que puede acceder a los métodos protegidos de A siempre y cuando lo haga como método heredado, que no como método de A.

Cuando se intenta compilar, el compilador detecta y reporta los siguientes errores semánticos:

```

$ javac a/*.java b/*.java c/*.java
a/A2.java:6: metodoPrivate() has private access in a.A
    a.metodoPrivate();    // mal
    ^
b/B.java:8: metodoPrivate() has private access in a.A
    a.metodoPrivate();    // mal
    ^
b/B.java:9: metodoFriendly() is not public in a.A;
                                cannot be accessed from outside package
    a.metodoFriendly();    // mal
    ^
b/B.java:10: metodoProtected() has protected access in a.A
    a.metodoProtected();  // mal
    ^
c/C.java:8: metodoPrivate() has private access in a.A
    a.metodoPrivate();    // mal
    ^
c/C.java:9: metodoFriendly() is not public in a.A;
                                cannot be accessed from outside package
    a.metodoFriendly();    // mal
    ^

```



```

c/C.java:10: metodoProtected() has protected access in a.A
  a.metodoProtected(); // mal
  ^
c/C.java:15: metodoPrivate() has private access in a.A
  c.metodoPrivate(); // mal
  ^
c/C.java:16: metodoFriendly() is not public in a.A;
               cannot be accessed from outside package
  c.metodoFriendly(); // mal
  ^
9 errors

```

## 177. void (palabra reservada)

Se usa como tipo del resultado de un método para indicar que no devuelve nada.

Ver "[métodos](#)".

## 178. while (palabra reservada)

Se usa para construir [bucles](#) que se ejecutan cero o más veces.

java	flujo
<pre> while (condición)     sentencia;  while (condición) {     sentencia 1;     sentencia 2;     ...     sentencia ...; } </pre>	<pre> graph TD     Entrada(( )) --&gt; Condicion{condición}     Condicion -- true --&gt; Sentencia[sentencia;]     Sentencia --&gt; Condicion     Condicion -- false --&gt; Salida(( )) </pre>

```

while (isHambriento())
    comer();

```

```

while (semaforo.isActivado()) {
    semaforo.setColor(ROJO);
    semaforo.setColor(VERDE);
    semaforo.setColor(AMARILLO);
}

```

```

int factorial(int n) {
    int fact = 1;
    while (n > 0) {
        fact *= n;
        n--;
    }
}

```

```
}    return fact;
```

Si el contenido del bucle es una única sentencia, se pueden obviar las llaves. No es obligatorio: siempre pueden ponerse llaves.

La palabra reservada “while” también se emplea en bucles “[do ... while](#)”.

Ver “[bucles](#)”.

# Las Bibliotecas de Java

Java proporciona un amplio conjunto de clases para agilizar tareas comunes.

Ver <http://java.sun.com/javase/6/docs/api/index.html>

En esta sección se recogen de forma no exhaustiva algunas de las clases de uso más frecuente.

## 1. ***ArrayList<E>* (clase) *java.util.ArrayList<E>***

Ver [List<E> \(interface\) \*java.util.List<E>\*](#)

## 2. ***Arrays* (clase) *java.util.Arrays***

Esta clase proporciona múltiples métodos para manipular [arrays](#) sobre múltiples tipos de datos **T**: primitivos (enteros, reales, caracteres y booleanos) y objetos.

**static void fill(T[] array, T valor)**

Llena el array con el valor indicado: todos los elementos iguales.

**static boolean equals(T[] a1, T[] a2)**

Chequea si los arrays son idénticos, comprobando valores primitivos (==) y referencias (con equals).

**static boolean deepEquals(Object[] a1, Object[] a2)**

Chequea si los arrays son idénticos, comprobando valores primitivos (==) y referencias (con equals). Además, si el array es multidimensional, profundiza en las sucesivas dimensiones.

**static String toString(T[] datos)**

Genera una cadena para impresión.

**static String deepToString(T[] datos)**

Genera una cadena para impresión, incluso si se trata de un array multidimensional.

**static T[] copyOf(T[] datos, int n)**

Genera una copia de los datos. Si n es mayor que datos.length, rellena con null. Si n es menor que datos.length, se ignora el exceso (es decir, trunca).

**static T[] copyOfRange(T[] datos, int desde, int hasta)**

Copia un segmento de los datos.

**static int binarySearch(T[] datos, T clave)**

Busca en qué posición del array datos se encuentra la clave dada. El array debe estar ordenado.

**static void sort(T[] datos)**

Ordena el array.

**static void sort(T[] datos, int desde, int hasta)**

Ordena el array entre las posiciones indicadas.

Ver "[class Arrays](#)"

### 3. **Boolean (clase) java.lang.Boolean**

Clase que sirve para tratar los [booleanos](#) como objetos, además de proporcionar una serie de métodos útiles:

**Boolean (boolean b)**

Constructor. Pasa de tipo primitivo a objeto.

**Boolean (String s)**

usa como boolean el resultado de:  
`s.equalsIgnoreCase("true")`

**static boolean parseBoolean(String s)**  
`return s.equalsIgnoreCase("true");`

**static String toString(boolean b)**  
pasa el valor b a una String

### 4. **Calendar (clase)**

Es una clase muy general para manejar fechas y horas.

En parte reemplaza las funciones de [Date](#), y además añade muchísima flexibilidad.

La clase carece de constructores públicos, así que la forma de crear objetos de esta clase es por medio de una factoría

```
Calendar ahora = Calendar.getInstance();  
Calendar ahora = Calendar.getInstance(Locale locale);  
Calendar ahora = Calendar.getInstance(TimeZone zone);  
Calendar ahora = Calendar.getInstance(TimeZone zone,  
                                     Locale locale);
```

La primera factoría utiliza la configuración por defecto de idioma del ordenador donde se ejecuta y el huso horario en el que esté configurado.

Por ejemplo

<b>Locale</b>
<pre>Locale aqui = Locale.getDefault(); Locale spanish = new Locale("es"); Locale spanish = new Locale("es", "ES"); Locale mexico = new Locale("es", "MX");</pre>
<b>TimeZone</b>
<pre>TimeZone aqui = TimeZone.getDefault(); TimeZone madrid = new TimeZone("Europe/Madrid");</pre>

Una vez construido el Calendar, podemos usar métodos getter y setter:

```
int get(int campo)  
Date getTime()  
long getTimeInMillis()
```

```

void set(int campo, int valor)
void set(int año, int mes, int día)
void set(int año, int mes, int día, int hora, int minuto)
void set(int año, int mes, int día,
        int hora, int minuto, int segundo)
void setTime(Date date)
void setTimeInMillis(long milisegundos)

```

El siguiente método añade una cierta cantidad a un determinado campo:

```
void add(int campo, int valor);
```

Puede ser útil para averiguar si un año es bisiesto:

```
boolean isLeapYear(int año);
```

Para imprimir fechas puede recurrir a la clase [java.util.Formatter](#) o a clases ad-hoc como son

```

java.util.DateFormat
java.util.SimpleDateFormat

```

## Campos

Los métodos que usan un campo como argumento, recurren a una serie de constantes definidas en la clase Calendar. Quizás lo mejor es verlo con un ejemplo:

### *Comparador.java*

```

public static void muestraCampos() {
    Calendar ahora = Calendar.getInstance();
    System.out.println(ahora.getTime());
    System.out.println("año: " +
        ahora.get(Calendar.YEAR));
    System.out.println("mes: " +
        ahora.get(Calendar.MONTH));
    System.out.println("día del mes: " +
        ahora.get(Calendar.DAY_OF_MONTH));
    System.out.println("día del año: " +
        ahora.get(Calendar.DAY_OF_YEAR));
    System.out.println("día de la semana: " +
        ahora.get(Calendar.DAY_OF_WEEK));
    System.out.println("hora: " +
        ahora.get(Calendar.HOUR));
    System.out.println("hora del día: " +
        ahora.get(Calendar.HOUR_OF_DAY));
    System.out.println("minutos: " +
        ahora.get(Calendar.MINUTE));
    System.out.println("segundos: " +
        ahora.get(Calendar.SECOND));
    System.out.println("milisegundos: " +
        ahora.get(Calendar.MILLISECOND));
}

```

### **Comparador.java**

```
Fri Mar 26 16:10:14 CET 2010
año: 2010
mes: 2
día del mes: 26
día del año: 85
día de la semana: 6
hora: 4
hora del día: 16
minutos: 10
segundos: 14
milisegundos: 310
```

Nótese que los meses se cuentan desde 0 (enero es el 0, marzo es el 2) y que el día de la semana comienza en domingo (domingo es el 0, viernes es el 6).

#### **Ejemplo: código para marcar una fecha**

---

Navidad de este año:

```
Calendar navidad = Calendar.getInstance();
navidad.set(Calendar.MONTH, Calendar.DECEMBER);
navidad.set(Calendar.DAY_OF_MONTH, 25);
System.out.println(navidad.getTime());
```

Navidad del año 2000:

```
Calendar navidad = Calendar.getInstance();
navidad.set(Calendar.YEAR, 2000);
navidad.set(Calendar.MONTH, Calendar.DECEMBER);
navidad.set(Calendar.DAY_OF_MONTH, 25);
System.out.println(navidad.getTime());
```

Navidad del año que viene:

```
Calendar navidad = Calendar.getInstance();
navidad.add(Calendar.YEAR, 1);
navidad.set(Calendar.MONTH, Calendar.DECEMBER);
navidad.set(Calendar.DAY_OF_MONTH, 25);
System.out.println(navidad.getTime());
```

#### **Ejemplo: código para calcular la distancia entre dos fechas**

---

Ferdie Adoboe (EEUU) corrió los 100 metros hacia atrás en 14 segundos en Amherst, el 28 de julio de 1983. Eso fue hace ... días:

```
Calendar record = Calendar.getInstance();
record.set(1983, Calendar.JULY, 28);
System.out.println(record.getTime());
Calendar hoy = Calendar.getInstance();
System.out.println(hoy.getTime());
long deltaMS = hoy.getTimeInMillis() -
record.getTimeInMillis();
```

```
long deltaDIAS = deltaMS / 1000 / 60 / 60 / 24;
System.out.println("hace " + deltaDIAS + " días");
```

Si mi amigo nació el 13 de octubre de 1990, hoy tiene ... años:

```
Calendar nace = Calendar.getInstance();
nace.set(1990, Calendar.OCTOBER, 13);
System.out.println(nace.getTime());
Calendar hoy = Calendar.getInstance();
System.out.println(hoy.getTime());
int cumplidos = hoy.get(Calendar.YEAR) -
nace.get(Calendar.YEAR);
// calculamos la fecha de su cumpleaños este año
nace.add(Calendar.YEAR, cumplidos);
if (hoy.before(nace))
    cumplidos -= 1;
System.out.println("tiene " + cumplidos + " años");
```

## 5. **Character (clase) *java.lang.Character***

Además de servir como clase envoltorio de caracteres, proporciona una serie de métodos útiles para analizar texto:

```
boolean Character.isDigit (char ch)
boolean Character.isLetter (char ch)
boolean Character.isLetterOrDigit (char ch)
boolean Character.isLowerCase (char ch)
boolean Character.isUpperCase (char ch)
boolean Character.isSpaceChar (char ch)
boolean Character.isWhitespace (char ch)
char Character.toLowerCase (char ch)
char Character.toUpperCase (char ch)
```

## 6. **Collator (clase) *java.text.Collator***

Las comparaciones entre String se limitan a usar el orden numérico de los códigos Unicode, lo que a veces choca con el orden al que estamos acostumbrados en los diccionarios, el orden lexicográfico, que depende de cada idioma. El siguiente programa muestra como usar la clase *java.text.Collator* para comparar cadenas con letras mayúsculas, minúsculas y caracteres acentuados:

<b>Comparador.java</b>
<pre>import java.text.Collator;  public class Comparador {     private static void compara(String s1, String s2) {         Collator collator = Collator.getInstance();         System.out.println("compareTo -&gt; " +             s1.compareTo(s2));         System.out.println("collator -&gt; " +             collator.compare(s1, s2));     }      public static void main(String[] argumentos) {         compara("Iñaki", "alberto");         compara("Iñaki", "Iván");     } }</pre>
<pre>\$ java Comparador compareTo -&gt; -24 collator -&gt; 1 compareTo -&gt; 123 collator -&gt; -1</pre>

## 7. **Collection<E> (interface) java.util.Collection<E>**

Una serie de estructuras de datos proporcionadas por la biblioteca estándar de java.

Las más relevantes son:

- [interface List<E>](#) - listas de cosas, respetando el orden y admitiendo duplicados
- [interface Set<E>](#) - conjuntos de cosas, sin duplicados

Todas las clases que implementan esta interface proporciona los siguientes métodos, al menos:

<b>interface Collection&lt;E&gt; extends Iterable&lt;E&gt;</b>
<pre><b>boolean add(E elemento)</b>     añade un elemento <b>boolean addAll(Collection&lt;E&gt; c)</b>     añade todos los elementos, uno a uno <b>void clear()</b>     vacía la lista <b>boolean contains(E elemento)</b>     true si hay en la lista un elemento "equals" al indicado <b>boolean containsAll(Collection&lt;E&gt; elementos)</b></pre>



<i><b>interface Collection&lt;E&gt; extends Iterable&lt;E&gt;</b></i>	
	true si hay en la lista un elemento “equals” para cada uno de los elementos
<b>boolean isEmpty()</b>	true si no hay elementos
<b>Iterator&lt;E&gt; iterator()</b>	devuelve un iterador sobre los elementos de la lista
<b>boolean remove(E elemento)</b>	elimina el primer elemento que encuentre en la lista y que sea “equals” el indicado; devuelve true si se elimina algún elemento
<b>boolean removeAll(Collection c)</b>	elimina todos los elementos que sean “equal” a alguno de los pasados como argumento
<b>int size()</b>	devuelve el número de elementos en la lista
<b>Object[] toArray()</b>	devuelve un array cargado con los elementos de la lista
<b>T[] toArray(T[] a)</b>	devuelve un array cargado con los elementos de la lista

El método `toArray()` devuelve un array nuevo de longitud igual al número de elementos en la colección.

El método `toArray(T[] a)` intenta utilizar el array “a” que se le pasa como argumento:

- si el tamaño del array es igual al número de elementos en la colección, el array “a” se utiliza tal cual para cargar los elementos y devolver el resultado
- si el tamaño del array “a” es mayor que el número de elementos en la colección, los datos se cargan en las primeras posiciones del array y en la primera posición que “sobra” se mete NULL
- si el tamaño del array “a” es insuficiente, se crea un nuevo array de tamaño igual al número de elementos en la colección y se devuelve como resultado

Consecuencia de esta peculiar forma de tratar el argumento, a menudo se utiliza esta construcción para generar un array de tipo deseado y de tamaño exactamente igual al número de elementos:

```
T[] resultado = coleccion.toArray( new T[0] );
```

En otros casos, si no queremos que java se dedique a crear arrays, creamos uno suficientemente grande y le decimos que lo llene hasta donde necesite

```
T[] resultado = new T[1000];
coleccion.toArray( resultado );
```

## **8. Comparable<T> (interface) java.lang.Comparable<T>**

Interfaz que sirve para marcar clases que disfrutan de una relación de orden total, lo que se traduce en que las clases que implementen esta interfaz deben proporcionar un método

```
int compareTo(T x)
```

que compara THIS con x, devolviendo

- un número negativo si `this < x`
- cero si `this = x`
- un número positivo si `this > x`

Los objetos que implementan esta interface se pueden usar en [conjuntos ordenados](#) y [mapas ordenados](#).

### Ejemplo

---

Ordena primero por nombre, después por teléfono (si el nombre es el mismo) y por último por dirección (si nombre y teléfono son iguales).

```
public class Contacto
    implements Comparable<Contacto> {
    private String nombre;
    private String telefono;
    private String direccion;

    public int compareTo(Contacto contacto) {
        int comparison = nombre.compareTo(contacto.getNombre());
        if (comparison != 0) {
            return comparison;
        }
        comparison = telefono.compareTo(contacto.getTelefono());
        if (comparison != 0) {
            return comparison;
        }
        return direccion.compareTo(contacto.getDireccion());
    }
}
```

## 9. **Comparator<T> (interface) java.util.Comparator<T>**

Interfaz que sirve para marcar clases que disfrutan de una relación de orden total, lo que se traduce en que las clases que implementen esta interfaz deben proporcionar un método

```
int compare(T x1, T x2)
```

que compara x1 con x2, devolviendo

- un número negativo si `x1 < x2`
- cero si `x1 = x2`
- un número positivo si `x1 > x2`

Los objetos que implementan esta interface se pueden usar en [conjuntos ordenados](#) y [mapas ordenados](#).

### Ejemplo

---

Esta clase compara String teniendo en cuenta, únicamente, su longitud:

```

public class Ejemplo
    implements Comparator<String> {

    public int compare(String s1, String s2) {
        int n1= s1.length();
        int n2= s2.length();
        return n1 - n2;
    }
}

```

## 10. *Date (clase) java.util.Date*

Sirve para contener fechas. Concretamente, fecha y hora con precisión de milisegundos.

En versiones antiguas de java, los objetos Date se usaban para todo tipo de actividad relacionada con fechas; pero resultaba poco flexible a efectos de internacionalización y se ha sustituido por [Calendar](#), que es más completa y flexible.

Hay dos constructores

```

Date ahora = new Date(); // fecha y hora actual
Date date = new Date(milisegundos);

```

donde 'milisegundos' es un intervalo de tiempo medido en milisegundos y empezando a contar el 1 de enero de 1970 (UTC). Así es, por ejemplo, el valor que devuelve

```
System.currentTimeMillis()
```

Otra forma de crear un objeto con la fecha y hora actual:

```
Date ahora = new Date(System.currentTimeMillis());
```

Quedan asimismo algunos métodos de utilidad

<i>java.util.Date</i>	
<pre> boolean after(Date d) boolean before(Date d) int compareTo(Date d) </pre>	si una fecha y hora (this) es anterior, posterior o igual a otra (d)
<pre>long getTime()</pre>	devuelve el tiempo transcurrido desde el 1 de enero de 1970 hasta this; en milisegundos
<pre>void setTime(long mili)</pre>	como el constructor, fija una fecha y hora
<pre>String toString()</pre>	<pre>Fri Mar 26 09:14:19 CET 2010</pre> <p>que es el viernes, 26 de marzo de 2010, a las nueve y cuarto, hora de Centroeuropa</p>

Ver "[java.util.Calendar](#)".

Ver "[java.util.Formatter](#)".

## 11. Double (clase)

Además de servir como objeto contenedor de valores reales ([double](#)), proporciona algunos métodos útiles:

```
Double(double value)
Double(String s)
static double parseDouble(String s)
static String toString(double v)
```

## 12. Enumeration<E> (interface) java.util Enumeration

Interfaz normalizada para recorrer ordenadamente los elementos de una colección.

Define 2 métodos a implementar:

<i>java.util Enumeration&lt;E&gt;</i>
<pre>public interface Enumeration&lt;E&gt; {     /**      * @return TRUE si la siguiente llamada a next()      * devolverá un elemento      */     public boolean hasMoreElements();      /**      * @return el siguiente elemento de la colección      * @throws <b>NoSuchElementException</b> - si no hay elemento que devolver      */     public E nextElement(); }</pre>

Con los iteradores se suelen construir los siguientes tipos de bucles

<i>bucles con iteradores</i>
<pre>for (Enumeration&lt;E&gt; ite = ...; ite.hasMoreElements(); ) {     E elemento = ite.nextElement();     ... }</pre>
<pre>Enumeration&lt;E&gt; ite = ...; while (ite.hasMoreElements ()) {     E elemento = ite.nextElement ();     ... }</pre>

La interface “Enumeration” ha sido revisada y ampliada por “[Iterator](#)”.

## 13. EnumSet (clase) java.util.EnumSet

Es un refinamiento de “[java.util.Set](#)” especializado en trabajar con tipos enumerados.

Dado un tipo enumerado

```
Dias enum { Lunes, Martes, Miércoles, Jueves,
            Viernes, Sábado, Domingo }
```

podemos ...

```

EnumSet<Dias> todos = EnumSet.allOf(Dias.class);
EnumSet<Dias> ninguno = EnumSet.noneOf(Dias.class);
EnumSet<Dias> festivos = EnumSet.of(Dias.Sábado, Dias.Domingo);
EnumSet<Dias> laborables =
    EnumSet.range(Dias.Lunes, Dias.Viernes);
EnumSet<Dias> conR =
    EnumSet.of(Dias.Martes, Dias.Miércoles, Dias.Viernes);
EnumSet<Dias> sinR = EnumSet.complementOf(conR);

```

Es fácil iterar sobre los elementos de un EnumSet

```
for (Dia dia: laborables) { ... }
```

## 14. File (clase) java.io.File

Clase java para referirse a ficheros y directorios en el sistema de ficheros del ordenador.

Un objeto File se puede construir a partir del nombre o ruta completa, o a partir del directorio en el que se encuentra:

```

File fichero = new File("C:\Users\yo\Documents\fichero.txt");
File directorio = new File("C:\Users\yo\Documents");
File otro = new File(directorio, "otro.txt");

```

La clase File proporciona muchos métodos, entre los que cabe destacar:

java.io.File	
boolean canRead()	TRUE si el fichero se puede leer
boolean canWrite()	TRUE si el fichero se puede escribir
boolean delete()	elimina el fichero; devuelve FALSE si no puede eliminarlo
boolean exists()	TRUE si el fichero existe
String getAbsolutePath()	devuelve la ruta completa
String getCanonicalPath()	devuelve la ruta completa
String getName()	el nombre del fichero, sin ruta
String getParent()	la ruta del 'padre' o directorio en el que se encuentra
File getParentFile()	el 'padre' o directorio en el que se encuentra
boolean isDirectorio()	TRUE si es un directorio
boolean isFile()	TRUE si no es un directorio
long length()	tamaño del fichero en bytes
String[] list()	si se trata de un directorio, un array con los nombres de los ficheros que contiene
File[] listFiles()	si se trata de un directorio, un array con los ficheros que contiene
boolean mkdir()	crea el directorio si no existe
boolean mkdirs()	crea todos los directorios que haga falta, si no existen
boolean renameTo(File nuevo)	cambio de nombre

Es muy frecuente usar los ficheros para leer o escribir. Java proporciona varias clases para facilitar estas operaciones:

	bytes	caracteres
lectura	<a href="#">InputStream</a>	<a href="#">Reader</a>
escritura	<a href="#">OutputStream</a>	<a href="#">Writer</a>

En el caso de trabajar con caracteres es importante tener en cuenta la "[codificación en bytes](#)".

## 15. **Formatter (clase) `java.util.Formatter`**

Esta clase permite generar [String](#) a partir de datos usando una especificación de formato.

La funcionalidad se presta a diferentes clases para un uso cómodo:

```
String s = String.format(formato, valores ...);
System.out.printf(formato, valores ...);
System.out.format(formato, valores ...);
```

### ejemplos

<i>uso de <code>format(formato, valores ...)</code></i>	
<i>formato, valores ...</i>	<i>String resultado</i>
"%d", 5	"5"
"%3d", 5	" 5"
"%.2f", Math.PI	"3,14"
"%8.4f", Math.PI	" 3,1416"
"%8.2feuros", 5.95	" 5,95euros"
"%,10.0f euros", 48000.95	" 48.001 euros"
"%s %s", "hola", "muchacho"	"hola muchacho"
"hola %s", "muchacho"	"hola muchacho"
"%5s", "hola"	" hola"
Date date= new Date(); "Fecha y hora: %tc", date	Fecha y hora: mar sep 13 09:18:18 CEST 2005
long date= System.currentTimeMillis(); "Fecha y hora: %tc", date	Fecha y hora: mar sep 13 09:18:18 CEST 2005
Date date= new Date(); "Son las %tH:%<tM", date	Son las 09:18

### ***uso de format(formato, valores ...)***

```
Date date= new Date();  
"Hoy es %tA, %<te de %<tB de %<tY", date  
  
Hoy es martes, 13 de septiembre de 2005
```

### **especificación de formato**

**% [ argumento \$ ] [ marca ] [ ancho ] [ . precision ] tipo**

donde las partes entre corchetes son opcionales.

#### **argumento \$**

De entre los valores que vienen a continuación, 1\$ se refiere al primer argumento, 2\$ al segundo y así sucesivamente. < se refiere al anterior.

Si no se especifica argumento, se van usando los valores presentes sucesivamente.

#### **marca**

Determina pequeñas variantes sobre la cadena generada:

Nada. Ajusta a la derecha, rellenando con blancos a la izquierda hasta alcanzar el ancho deseado.

Carácter blanco. Se inserta un blanco a la izquierda en valores positivos.

– Ajusta a la izquierda, rellenado con blancos a la derecha hasta alcanzar el ancho indicado.

# Formato alternativo.

+ Incluir siempre el signo de la cantidad numérica, sea positivo o negativo.

( Si el valor es negativo, colocarlo entre paréntesis.

0 Rellenar a la izquierda con ceros.

, Incluir separadores locales entre miles, millones, etc.

#### **ancho**

Indica la longitud mínima de la cadena generada. Si se necesita más espacio, la cadena será más larga. Si no, se rellena con blancos a la izquierda (salvo si se indica la marca '-' en cuyo caso se rellena por la derecha; o si se indica la marca '0' en cuyo caso se rellena con ceros).

#### **precisión**

%f Número de cifras decimales (tras la coma)

%e

%g Número de cifras significativas.

%b Ancho máximo. Se trunca la palabra.

%s Ancho máximo. Si la cadena es más larga, se trunca.

#### **tipo**

%n Fin de línea.

%%	Carácter '%'.
%s %S	Cadena de caracteres.
%d	Número entero: notación decimal.
%o	Número entero: notación octal.
%x %X	Número entero: notación hexadecimal.
%c %C	Número entero: como carácter.
%f	Número real, sin usar notación científica (sin exponente)
%e %E	Número real: notación científica (con exponente)
%g %G	Número real: como %f, excepto si se sale del ancho, en cuyo caso se salta a formato %e.
%b %B	Valor booleano.
%t %T	Fecha y hora.
%tA	día de la semana (ej. lunes)
%ta	día de la semana abreviado (ej. lun)
%tB	mes (ej. junio)
%tb	mes abreviado (ej. jun)
%tC	siglo: año / 100: 2 cifras
%tc	"%ta %tb %td %tT %tZ %tY"
%tD	"%tm/%td/%ty"
%td	día del mes: 01..31
%tE	tiempo en milisegundos desde el 1 de enero de 1970
%te	día del mes: 1..31
%tF	"%tY-%tm-%td"
%tH	hora: 00..23
%th	"%tb"
%tI	hora: 01-12
%tj	día del año: 001..366
%tk	hora: 0..23
%tL	milisegundos: 000..999
%tl	hora: 1..12
%tM	minuto: 00..59
%tm	mes: 01-12
%tN	nanosegundos (9 cifras)
%tp	"am" o "pm"
%Tp	"AM" o "PM"
%tQ	milisegundos desde el 1 de enero de 1970.



%tR     "%tH:%tM"  
 %tr     "%tI:%tM:%tS %Tp"  
 %tS     segundos: 00..59  
 %ts     segundos desde el 1 de enero de 1970  
 %tT     "%tH:%tM:%tS"  
 %tY     año: 2005  
 %ty     dos últimas cifras del año: 05  
 %tZ     zona horaria abreviada  
 %tz     zona horaria: diferencia respecto de GMT

Ver "[Números en notación local](#)".

## Strings

---

Ejemplos de uso habitual con String:

"[%s]", "cadena normal"	[cadena normal]
"[%10s]", "cadena normal"	[cadena normal]
"[%10.10s]", "cadena normal"	[cadena nor]
"[%10s]", "corta"	[       corta]
"[%-10s]", "corta"	[corta     ]
"[%10.3s]", "corta"	[       cor]
"[%-10.3s]", "corta"	[cor       ]
String format = String.format("[%0ds]%n", 10); format(format, "corta");	[       corta]

## Números en notación local

---

Ver "[Formatter](#)".

Cuando interesa escribir cantidades numéricas usando el convenio local de números (caracteres para separar grupos de 3 dígitos) se recurre a la siguiente notación:

```
String format(Locale locale, String format, Object ... args)
```

```
import java.util.*;

public class Decimales {

    public static void main(String[] args) {
        test("de", "DE", 1234.56);
        test("en", "US", 1234.56);
        test("es", "ES", 1234.56);
        test("fr", "FR", 1234.56);
    }
}
```

```

    test("it", "IT", 1234.56);
    test("pt", "PT", 1234.56);
}

private static void test(String idioma, String pais,
                        double n) {
    Locale locale= new Locale(idioma, pais);
    System.out.println(String.format(locale,
        "%s: %, .2f",  locale, n));
}
}

```

```

de_DE: 1.234,56
en_US: 1,234.56
es_ES: 1.234,56
fr_FR: 1á234,56
it_IT: 1.234,56
pt_PT: 1.234,56

```

## Fecha y hora

En general

```

Date date= ...
String.format(..., date)

```

<b>fecha y hora</b>	%tc %tZ %tz	mié oct 07 06:50:16 CEST 2009 CEST +0100
<b>fecha</b>	%tD %tF	10/07/09 2009-10-07
<b>día</b>	%tA %ta %td %te %tj	miércoles mié 01-31 1-31 001-366
<b>mes</b>	%tB %tb %th %tm	octubre oct oct 01-12
<b>año</b>	%tC %tY %ty	20 (de 2009) 2009 09 (de 2009)

hora	%tR %tr %tT	07:03 07:03:30 AM 07:03:30
hora	%tH %tI %tk %tl %tp %Tp	00-23 01-12 0-23 1-12 "am" o "pm" "AM" o "PM"
minutos	%tM	00-59
segundos	%tS %ts	00-59 segundos desde el 1.1.1970
milisegundos	%tL %tQ	000-999 milisegundos desde el 1.1.1970
nanosegundos	%tN	9 cifras

## 16. *HashMap<K, V>* (clase) *java.util.HashMap<K, V>*

Implementación de Map<K, V> que se centra en la rapidez de ejecución.

Ver [Map<K, V>](#) (interface) *java.util.Map<K, V>*

## 17. *HashSet<E>* (clase) *java.util.HashSet<E>*

Implementación de Set<E> que se centra en la rapidez de ejecución.

Ver [Set<E>](#) (interface) *java.util.Set<E>*

## 18. *InputStream* (clase abstracta) *java.io.InputStream*

Madre de un conjunto de clases para leer ficheros byte a byte.

La clase derivada más habitual es

<b>java.io.FileInputStream</b>	
<code>FileInputStream(File file)</code>	constructor
<code>FileInputStream(String nombre)</code>	constructor

Todas las clases derivadas de InputStream proporcionan estos métodos:

<b>java.io.InputStream</b>	
<code>int available()</code>	una estimación del número de bytes que quedan por leer
<code>void close()</code>	cierra el fichero
<code>int read()</code>	lee un byte devuelve el byte leído pasado a entero devuelve -1 si el fichero se ha acabado

<code>int read(byte[] bytes)</code>	lee un número 'n' de bytes igual o menor que la longitud del array 'bytes' devuelve el número de bytes leídos; los bytes leídos están en las posiciones [0 .. n-1] del array 'bytes' devuelve -1 si el fichero se ha acabado
<code>int read(byte[] bytes, int start, int n)</code>	lee un número 'n' de bytes devuelve el número de bytes leídos; los bytes leídos están en las posiciones [start .. start+n-1] del array 'bytes' devuelve -1 si el fichero se ha acabado

Es habitual preparar un array de cierto tamaño, pecando por exceso, para ir leyendo montones de bytes. En cada caso particular habrá que decidir cómo se van leyendo bytes para hacer lo que haya que hacerles. El ejemplo siguiente lee el fichero entero, devolviendo todos los bytes en un array:

```
/**
 * Lectura de todos los bytes de un fichero.
 *
 * @param fichero nombre del fichero del que queremos leer.
 * @return todos los bytes en el fichero.
 * @throws IOException si hay problemas al abrir o al escribir.
 */
public byte[] leeTodosLosBytes(String fichero)
    throws IOException {
    File file = new File(fichero);
    int longitud = (int) file.length();
    InputStream is = new FileInputStream(fichero);
    byte[] total = new byte[longitud];
    int leidos = 0;
    while (leidos < longitud) {
        int n = is.read(total, leidos, 1000);
        if (n < 0)
            break;
        leidos += n;
    }
    is.close();
    return total;
}
```

Java proporciona una clase derivada, [BufferedInputStream](#), que proporciona la misma funcionalidad, pero realiza una gestión más eficiente del fichero, leyendo muchos bytes de golpe para que las llamadas a los métodos 'read()' sean más rápidas.

El ejemplo anterior probablemente funcione más deprisa escrito de la siguiente forma:

```

public byte[] leeTodosLosBytes(String fichero)
    throws IOException {
    File file = new File(fichero);
    int longitud = (int) file.length();
    InputStream is =
        new BufferedInputStream(
            new FileInputStream(fichero));
    byte[] total = new byte[longitud];
    int leidos = 0;
    while (leidos < longitud) {
        int n = is.read(total, leidos, 1000);
        if (n < 0)
            break;
        leidos += n;
    }
    is.close();
    return total;
}

```

InputStream lee bytes (8 bits). Si necesita leer caracteres (16 bits), debe utilizar la clase [Reader](#). Ver [File](#), [Reader](#), [OutputStream](#).

## 19. *Integer (clase) java.lang.Integer*

Además de proporcionar un objeto para contener [números](#) enteros, aporta algunos métodos interesantes

```

Integer.MIN_VALUE
Integer.MAX_VALUE
Integer(int v)
Integer(String s)
static int parseInt(String s)
static int parseInt(String s, int base)
static String toString(int v)
static String toBinaryString(int v)
static String toOctalString(int v)
static String toHexString(int v)

```

## 20. *Iterable<T> (interface) java.lang.Iterable<T>*

Las clases que implementan esta interfaz deben proporcionar un método “iterator()” que devuelva un [iterador](#) sobre la objetos de clase T:

```

public interface Iterable<T> {
    public java.util.Iterator<T> iterator();
}

```

Los objetos de clases que implementan esta interface son susceptibles de usarse directamente en sentencias “[for each](#)” para ir recorriendo los objetos que va devolviendo el iterador. Sea la clase:

```

class X implements Iterable<T> {
    Iterator<T> iterator() { ... }
    ...
}

```

Si X es un objeto de esa clase, se pueden recorrer sus elementos fácilmente:

```
X objeto = ...;
for (T elemento: objeto) { ... }
```

Ver “[recorridos con iterador \(for each\)](#)”.

## 21. *Iterator<E> (interface) java.util.Iterator<E>*

Interfaz normalizada para recorrer ordenadamente los elementos de una colección.

Define 3 métodos a implementar:

<i>java.util.Iterator&lt;E&gt;</i>
<pre>public interface Iterator&lt;E&gt; {     /**      * @return TRUE si la siguiente llamada a next()      *         devolverá un elemento      */     public boolean hasNext();      /**      * @return el siguiente elemento de la colección      * @throws <b>NoSuchElementException</b>      *         si no hay ningún elemento que devolver      */     public E next();      /**      * Elimina de la colección el último elemento devuelto por next()      * @throws <b>UnsupportedOperationException</b>      *         si la operación no es posible      * @throws <b>IllegalStateException</b>      *         si no se acaba de llamar a next()      */     public void remove(); }</pre>

Con los iteradores se suelen construir los siguientes tipos de bucles

<i>bucles con iteradores</i>
<pre>for (Iterator&lt;E&gt; ite = ...; ite.hasNext(); ) {     E elemento = ite.next();     ... }</pre>
<pre>Iterator&lt;E&gt; ite = ...; while (ite.hasNext()) {     E elemento = ite.next();     ... }</pre>

Ver también “[java.util.Enumeration](#)” que era la interface que se usaba antes de existir “Iterator”.

### ¿De dónde saco un iterador?

Lo más habitual es que las clases de la librería de java lo proporcionen. Por ejemplo:

### [java.util.List<T>](#)

```
List<T> lista = new ArrayList<T>();  
.  
.  
.  
Iterator<T> ite = lista.iterator();  
while (ite.hasNext()) {  
    T elemento = itr.next();  
    .  
    .  
    .  
}
```

### [java.util.Set<T>](#)

```
Set<T> conjunto = new HashSet<T>();  
.  
.  
.  
Iterator<T> ite = conjunto.iterator();  
while (ite.hasNext()) {  
    T elemento = itr.next();  
    .  
    .  
    .  
}
```

Si tenemos un array, podemos pasarlo a lista e iterar sobre la lista:

```
T[] array  
List<T> lista = Arrays.asList(array);  
Iterator<T> ite = lista.iterator();  
...
```

### **¿Qué relación hay entre `Iterator<T>` e `Iterable<T>`?**

---

Cuando una clase implementa la interface [Iterable<T>](#) tiene que proporcionar un método que devuelva un iterador

```
class X implements Iterable<T> {  
    public Iterator<T> iterator();  
    ...  
}
```

Java, visto que la clase tiene ese método, permite hacer bucles de forma aún más compacta

<b><i>forma compacta</i></b>
<pre>X x = new X(...);  for (T elemento: x) {     . . . se hace lo que se tenga que hacer con cada elemento }</pre>
<b><i>forma equivalente desarrollada</i></b>
<pre>X x = new X(...);  Iterator&lt;T&gt; ite = x.iterator(); while (ite.hasNext()) {     T elemento = ite.next();     . . . se hace lo que se tenga que hacer con cada elemento }</pre>

### **ejemplo: iterador sobre un array**

---

Es fácil construir clases que se puedan usar como iteradores, bastando llevar cuenta interna de dónde se está en cada momento.

El ejemplo siguiente proporciona iteradores sobre [arrays](#).

<i><b>class IteraArray&lt;T&gt; implements Iterator&lt;T&gt;</b></i>
<pre>import java.util.*;  public class IteraArray&lt;T&gt;     implements Iterator&lt;T&gt; {     private T[] objetos;     private int posicion = 0;      public IteraArray (T[] objetos) {         this.objetos = (T[]) new Object[objetos.length];         System.arraycopy(objetos, 0, this.objetos, 0, objetos.length);         posicion = 0;     }      public boolean hasNext() {         return posicion &lt; objetos.length;     }      public T next() {         if (posicion &lt; objetos.length)             return objetos[posicion++];         throw new NoSuchElementException();     }      public void remove() {         throw new UnsupportedOperationException();     } }</pre>

Una vez tenemos el iterador se puede usar de forma compacta

```
public static void main(String[] args) {
    IteraArray<String> it = new IteraArray<String>(args);
    for (String s: it)
        System.out.println(s);
}
```

O de forma más extendida:

```
public static void main(String[] args) {
    IteraArray<String> it = new IteraArray<String>(args);
    while (it.hasNext())
        System.out.println(it.next());
}
```

### **ejemplo: iterador que genera una serie aleatoria**

---

El siguiente ejemplo envuelve la generación de números aleatorios en un iterador que devuelve una serie interminable de números aleatorios.



***class SerieAleatoria implements Iterator<Integer>***

```
public class SerieAleatoria
    implements Iterator<Integer> {
    private Random random;

    public SerieAleatoria() {
        random = new Random();
    }

    public boolean hasNext() {
        return true;
    }

    public Integer next() {
        return random.nextInt();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

**ejemplo: iterador sobre otro iterador**

El siguiente ejemplo es una clase que trabaja sobre una serie de números, dejando pasar sólo aquellos que son múltiplos de N. Para ello necesita consumir varios datos de entrada en cada paso, hasta que encuentra un número que valga o se acabe la lista de entrada.

***class FiltraMultiplos implements Iterator<Integer>***

```
public class FiltraMultiplos
    implements Iterator<Integer> {
    private final Iterator<Integer> entrada;
    private final int n;
    private boolean hayMas;
    private Integer siguiente;

    public FiltraMultiplos(Iterator<Integer> entrada, int n) {
        this.entrada = entrada;
        this.n = n;
        calculaSiguientePaso();
    }

    private void calculaSiguientePaso() {
        while (true) {
            hayMas = entrada.hasNext();
            if (!hayMas)
                break;
            siguiente = entrada.next();
            if (siguiente % n == 0)
                break;
        }
    }
}
```

```

    public boolean hasNext() {
        return hayMas;
    }

    public Integer next() {
        if (!hayMas)
            throw new NoSuchElementException();
        Integer devolver = siguiente;
        calculaSiguientePaso();
        return devolver;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

```

## 22. *LinkedList<E>* (clase) *java.util.LinkedList<E>*

Listas optimizadas para acelerar la inserción y extracción de elementos cualesquiera de la lista.

Ver [List<E> \(interface\) java.util.List<E>](#)

## 23. *List<E>* (interface) *java.util.List<E>*

Dentro de las [colecciones](#) de datos que java proporciona en su biblioteca básica, esta interfaz recoge las secuencias de datos en las que

- se respeta el orden en el que se insertan elementos
- pueden haber elementos duplicados
- el tamaño de la lista se adapta dinámicamente a lo que haga falta

El resultado es una especie de [array](#) de tamaño adaptable.

### métodos de la interface

---

En lo que sigue, un elemento “x” es “[equals](#)” a otro elemento “y” sí y sólo si  
“x.equals(y) == true”.

<i>interface List&lt;E&gt; extends Collection&lt;E&gt;</i>
<b>boolean add(E elemento)</b> añade un elemento al final de la lista <b>void add(int posicion, E elemento)</b> inserta un elemento en la posición indicada <b>void clear()</b> vacía la lista <b>boolean contains(E elemento)</b> true si hay en la lista un elemento “equals” al indicado <b>boolean equals(Object x)</b>

<b><i>interface List&lt;E&gt; extends Collection&lt;E&gt;</i></b>	
	una lista es igual a otra si contienen en las mismas posiciones elementos que son respectivamente "equals"
<b>E get(int posicion)</b>	devuelve el elemento en la posición indicada
<b>int indexOf(E elemento)</b>	devuelve la posición en la que se haya el primer elemento "equals" al indicado; o -1 si no hay ningún elemento "equals"
<b>boolean isEmpty()</b>	true si no hay elementos
<b>Iterator&lt;E&gt; iterator()</b>	devuelve un iterador sobre los elementos de la lista
<b>E remove(int posicion)</b>	elimina el elemento en la posición indicada, devolviendo lo que elimina
<b>boolean remove(E elemento)</b>	elimina el primer elemento de la lista que es "equals" el indicado; devuelve true si se elimina algún elemento
<b>E set(int posicion, E elemento)</b>	reemplaza el elemento X que hubiera en la posición indicada; devuelve lo que había antes, es decir X
<b>int size()</b>	devuelve el número de elementos en la lista
<b>Object[] toArray()</b>	devuelve un array cargado con los elementos de la lista

## **implementaciones estándar**

---

El propio paquete java.util proporciona algunas implementaciones de la interface List, sin perjuicio de que se puedan programar otras.

**class ArrayList<E> implements List<E>**

Es una implementación muy eficiente en cuanto a uso de memoria. Es rápida en todas las operaciones, excepto en las que afectan a elementos intermedios: inserción y borrado. Puede decirse que es un "[array](#)" de tamaño dinámico.

**class LinkedList<E> implements List<E>**

Es una implementación basada en listas encadenadas. Esto ofrece una buena velocidad en operaciones sobre términos intermedios (inserción y borrado) a cambio de ralentizar las demás operaciones.

**class Vector<E> implements List<E>**

Similar a "ArrayList" pero con métodos sincronizados, lo que permite ser usada en programas concurrentes. Todo es más lento que con una ArrayList.

## ejemplo de uso

```
List<Integer> lista = new ArrayList<Integer>();

lista.add(1);
lista.add(9);
lista.add(1, 5);
System.out.println(lista.size());
System.out.println(lista.get(0));
System.out.println(lista.get(1));
System.out.println(lista.get(2));
for (int n: lista) {
    System.out.print(n + " ");
}
System.out.println();
```

## list frente a arrays

[List](#) y [arrays](#) son intercambiables, con la principal diferencia de que en un array hay que indicar el tamaño al construirlo, mientras que las List se adaptan automáticamente al número de datos que contienen.

<i>arrays</i>	<i>List</i>
<code>String[] x;</code>	<code>List&lt;String&gt; x;</code>
<code>x = new String[1000];</code>	<code>x = new ArrayList&lt;String&gt;();</code>
<code>... = x[20];</code>	<code>... = x.get(20);</code>
<code>x[20] = "1492";</code>	<code>x.set(20, "1492");</code>
	<code>x.add("2001");</code>

Si necesita un array y tiene una lista:

```
java.util.List list = Arrays.asList(array);
```

Trabaja sobre Object:

```
public static List asList(Object[] a)
```

Si tiene una lista y necesita un array:

```
Object array[] = list.toArray();
```

Trabaja sobre Object:

```
public Object[] toArray()
```

Si ya tiene el array creado y desea llenarlo con los elementos de la lista:

```
public Object[] toArray(Object[] a)
```

Una ventaja de este último método es que respeta el tipo de los elementos, lo que combinado con el hecho de que crea el array si no existe o si no tiene espacio para todos los elementos de la lista, permite escribir cosas de este tipo:

```
String[] x = (String[]) v.toArray(new String[0]);
```

Ver [arrays o listas?](#)

## listas de Object

---

Normalmente las listas se utilizan indicando el tipo de objetos que pueden contener. Pero también se pueden utilizar listas sobre Objetos en general, lo que permite listas heterogéneas a cambio de ser (típicamente) necesario el uso de [downcasting](#) en la recuperación de los elementos.

Ejemplo:

```
List lista = new ArrayList();

lista.add(1);
lista.add(9);
lista.add(1, 5);
System.out.println(lista.size());
System.out.println(lista.get(0));
System.out.println(lista.get(1));
System.out.println(lista.get(2));
for (Iterator it = lista.iterator(); it.hasNext();) {
    int n = (Integer) it.next();
    System.out.print(n + " ");
}
System.out.println();
```

## 24. Map<K, V> (interface) java.util.Map<K, V>

Dentro de las colecciones de datos que java proporciona en su biblioteca básica, esta interfaz recoge una estructura de datos que permite almacenar asociaciones { clave, valor } de forma que dada una clave podemos localizar inmediatamente el valor asociado. A veces se denominan diccionarios o incluso "[arrays](#) asociativos". Se satisfacen las siguientes propiedades

- las claves, de tipo K, no pueden estar duplicadas
- los valores, de tipo V, pueden ser cualesquiera
- el tamaño del mapa se adapta dinámicamente a lo que haga falta

El resultado es una especie de *array* de tamaño adaptable que, en vez de indexarse por posición, se indexa por medio de una clave.

### métodos de la interface

---

En lo que sigue, un elemento "x" es "equals" a otro elemento "y" sí y sólo si

"x.equals(y) == true".

<i>interface Map&lt;K, V&gt;</i>
<b>void clear()</b> elimina todas las claves y valores
<b>boolean containsKey(Object clave)</b> devuelve true si alguna clave es "equals" a la indicada
<b>boolean containsValue(Object valor)</b> devuelve true si algún valor es "equals" al indicado

<i><b>interface Map&lt;K, V&gt;</b></i>
<b>boolean equals(Object x)</b> devuelve true si contiene las mismas claves y valores asociados
<b>V get(Object clave)</b> devuelve el valor asociado a la clave indicada
<b>boolean isEmpty()</b> devuelve true si no hay claves ni valores
<b>Set&lt;K&gt; keySet()</b> devuelve el conjunto de claves
<b>V put(K clave, V valor)</b> asocia el valor a la clave; devuelve el valor que estaba asociado anteriormente, o NULL si no había nada para esa clave
<b>V remove(Object clave)</b> elimina la clave y el valor asociado; devuelve el valor que estaba asociado anteriormente, o NULL si no había nada para esa clave
<b>int size()</b> número de asociaciones { clave, valor }
<b>Collection&lt;V&gt; values()</b> devuelve una estructura de datos iterable sobre los valores asociados

### implementaciones estándar

---

El propio paquete java.util proporciona algunas implementaciones de la interface Map, sin perjuicio de que se puedan programar otras.

**class HashMap<K, V> implements Map<K, V>**

Es una implementación muy eficiente en cuanto a uso de memoria. Es rápida en todas las operaciones. Puede decirse que es un “array asociativo” de tamaño dinámico.

**class LinkedHashMap<K, V> implements Map<K, V>**

Es una implementación basada en listas encadenadas. Respeta el orden de inserción, a cambio de ser más lenta.

**class Hashtable<K, V> implements Map<K, V>**

Similar a “HashMap” pero con métodos sincronizados, lo que permite ser usada en programas concurrentes. Todo es más lento que con una HashMap.

### variantes ordenadas

---

Son aquellas implementaciones que Map<K, V> que mantienen las claves ordenadas; es decir, que cuando se itera sobre el Map, las claves van saliendo en orden. Para poder ordenar las claves, es necesario que sean de objetos comparables.

**interface SortedMap<K, V> extends Map<K, V>**

Variante de Map<K, V> con la propiedad de que mantiene las claves ordenadas; es decir, cuando se itera sobre el Map, las claves están en orden. Las claves deben ser Comparable.

**class TreeMap<K, V> implements SortedMap<K, V>**

Es una implementación que garantiza el orden de las claves cuando se itera sobre ellas. Es más voluminosa y lenta.

### ejemplos de uso

<i><b>class HashMap</b></i>	
<pre>Map&lt;String, String&gt; mapa =     new <b>HashMap</b>&lt;String, String&gt;();  mapa.put("uno", "one"); mapa.put("dos", "two"); mapa.put("tres", "three"); mapa.put("cuatro", "four"); mapa.put("tres", "33"); System.out.println(mapa.size()); for (String clave: mapa.keySet()) {     String valor = mapa.get(clave);     System.out.println(clave + " -&gt; " + valor); }</pre>	<pre>4 cuatro -&gt; four tres -&gt; 33 uno -&gt; one dos -&gt; two</pre>
<i><b>class TreeMap</b></i>	
<pre>Map&lt;String, String&gt; mapa =     new <b>TreeMap</b>&lt;String, String&gt;();  mapa.put("uno", "one"); mapa.put("dos", "two"); mapa.put("tres", "three"); mapa.put("cuatro", "four"); mapa.put("tres", "33"); System.out.println(mapa.size()); for (String clave: mapa.keySet()) {     String valor = mapa.get(clave);     System.out.println(clave + " -&gt; " + valor); }</pre>	<pre>4 cuatro -&gt; four dos -&gt; two tres -&gt; 33 uno -&gt; one</pre>

<b>class LinkedHashMap</b>	
<pre>Map&lt;String, String&gt; mapa =     new <b>LinkedHashMap</b>&lt;String, String&gt;();  mapa.put("uno", "one"); mapa.put("dos", "two"); mapa.put("tres", "three"); mapa.put("cuatro", "four"); mapa.put("tres", "33"); System.out.println(mapa.size()); for (String clave: mapa.keySet()) {     String valor = mapa.get(clave);     System.out.println(clave + " -&gt; " + valor); }</pre>	<pre>4 uno -&gt; one dos -&gt; two tres -&gt; 33 cuatro -&gt; four</pre>

### mapas multi-clave

Los mapas utilizan un sólo objeto como clave. Cuando se quieren usar varios objetos como clave, es necesario crear un objeto que contenga los diferentes elementos de la clave, cuidando de redefinir los métodos equals() y hashCode() para obtener los resultados deseados.

Ejemplo.

Queremos guardar objetos asociados a 2 claves: las coordenadas X-Y en un plano. No podemos usar un array bidimensional bien porque las coordenadas no están acotadas (y el array sería demasiado grande), bien porque la tabla es poco densa (la mayor parte de las posiciones están vacías y el array estaría mayormente desperdiciado con celdas null).

En estos casos recurrimos a crear objetos que engloben la X y la Y:

<b>clave combinada: class Punto</b>
<pre>public class Punto {     private final int x, y;      public Punto(int x, int y) {         this.x = x;         this.y = y;     }      @Override     public boolean equals(Object o) {         if (this == o) return true;         if (o == null) return false;         if (!(o instanceof Punto)) return false;         Punto punto = (Punto) o;         return x == punto.x &amp;&amp; y == punto.y;     }      @Override</pre>



```

    public int hashCode() {
        return 31 * x + y;
    }
}

```

Y ahora podemos usar estas claves para acceder al mapa:

<i>tabla multiclave</i>
<pre> public class SparseTable {     private final Map&lt;Punto, Object&gt; map =         new HashMap&lt;Punto, Object&gt;();      public void put(int x, int y, Object o) {         map.put(new Punto(x, y), o);     }      public Object get(int x, int y) {         return map.get(new Punto(x, y));     } } </pre>

El método es fácilmente generalizable a cualquier conjunto de datos que queramos usar como clave de acceso, sin más que envolverlo en la clave combinada correspondiente.

## mapas de Object

Normalmente los mapas se utilizan indicando el tipo de las claves y valores que pueden contener. Pero también se pueden utilizar mapas sobre Objetos en general, lo que permite mapas heterogéneos a cambio de ser (típicamente) necesario el uso de [downcasting](#) en la recuperación de los elementos.

El mismo ejemplo anterior quedaría así:

```

Map mapa = new HashMap();

mapa.put("uno", "one");
mapa.put("dos", "two");
mapa.put("tres", "three");
mapa.put("cuatro", "four");
mapa.put("tres", "33");
System.out.println(mapa.size());
for (Iterator it = mapa.keySet().iterator(); it.hasNext(); ) {
    String clave = (String) it.next();
    String valor = (String) mapa.get(clave);
    System.out.println(clave + " -> " + valor);
}

```

## 25. Math (clase) java.lang.Math

Java proporciona en la clase Math un amplio surtido de funciones matemáticas. A continuación de citan algunas de uso corriente, sin ánimo de ser exhaustivos:

método	lo que hace
--------	-------------

<b>método</b>	<b>lo que hace</b>
<code>T abs (T x)</code>	valor absoluto del dato; T puede ser entero o real
<code>T max (T x, T y)</code>	máximo de dos valores; T puede ser entero o real
<code>T min (T x, T y)</code>	mínimo de dos valores; T puede ser entero o real
<code>double sqrt (double x)</code>	raíz cuadrada
<code>double exp (double x)</code>	$e^x$
<code>double log (double x)</code>	$\ln(x)$ – logaritmo neperiano
<code>double log10 (double x)</code>	$\log_{10}(x)$ - logaritmo en base 10
<code>double pow (double a, double b)</code>	$a^b$
<code>long round (double x)</code> <code>int round (float x)</code> <code>double rint (double x)</code>	redondeo al entero más cercano
<code>double sin (double angulo)</code> <code>double cos (double angulo)</code> <code>double tan (double angulo)</code> <code>double asin (double seno)</code> <code>double acos (double coseno)</code> <code>double atan (double tangente)</code> <code>double atan2 (double dy, double dx)</code>	funciones trigonométricas, donde los ángulos se expresan en radianes
<code>double toDegrees (double rads)</code> <code>double toRadians (double grados)</code>	conversiones entre ángulos expresados en radianes ( $0 \dots 2\pi$ ) y en grados sexagesimales ( $0^\circ \dots 360^\circ$ ).

## 26. Object (clase) *java.lang.Object*

Es una clase de java con la particularidad de ser la “madre de todas las clases”. Eso quiere decir que todas las clases que escriban los programadores heredan de Object, bien directamente (si no se dice nada) o indirectamente (si la clase extiende a otra).

Así

```
class Ejemplo { ... }
```

es a todos los efectos equivalente a:

```
class Ejemplo extends Object { ... }
```

La clase Object define métodos que, salvo reescritura en las [subclases](#), son compartidos por absolutamente todos los objetos que se creen. Los más relevantes de estos métodos son

<i><b>java.lang.Object</b></i>
<pre> package java.lang;  public class Object {      /**      * @return la clase del objeto en ejecución.      */     public final native Class getClass();      /**      * @return un hashCode para el objeto en ejecución.      */     public native int hashCode();      /**      * @return true si este objeto es igual al pasado como argumento.      */     public boolean equals(Object obj) {         return this == obj;     }      /**      * @return una representación textual del objeto.      */     public String toString() {         return getClass().getName() +             "@ " + Integer.toHexString(hashCode());     } } </pre>

Los métodos reseñados no son los únicos; simplemente son los que el autor ha considerado más relevantes el propósito de este documento.

## **27. *OutputStream (clase abstracta) java.io.OutputStream***

Madre de un conjunto de clases para escribir ficheros byte a byte.

La clase derivada más habitual es

<b>java.io.FileOutputStream</b>	
<code>FileOutputStream(File file)</code>	constructor
<code>FileOutputStream(File file, boolean append)</code>	constructor: añade al final
<code>FileOutputStream(String nombre)</code>	constructor
<code>FileOutputStream(String nombre, boolean append)</code>	constructor: añade al final

Todas las clases derivadas de OutputStream proporcionan estos métodos:

java.io.OutputStream	
void close()	cierra el fichero, asegurando que todo queda bien escrito en el fichero en disco
void write(byte[] b)	escribe en el fichero el array de bytes
void write(byte[] b, int start, int n)	escribe 'n' bytes en el fichero, concretamente, los del array 'b', empezando en la posición 'start'.

Ejemplo de uso:

```
/**
 * Escritura de bytes en ficheros.
 *
 * @param fichero nombre del fichero al que queremos escribir.
 * @param bytes bytes que queremos mandar al fichero.
 * @throws IOException si hay problemas al abrir o al escribir.
 */
public void escribeBytes(String fichero, byte[] bytes)
    throws IOException {
    OutputStream os = new FileOutputStream(fichero);
    os.write(bytes);
    os.close();
}
```

OutputStream escribe bytes (8 bits). Si necesita escribir caracteres (16 bits), debe utilizar la clase [Writer](#).

Ver [File](#), [Writer](#), [InputStream](#).

### **PrintStream (clase) java.io.PrintStream**

Clase derivada de OutputStream, para escribir bytes en un fichero, proporcionando una serie de métodos que facilitan tareas habituales:

java.io.PrintStream	
PrintStream(File file)	constructor
PrintStream(String fichero)	constructor
PrintStream(OutputStream out)	constructor
PrintStream(File file, String <a href="#">encoding</a> )	constructor
PrintStream(String fichero, String <a href="#">encoding</a> )	constructor
PrintStream format(String format, Object ... args)	imprime los objetos en el fichero ver <a href="#">Formatter</a>
PrintStream printf(String format, Object ... args)	imprime los objetos en el fichero ver <a href="#">Formatter</a>
void print(boolean b)	imprime el argumento
void print(char c)	

<pre>void print(char[] s) void print(double d) void print(float f) void print(int i) void print(long l) void print(Object object) void print(String s)</pre>	
<pre>void println()</pre>	imprime un cambio de línea
<pre>void println(boolean b) void println(char c) void println(char[] s) void println(double d) void println(float f) void println(int i) void println(long l) void println(Object object) void println(String s)</pre>	imprime el argumento, seguido de un cambio de línea

## 28. *Properties (clase) java.util.Properties*

Es una variante de [Map<K,V>](#) especializada en textos, es decir, un caso particular de Map<String, String>, aunque tiene algunos métodos para facilitar su uso, especialmente escribiendo y leyendo de ficheros de texto.

Métodos más utilizados:

<b>java.util.Properties</b>	
Property()	constructor
Property(Properties defaults)	construye un objeto Properties utilizando otro como defecto; es decir, si el nuevo objeto no tiene la clave que se solicita, se busca en el objeto "defaults".
Object setProperty(String clave, String valor)	asocia el valor a la clave devuelve el antiguo valor asociado a la clave, o NULL si la clave no existía antes
String getProperty(String clave)	devuelve el valor asociado a la clave devuelve NULL si no hay valor asociado
String getProperty(String clave, String X)	devuelve el valor asociado a la clave devuelve X si no hay valor asociado
Set<String> stringPropertyNames()	conjunto de claves que tienen asociado un valor, incluyendo posibles objetos Properties que se usan como respaldo (ver segundo constructor)

## Uso típico

---

```
Properties properties = new Properties();
properties.setProperty("titulo", "Aplicación Gráfica");
properties.setProperty("ancho", "600");
properties.setProperty("alto", "400");
```

```
String titulo = properties.getProperty("titulo", "sin nombre");
int ancho = Integer.parseInt(properties.getProperty("ancho", "400");
```

## Ficheros

---

Es fácil escribir y leer de ficheros, lo que hace que Properties se utilice frecuentemente para tener ficheros externos de configuración.

java.util.Properties	
void load(InputStream input)	lee bytes lo escrito con store(OutputStream)
void load(Reader reader)	lee caracteres lo escrito con store(Reader)
void loadFromXML( InputStream input)	lee un fichero en XML lo escrito con storeToXML(OutputStream, ...)
void store(OutputStream out, String comentario)	escribe en un fichero de bytes
void store(Writer out, String comentario)	escribe en un fichero de caracteres
void storeToXML(OutputStream out, String comentario)	escribe en un fichero XML
void storeToXML(OutputStream out, String comentario, String codificacion)	escribe en un fichero XML utilizando una <a href="#">codificación</a> explícita

Los formatos son bastante evidentes:

```
void guarda(Properties p, File f)
    throws IOException {
    Writer writer = new FileWriter(f);
    p.store(writer, "mis parejas");
}
```

```
#mis parejas
#Wed Dec 02 09:01:09 CET 2009
alto=400
titulo=Aplicación Gráfica
ancho=600
```

```
void carga(Properties p, File f)
    throws IOException {
    Reader reader= new FileReader(f);
    p.load(reader);
}
```

```
void guardaXML(Properties p, File f)
    throws IOException {
    OutputStream out = new FileOutputStream(f);
    p.storeToXML(out, "mis parejas", "iso-8859-1");
}
```

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>mis parejas</comment>
<entry key="alto">400</entry>
<entry key="titulo">Aplicación Gráfica</entry>
<entry key="ancho">600</entry>
</properties>
```

```
void cargaXML(Properties p, File f)
    throws IOException {
    InputStream stream= new FileInputStream(f);
    p.loadFromXML(stream);
}
```

## 29. *Random (clase) java.util.Random*

Los objetos de esta clase sirven para generar números aleatorios.

La clase disfruta de 2 constructores:

```
public Random()
```

Crea un objeto que genera números aleatorios.

```
public Random(long semilla)
```

Crea objetos que generan números aleatorios con la particularidad de que si se crean dos objetos con la misma semilla, ambos objetos generarán la misma serie de números aleatorios.

Una vez generado un objeto Random, se dispone de varios métodos. Quizás los más usados sean los siguientes:

<b>class java.util.Random</b>	
<code>boolean nextBoolean()</code>	true o false, aleatoriamente
<code>int nextInt()</code>	un entero cualquiera
<code>int nextInt(int n)</code>	un entero aleatorio entre 0 y n: $0 \leq \text{aleatorio} < n$
<code>long nextLong()</code>	un entero long cualquiera
<code>void nextBytes(byte[] bytes)</code>	rellena el array con bytes aleatorios
<code>double nextDouble()</code>	un real (double) aleatorio
<code>float nextFloat()</code>	un real (float) aleatorio
<code>double nextGaussian()</code>	un real aleatorio con una distribución estándar de media 0.0 y desviación típica 1.0

## 30. *Reader (clase abstracta) java.io.Reader*

Madre de un conjunto de clases para leer ficheros carácter a carácter.

Java convierte los bytes que contiene el fichero en caracteres [unicode](#). Vea "[codificación de los caracteres](#)".

La clase derivada más habitual es

<b>java.io.FileReader</b>	
<code>FileReader(File file)</code>	constructor
<code>FileReader(String nombre)</code>	constructor

Todas las clases derivadas de Reader proporcionan estos métodos:

<b>java.io.Reader</b>	
<code>void close()</code>	cierra el fichero
<code>int read()</code>	lee un carácter



	devuelve el carácter leído pasado a entero devuelve -1 si el fichero se ha acabado
<code>int read(char[] chars)</code>	lee un número 'n' de caracteres igual o menor que la longitud del array 'chars' devuelve el número de caracteres leídos; los caracteres leídos están en las posiciones [0 .. n-1] del array 'chars' devuelve -1 si el fichero se ha acabado
<code>int read(char[] chars,           int start,           int n)</code>	lee un número 'n' de caracteres devuelve el número de caracteres leídos; los caracteres leídos están en las posiciones [start .. start+n-1] del array 'chars' devuelve -1 si el fichero se ha acabado

Es habitual preparar un array de cierto tamaño, pecando por exceso, para ir leyendo montones de caracteres. En cada caso particular habrá que decidir cómo se van leyendo caracteres para hacer lo que haya que hacerles. El ejemplo siguiente lee el fichero entero, devolviendo todos los caracteres en una cadena:

```
/**
 * Lectura de todos los caracteres de un fichero.
 *
 * @param fichero nombre del fichero del que queremos leer.
 * @return todos los caracteres en el fichero.
 * @throws IOException si hay problemas al abrir o al escribir.
 */
public String leeTodosLosCaracteres(String fichero)
    throws IOException {
    StringBuilder buffer = new StringBuilder();
    Reader reader = new FileReader(fichero);
    char[] chars = new char[1024];
    while (true) {
        int n = reader.read(chars);
        if (n < 0)
            break;
        buffer.append(chars, 0, n);
    }
    reader.close();
    return buffer.toString();
}
```

Vea "[BufferedReader](#)" para leer caracteres más deprisa y para leer línea a línea.

Reader lee caracteres (16 bits). Si necesita leer bytes (8 bits), debe utilizar la clase [InputStream](#).

Ver [File](#), [InputStream](#), [Writer](#).

### **BufferedReader (clase) java.io.BufferedReader**

Es una clase derivada de Reader que proporciona

más rapidez, al leer montones de caracteres antes de que se los pidan

un método "readLine()" que devuelve los caracteres línea a línea

BufferedReader realiza una gestión más eficiente del fichero, leyendo muchos caracteres de golpe para que las llamadas a los métodos 'read()' sean más rápidas.

El ejemplo anterior probablemente funcione más deprisa escrito de la siguiente forma:

```
public String leeTodosLosCaracteres(String fichero)
    throws IOException {
    StringBuilder buffer = new StringBuilder();
    Reader reader =
        new BufferedReader(
            new FileReader(fichero));
    char[] chars = new char[1024];
    while (true) {
        int n = reader.read(chars);
        if (n < 0)
            break;
        buffer.append(chars, 0, n);
    }
    reader.close();
    return buffer.toString();
}
```

El ejemplo siguiente muestra el uso del método "readLine()" para leer línea a línea:

```
public List<String> leeTodasLasLineas(String fichero)
    throws IOException {
    List<String> lineas = new ArrayList<String>();
    BufferedReader reader =
        new BufferedReader(
            new FileReader(fichero));
    while (true) {
        String linea = reader.readLine();
        if (linea == null)
            break;
        lineas.add(linea);
    }
    reader.close();
    return lineas;
}
```

### 31. **Scanner (clase) java.util.Scanner**

Analizador léxico simple. Se construye sobre algún tipo de fuente de caracteres:

```
Scanner (String source)
Scanner (Readable source)
Scanner (Reader source)
Scanner (InputStream source)
Scanner (File source)
```

Sobre la fuente de caracteres, va seleccionando lexemas (*tokens*) separados por espacio en blanco. Proporciona los resultados por medio de una interface Iterator:

```
String s =
    "Martes, 13 de septiembre de 2005, actualizado a las 16:16 h.";
    Scanner scanner = new Scanner(s);
    for (Iterator it = scanner; it.hasNext(); ) {
        String token = (String) it.next();
        System.out.println(token);
    }
```

```
Martes,
13
de
septiembre
de
2005,
actualizado
a
las
16:16
h.
```

Se puede parametrizar Scanner para que utilice cualquier otro tipo de separador de *tokens*.

Además, ofrece una serie de métodos que, habiendo leído un *token*, lo intentan interpretar como algún tipo primitivo de java:

```
int nextInt()           double nextDouble()   boolean nextBoolean()
byte nextByte()         float nextFloat()
short nextShort()
long nextLong()
```

Para analizar valores numéricos se emplean los convenios locales de puntuación.

Dichos métodos intentan interpretar el token que toca leer, lanzando una excepción si no puede:

- InputMismatchException si el token no responde al tipo deseado
- NoSuchElementException si no quedan más tokens

Por último cabe mencionar el método

```
String nextLine()
```

que devuelve lo que queda por leer de la línea actual; es decir, desde donde estemos hasta el primer fin de línea. Llamadas consecutivas a nextLine() van proporcionando líneas sucesivas de texto.

### **Ejemplo: interacción con consola**

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    while (true) {
        System.out.print("¿Sigo? [si/no] ");
        String respuesta = scanner.nextLine();
        if (respuesta.equalsIgnoreCase("si"))
```

```

        continue;
    if (respuesta.equalsIgnoreCase("no"))
        break;
    System.out.println("no entiendo esa respuesta: "
                        + respuesta);
}
System.out.println("Adios.");
}

```

### Ejemplo: entrada de datos por consola

```

Scanner scanner = new Scanner(System.in);
System.out.print("Escriba dos números: ");
double x = scanner.nextDouble();
double y = scanner.nextDouble();
System.out.println("producto: " + (x + y));

```

```

Escriba dos números: 3,1416 2
producto: 5.1416

```

## 32. Set<E> (interface) java.util.Set<E>

Dentro de las [colecciones](#) de datos que java proporciona en su biblioteca básica, esta interfaz recoge los conjuntos de datos que se caracterizan porque:

- no se respeta el orden en el que se insertan elementos
- no pueden haber elementos duplicados
- el tamaño del conjunto se adapta dinámicamente a lo que haga falta

### métodos de la interface

En lo que sigue, un elemento “x” es “equals” a otro elemento “y” sí y sólo si “x.equals(y) == true”.

<i><b>interface Set&lt;E&gt; extends Collection&lt;E&gt;</b></i>
<b>boolean add(E elemento)</b> añade un elemento al conjunto, si no estaba ya; devuelve true si el conjunto crece <b>void clear()</b> vacía el conjunto <b>boolean contains(E elemento)</b> devuelve true si existe en el conjunto algún elemento “equals” al indicado <b>boolean equals(Object x)</b> devuelve true si uno y otro conjunto contienen el mismo número de elementos y los de un conjunto son respectivamente “equals” los del otro <b>boolean isEmpty()</b> devuelve true si el conjunto está vacío

<i><b>interface Set&lt;E&gt; extends Collection&lt;E&gt;</b></i>
<b>Iterator&lt;E&gt; iterator()</b> devuelve un iterador sobre los elementos del conjunto
<b>boolean remove(E elemento)</b> si existe un elemento "equals" al indicado, se elimina; devuelve true si varía el tamaño del conjunto
<b>int size()</b> número de elementos en el conjunto

### **implementaciones estándar**

El propio paquete java.util proporciona algunas implementaciones de la interface Set, sin perjuicio de que se puedan programar otras.

**class HashSet<E> implements Set<E>**

Es una implementación muy eficiente en cuanto a uso de memoria. Es rápida en todas las operaciones.

**class TreeSet<E> implements Set<E>**

Es una implementación más lenta y pesada; pero presenta la ventaja de que el iterador recorre los elementos del conjunto en orden.

### **ejemplo de uso**

```
Set<Integer> conjunto = new HashSet<Integer>();

conjunto.add(1);
conjunto.add(9);
conjunto.add(5);
conjunto.add(9);
System.out.println(conjunto.size());
for (int n: conjunto) {
    System.out.print(n + " ");
}
System.out.println();
```

### **conjuntos de Object**

Normalmente los conjuntos se utilizan indicando el tipo de objetos que pueden contener. Pero también se pueden utilizar conjuntos sobre Objetos en general, lo que permite conjuntos heterogéneos a cambio de ser (típicamente) necesario el uso de [downcasting](#) en la recuperación de los elementos.

El mismo ejemplo anterior quedaría así:

```
Set conjunto = new HashSet();

conjunto.add(1);
conjunto.add(9);
conjunto.add(5);
conjunto.add(9);
```

```
System.out.println(conjunto.size());
for (Iterator it = conjunto.iterator(); it.hasNext();) {
    int n = (Integer) it.next();
    System.out.print(n + " ");
}
System.out.println();
```

### 33. **SortedMap<K, V> (clase) java.util.SortedMap**

Asociaciones tipo Map<K, V> donde los elementos se recorren en orden.

Ver [Map<K, V> \(interface\) java.util.Map<K, V>](#)

### 34. **Stacks (concepto)**

Ver "[pilas](#)".

### 35. **String (clase) java.lang.String**

Los "String" son objetos de java con una sintaxis especialmente cómoda para representar cadenas de [caracteres](#). Los caracteres se codifican usando [Unicode](#).

Java permite escribir directamente cadenas entre comillas.

```
String autor = "Miguel de Cervantes";
String colega = "Doménikos Theotokópoulos, \"El Greco\"";
```

Para incluir el carácter comillas dobles en una cadena, escriba "\"".

Sobre cadenas se define la operación de concatenar:

```
String nombre = "Miguel";
String apellido = "Muñoz";
String colega = nombre + " " + apellido;
```

La clase String define muchos métodos interesantes para hacer programas:

**int length()**

longitud: número de caracteres

**boolean equals(String b)**

determina si la cadena contiene los mismos caracteres que la cadena "b"

**boolean equalsIgnoreCase(String b)**

determina si la cadena contiene los mismo caracteres que la cadena "b", independientemente de que estén en mayúsculas o minúsculas

**int compareTo(String b)**

compara contra la cadena "b", devolviendo

- un valor negativo si la cadena es anterior a "b";
- cero (0) si la cadena es igual a "b";
- un valor positivo si la cadena es posterior a "b".

En las comparaciones se usa el código Unicode, lo que no siempre responde al orden lexicográfico (de diccionario).

**String trim()**

crea un nuevo objeto eliminado el espacio en blanco que pudiera haber al principio o al

final

**char charAt(int posicion)**

extrae en carácter en la posición indicada

**char[] toCharArray()**

convierte la cadena en un [array](#) de [caracteres](#)

**String substring(int a, int z)**

extrae la sub-cadena entre las posiciones a y z

**String substring(int desde)**

extrae la sub-cadena desde la posición indicada

**int indexOf(char carácter)**

**int indexOf(String cadena)**

indica en qué posición se encuentra el carácter (o cadena) indicado por primera vez, buscando desde el principio

**int lastIndexOf(char carácter)**

**int lastIndexOf(String cadena)**

indica en qué posición se encuentra el carácter (o cadena) indicado por primera vez, buscando desde el final

**boolean startsWith(String prefijo)**

dice si la cadena comienza con el prefijo indicado

**boolean endsWith(String sufijo)**

dice si la cadena termina con el sufijo indicado

**String[] split(String patron)**

fragmenta la cadena en varias subcadenas utilizando el patrón indicado como separador

<b>código</b>	<b>resultado</b>
"Miguel".length()	6
"Miguel".equals("Miguel")	true
"Miguel".equals("miguel")	false
"Miguel".equalsIgnoreCase("miguel")	true
"Miguel".compareTo("Saturnino")	-6 (cualquier entero < 0)
"Miguel".compareTo("Miguel")	0
"Miguel".compareTo("Michelin")	4 (cualquier entero > 0)
"Miguel".charAt(1)	'i'
"Miguel".charAt(4)	'e'
"Miguel".toCharArray()	{ 'M', 'i', 'g', 'u', 'e', 'l' }
"Miguel".substring(1, 4)	"igu"

<b>código</b>	<b>resultado</b>
<code>"Miguel.substring(1)</code>	<code>"iguel"</code>
<code>"tragaldabas".indexOf('a')</code>	<code>2</code>
<code>"tragaldabas".lastIndexOf('a')</code>	<code>9</code>
<code>"tragaldabas".startsWith("tragón")</code>	<code>false</code>
<code>"tragaldabas".endsWith("dabas")</code>	<code>true</code>
<code>"tragaldabas".split("a")</code>	<code>{ "tr", "g", "ld", "b", "s" }</code>

Los objetos String son inmutables. No hay forma de modificarlos una vez contruidos. Por ello algunas operaciones sobre Strings son en extremo ineficientes debido a la creación de objetos que se van a destruir a continuación. Por ello es útil la clase [StringBuilder](#) que permite disponer de un solo objeto que se puede modificar.

Las comparaciones entre String se limitan a usar el orden numérico de los códigos [Unicode](#), lo que a veces choca con el orden al que estamos acostumbrados en los diccionarios, el orden lexicográfico, que depende de cada idioma. Para realizar comparaciones usando el orden lexicográfico propio de cada idioma, véase [Collator](#).

## **Saltos de línea**

---

En sistemas tipo UNIX, Linux, ...

```
"línea 1" + "\n" + "línea 2"
```

En sistemas tipo Windows

```
"línea 1" + "\r\n" + "línea 2"
```

Si queremos que el código se adapte al sistema en el que ejecuta

```
String LS = System.getProperty("line.separator");
"línea 1" LS + "línea 2"
```

o

```
String.format("%s%n%s", "línea 1", "línea 2");
```

Si estamos escribiendo en un fichero, también podemos hacerlo directamente:

```
writer.print("línea 1");
writer.println();
writer.print("línea 2");
```

o

```
writer.println("línea 1");
writer.println("línea 2");
```

## **36. StringBuffer (clase) java.lang.StringBuffer**

A diferencia de la clase estándar [String](#), [StringBuffer](#) permite trabajar con cadenas de [caracteres](#) modificables. Ver también [StringBuilder](#), que es más rápida.



El siguiente ejemplo muestra dos métodos para construir un objeto que un listado de N números separados por comas.

```
public class UsandoStringBuffer {

    private static String metodo1(int n) {
        String resultado = "";
        for (int i = 0; i < n; i++)
            resultado+= i + ", ";
        return resultado;
    }

    private static String metodo2(int n) {
        StringBuffer buffer = new StringBuffer();
        for (int i = 0; i < n; i++)
            buffer.append(i).append(", ");
        return buffer.toString();
    }

    public static void main(String[] argumentos) {
        int n = Integer.parseInt(argumentos[0]);
        long t1, t2;

        t1 = System.currentTimeMillis();
        metodo1(n);
        t2 = System.currentTimeMillis();
        System.out.println("método 1: " + (t2 - t1) + "ms");

        t1 = System.currentTimeMillis();
        metodo2(n);
        t2 = System.currentTimeMillis();
        System.out.println("método 2: " + (t2 - t1) + "ms");
    }
}
```

La diferencia en tiempos de ejecución es notoria:

```
$ java UsandoStringBuffer 100
método 1: 0ms
método 2: 0ms
$ java UsandoStringBuffer 1000
método 1: 47ms
método 2: 0ms
$ java UsandoStringBuffer 10000
método 1: 17218ms
método 2: 16ms
```

El tiempo exacto que tarda en ejecutarse el programa depende de cada ordenador y de qué más programas hay en ejecución en un momento dado. Lo importante es la proporción de tiempos, que es claramente favorable al uso de **StringBuffer**. La diferencia, inapreciable cuando hay pocos objetos que manejar, se torna apabullante cuando el número de objetos crece.

Ver "[StringBuilder](#)": similar; pero más rápida.

### 37. *StringBuilder* (clase) *java.lang.StringBuilder*

A diferencia de la clase estándar [\*String\*](#), *StringBuilder* permite trabajar con cadenas de [caracteres](#) modificables.

El siguiente ejemplo muestra dos métodos para construir un objeto que un listado de N números separados por comas.

```
public class UsandoStringBuilder {

    private static String metodo1(int n) {
        String resultado = "";
        for (int i = 0; i < n; i++)
            resultado+= i + ", ";
        return resultado;
    }

    private static String metodo2(int n) {
        StringBuilder buffer = new StringBuilder();
        for (int i = 0; i < n; i++)
            buffer.append(i).append(", ");
        return buffer.toString();
    }

    public static void main(String[] argumentos) {
        int n = Integer.parseInt(argumentos[0]);
        long t1, t2;

        t1 = System.currentTimeMillis();
        metodo1(n);
        t2 = System.currentTimeMillis();
        System.out.println("método 1: " + (t2 - t1) + "ms");

        t1 = System.currentTimeMillis();
        metodo2(n);
        t2 = System.currentTimeMillis();
        System.out.println("método 2: " + (t2 - t1) + "ms");
    }
}
```

La diferencia en tiempos de ejecución es notoria:

```
$ java UsandoStringBuilder 100
método 1: 16ms
método 2: 0ms
$ java UsandoStringBuilder 1000
método 1: 47ms
método 2: 0ms
$ java UsandoStringBuilder 10000
método 1: 13922ms
método 2: 0ms
```

El tiempo exacto que tarda en ejecutarse el programa depende de cada ordenador y de qué más programas hay en ejecución en un momento dado. Lo importante es la proporción de tiempos, que es claramente favorable al uso de *StringBuilder*. La diferencia, inapreciable cuando hay pocos objetos que manejar, se torna apabullante cuando el número de objetos crece.

### 38. *System.err*

Conecta con una salida alternativa a la estándar, permitiendo escribir en ella. Similar a [System.out](#) y, además, pueden convivir:

<i>class SystemErr</i>
<pre>public class SystemErr {     public static void main(String[] argumentos) {         System.out.println("System.out: salida estándar");         System.err.println("System.err: salida alternativa");     } }</pre>

Normalmente lo que se escribe aparece en la pantalla

```
$ java SystemErr
System.out: salida estándar
System.err: salida alternativa
```

pero es fácil que el usuario envíe la salida estándar a un fichero, en cuyo caso sólo la salida alternativa aparece en pantalla

```
$ java SystemErr > salida.txt
System.err: salida alternativa
```

Desde el sistema operativo, es fácil para el usuario redirigir la salida alternativa a un fichero

```
$ java SystemErr 2> errores.txt
System.out: salida estándar
```

e incluso dirigir cada salida a un fichero específico

```
$ java SystemErr > salida.txt 2> errores.txt
```

o ambas salidas al mismo fichero

```
java SystemErr > todo.txt 2>&1
```

y también es posible cambiar la salida desde dentro del programa

```
PrintStream salida = new PrintStream("errores.txt");
System.setErr(salida);
```

La definición en la clase System es así:

```
public final static PrintStream err;
```

Ver [System.out](#).

### 39. *System.in*

Conecta con la entrada estándar de la aplicación, permitiendo leer de ella.

Lo más típico es leer líneas para analizarlas

<i>class SystemIn</i>
<pre>import java.io.*;  public class SystemIn {     public static void main(String[] argumentos)         throws Exception {         BufferedReader reader =             new BufferedReader(                 new InputStreamReader(System.in));         String linea = reader.readLine();         int nLineas = 1;         while (linea != null) {             System.out.println("\t" + nLineas + ": " + linea);             linea = reader.readLine();             nLineas++;         }     } }</pre>
<pre>\$ java SystemIn una     1: una dos     2: dos y tres     3: y tres ^Z</pre>

También se puede usar un analizador lexicográfico. Ver [“Scanner”](#).

Desde el sistema operativo, es fácil para el usuario redirigir la entrada desde un fichero

```
$ java SystemIn < entrada.txt
```

y también es posible cambiar la entrada desde dentro del programa

```
InputStream entrada = new FileInputStream("entrada.txt");
System.setIn(entrada);
```

La definición en la clase System es así:

```
public final static InputStream in;
```

## 40. **System.out**

Conecta con la salida estándar de la aplicación, permitiendo escribir en ella. Lo más típico

**System.out.print(x)**

imprime la representación textual de “x”; donde “x” es un valor de algún tipo primitivo, o un objeto, en cuyo caso se imprime x.toString().

**System.out.println(x)**

hace lo mismo que “print(x)” pero, además, cambia a la línea siguiente.

**System.out.printf(formato, x)**

**System.out.format(formato, x)**

llama a ["formatter"](#), imprimiendo el resultado..

<i><b>class SystemOut</b></i>
<pre>public class SystemOut {     public static void main(String[] argumentos) {         String nombre ="Aureliano Buendía";         System.out.print("de nombre: ");         System.out.println(nombre);     } }</pre>

Normalmente lo que se escribe aparece en la pantalla

```
$ java SystemOut
de nombre: Aureliano Buendía
```

Desde el sistema operativo, es fácil para el usuario redirigir la salida a un fichero

```
$ java SystemOut > salida.txt
```

y también es posible cambiar la salida desde dentro del programa

```
PrintStream salida = new PrintStream("salida.txt");
System.setOut(salida);
```

La definición en la clase System es así:

```
public final static PrintStream out;
```

Ver "[System.err](#)".

#### **41. *TreeMap*<K, V> (clase) *java.util.TreeMap*<K, V>**

Asociación tipo Map<K, V> donde los elementos se recorren en orden.

Ver [Map<K, V> \(interface\) java.util.Map<K, V>](#)

#### **42. *TreeSet*<E> (clase) *java.util.TreeSet*<E>**

Conjuntos en donde los elementos se recorren en orden.

Ver [Set<E> \(interface\) java.util.Set<E>](#)

#### **43. *Vector*<E> (clase) *java.util.Vector*<E>**

Listas "normales". Es una variante que se puede considerar obsoleta.

Ver [ArrayList<E> \(clase\) java.util.ArrayList<E>](#)

Ver [List<E> \(interface\) java.util.List<E>](#)

#### **44. *Writer* (clase abstracta) *java.io.Writer***

Madre de un conjunto de clases para escribir ficheros carácter a carácter.

Java convierte los caracteres [unicode](#) en bytes para guardar en el fichero. Vea "[codificación de los caracteres](#)".

La clase derivada más habitual es

<b>java.io.FileWriter</b>	
<code>FileWriter(File file)</code>	constructor
<code>FileWriter (File file, boolean append)</code>	constructor: añade al final
<code>FileWriter (String nombre)</code>	constructor
<code>FileWriter (String nombre, boolean append)</code>	constructor: añade al final

Todas las clases derivadas de `Writer` proporcionan estos métodos:

<b>java.io.Writer</b>	
<code>Writer append(char c)</code>	añade un carácter al final del fichero
<code>void close()</code>	cierra el fichero, asegurando que todo queda bien escrito en el fichero en disco
<code>void flush()</code>	asegura que todos los caracteres quedan bien escritos en el disco, sin cerrar el fichero
<code>void write(char[] chars)</code>	escribe en el fichero el array de caracteres
<code>void write(char[] chars,           int start,           int n)</code>	escribe 'n' caracteres en el fichero, concretamente, los del array 'chars', empezando en la posición 'start'.
<code>void write(String s)</code>	escribe en el fichero la cadena 's'
<code>void write(String s,           int start,           int n)</code>	escribe 'n' caracteres en el fichero, concretamente, los de la cadena 's', empezando en la posición 'start'.

Ejemplo de uso:

```
/**
 * Escritura de caracteres en ficheros.
 *
 * @param fichero nombre del fichero al que queremos escribir.
 * @param chars caracteres que queremos mandar al fichero.
 * @throws IOException salta si hay problemas
 *                      al abrir o al escribir.
 */
public void escribeCaracteres(String fichero, char[] chars)
    throws IOException {
    Writer os = new FileWriter(fichero);
    os.write(chars);
    os.close();
}
```

`OutputStream` escribe caracteres (16 bits). Si necesita escribir bytes (8 bits), debe utilizar la clase [OutputStream](#).

Ver `PrintWriter`, [File](#), [OutputStream](#), [Reader](#).

## **PrintWriter (clase) java.io.PrintWriter**

Clase derivada de [Writer](#), para escribir caracteres en un fichero, proporcionando una serie de métodos que facilitan tareas habituales:

<b>java.io.PrintWriter</b>	
<code>PrintWriter(File file)</code>	constructor
<code>PrintWriter(String fichero)</code>	constructor
<code>PrintWriter(Writer writer)</code>	constructor
<code>PrintWriter(File file, String <a href="#">encoding</a>)</code>	constructor
<code>PrintWriter(String fichero, String <a href="#">encoding</a>)</code>	constructor
<code>PrintWriter format(String format, Object ... args)</code>	imprime los objetos en el fichero ver <a href="#">Formatter</a>
<code>PrintWriter printf(String format, Object ... args)</code>	imprime los objetos en el fichero ver <a href="#">Formatter</a>
<code>void print(boolean b)</code> <code>void print(char c)</code> <code>void print(char[] s)</code> <code>void print(double d)</code> <code>void print(float f)</code> <code>void print(int i)</code> <code>void print(long l)</code> <code>void print(Object object)</code> <code>void print(String s)</code>	imprime el argumento
<code>void println()</code>	imprime un cambio de línea
<code>void println(boolean b)</code> <code>void println(char c)</code> <code>void println(char[] s)</code> <code>void println(double d)</code> <code>void println(float f)</code> <code>void println(int i)</code> <code>void println(long l)</code> <code>void println(Object object)</code> <code>void println(String s)</code>	imprime el argumento, seguido de un cambio de línea

# Diccionario

## 1. Acrónimos

<b>ADT</b>	Abstract Data Type
<b>API</b>	Application Programming Interface
<b>CLI</b>	Command Line Interface
<b>IDE</b>	Integrated Development Environment
<b>JDK</b>	Java Development Kit
<b>JRE</b>	Java Runtime Environment
<b>JVM</b>	Java Virtual Machine
<b>OO</b>	Object Oriented
<b>OOP</b>	Object Oriented Programming
<b>OS</b>	Operating System
<b>POO</b>	Programación Orientada a Objetos (OOP)
<b>SDK</b>	Software Development Kit
<b>SO</b>	Sistema Operativo (OS)
<b>TAD</b>	Tipo Abstracto de Datos (ADT)
<b>UML</b>	Unifid Modelling Language
<b>VM</b>	Virtual Machine

## 2. Términos en inglés

<i>inglés</i>	<i>español</i>
actual argument	<a href="#">argumento</a> real (o concreto)
application	aplicación (o programa)
argument	<a href="#">argumento</a> (o parámetro)
<a href="#">array</a>	
assignment	<a href="#">asignación</a>
body	cuerpo
bug	defecto
<a href="#">casting</a>	
class	<a href="#">clase</a>
code	<a href="#">código</a>
comment	<a href="#">comentario</a>



<b><i>inglés</i></b>	<b><i>español</i></b>
compiler	<a href="#"><u>compilador</u></a>
composition	<a href="#"><u>composición</u></a>
constant	<a href="#"><u>constante</u></a>
constructor	<a href="#"><u>constructor</u></a>
data	datos
debug	depurar
declaration	<a href="#"><u>declaración</u></a>
delegation	<a href="#"><u>delegación</u></a>
<a href="#"><u>downcasting</u></a>	
dynamic method binding	<a href="#"><u>elección dinámica de método</u></a>
dynamic method dispatch	
dynamic method lookup	
encapsulation	<a href="#"><u>encapsulación</u></a>
exception	<a href="#"><u>excepción</u></a>
execution	<a href="#"><u>ejecución</u></a>
factory	<a href="#"><u>fábrica</u></a>
field	<a href="#"><u>campo</u></a>
file	<a href="#"><u>fichero</u></a>
formal argument	<a href="#"><u>argumento</u></a> formal
generics	<a href="#"><u>genéricos</u></a>
getter	<a href="#"><u>método de acceso</u></a>
hiding	<a href="#"><u>ocultación</u></a>
identifier	<a href="#"><u>identificador</u></a>
inheritance	<a href="#"><u>herencia</u></a>
interpreter	<a href="#"><u>intérprete</u></a>
iterator	<a href="#"><u>iterador</u></a>
keyword	<a href="#"><u>palabra reservada</u></a>

<i><b>inglés</b></i>	<i><b>español</b></i>
label	<a href="#"><u>etiqueta</u></a>
list	<a href="#"><u>lista</u></a>
loop	<a href="#"><u>bucle</u></a>
map	<a href="#"><u>mapa, función, tabla, diccionario</u></a>
member	<a href="#"><u>miembro</u></a>
method	<a href="#"><u>método</u></a>
narrowing	<a href="#"><u>reducción</u></a>
object	<a href="#"><u>objeto</u></a>
overflow	<a href="#"><u>desbordamiento</u></a>
overloading	<a href="#"><u>sobrecarga</u></a>
overriding	<a href="#"><u>redefinición</u></a>
package	<a href="#"><u>paquete</u></a>
polymorphism	<a href="#"><u>polimorfismo</u></a>
private	<a href="#"><u>privado</u></a>
program	<a href="#"><u>programa</u></a>
public	<a href="#"><u>público</u></a>
queue	<a href="#"><u>cola</u></a>
refactoring	<a href="#"><u>re-estructuración</u></a>
reference	<a href="#"><u>referencia</u></a>
scope	<a href="#"><u>ámbito</u></a>
semantics	semántica
set	<a href="#"><u>conjunto</u></a>
setter	<a href="#"><u>método de carga</u></a>
short-circuit	<a href="#"><u>cortocircuito</u></a>
signature	<a href="#"><u>signatura</u></a>
source code	<a href="#"><u>código fuente</u></a>
stack	<a href="#"><u>pila</u></a>

<i>inglés</i>	<i>español</i>
statement	sentencia
substitution	<a href="#">sustitución</a>
syntax	sintaxis
test	prueba
type	tipo
type parameter	tipo formal
<a href="#">underflow</a>	
<a href="#">upcasting</a>	
variable	<a href="#">variable</a>
widening	<a href="#">promoción</a>
wrapper	contenedor