Algoritmos y programas

Algoritmo

Secuencia ordenada de pasos que resuelve un problema concreto.

Características

- Corrección
  (sin errores).
- Precisión
  (ausencia de ambigüedades).
- Repetitividad
  (solución genérica de un problema dado).
- Finitud
  (número finito de órdenes no implica finitud).
- Eficiencia
  (temporal [tiempo necesario] y espacial [memoria utilizada])

Programa

Implementación de un algoritmo en un lenguaje de programación

Conjunto ordenado de instrucciones que se dan al ordenador indicándole las operaciones o tareas que ha de realizar para resolver un problema.

Lenguajes de programación

Una **instrucción** es un conjunto de símbolos que representa una orden para el ordenador: la ejecución de una operación con datos.

Las instrucciones se escriben en un lenguaje de programación:

- Se forman con símbolos tomados de un determinado repertorio (componentes léxicos)
- Se construyen siguiendo unas reglas precisas (sintaxis)

Lenguaje máquina
El único que entiende directamente la CPU del ordenador

× Depende del modelo de ordenador

× Repertorio de instrucciones reducido (operaciones muy elementales)

× Muy difícil programar en él (en binario, con cadenas de ceros y unos)

Lenguaje ensamblador
Equivalente al lenguaje máquina, cada línea de código se traduce en una instrucción para la máquina.

✓ Le asocia mnemónicos a las operaciones que entiende la CPU

× Repertorio de instrucciones reducido (operaciones muy elementales)

× Programas difíciles de entender

Lenguajes de alto nivel
Permiten que el programador exprese el procesamiento de datos de forma simbólica, sin tener en cuenta los detalles específicos de la máquina.

✓ Independientes del modelo de ordenador

✓ Proporcionan un mayor nivel de abstracción

Ejemplos de lenguajes de programación de alto nivel

FORTRAN (FORmula TRANslation)
© 1957, IBM (John Backus)
Orientado a la resolución de problemas científicos y técnicos

COBOL (COmmon Business Oriented Language)
© 1959, Codasyl (Committee on Data System Languages)
Aplicaciones comerciales de gestión

LISP (LISt Processing)
© 1959, John McCarthy (MIT)
Procesamiento de datos no numéricos (usado en IA)

BASIC (Beginner's All-purpose Symbolic Instruction Code)
© 1964, John Kemeny & Thomas Kurtz (Darmouth College)
Lenguaje interactivo para principiantes

Simula
© 1967, Ole-Johan Dahl & Krysten Nygaard (Noruega)
Primer lenguaje de programación orientada a objetos

Pascal
© 1971, Niklaus Wirth
Lenguaje estructurado diseñado para aprender a programar

C
© 1972, Denis Ritchie (Bell Labs)
Lenguaje pequeño, flexible y eficiente

Smalltalk
© 1972, Alan Kay (Xerox PARC)
Origen de los interfaces WIMP (Windows, Icons, Mouse & Pull-down menus)

PROLOG (PROgramming in Logic)
© 1972, Alain Colmerauer (Universidad de Marsella)
Basado en Lógica (usado en IA)

Ada
© 1980, US Department of Defense
Basado en Pascal, muy usado en aplicaciones militares

C++
© 1983, Bjarne Stroustroup (AT&T Bell Labs)
Extensión de C que permite la programación orientada a objetos

Java
© 1995, Sun Microsystems
Similar a C++, aunque más sencillo de aprender y usar.

C#
© 2000, Microsoft Corporation
Alternativa de Microsoft a Java, muy similar a éste

Clasificación de los lenguajes de programación de alto nivel

- **Lenguajes imperativos**:
  Los programas indican al ordenador de forma inequívoca los pasos a seguir para la resolución de un problema.

  - **Programación estructurada**:
    La estructura del texto del programa debe auxiliarnos para entender la función que realiza: estrategia "divide y vencerás" (la resolución de un problema se divide en tareas y, éstas, en subtareas).

    Ejemplos: C, Pascal, Fortran...

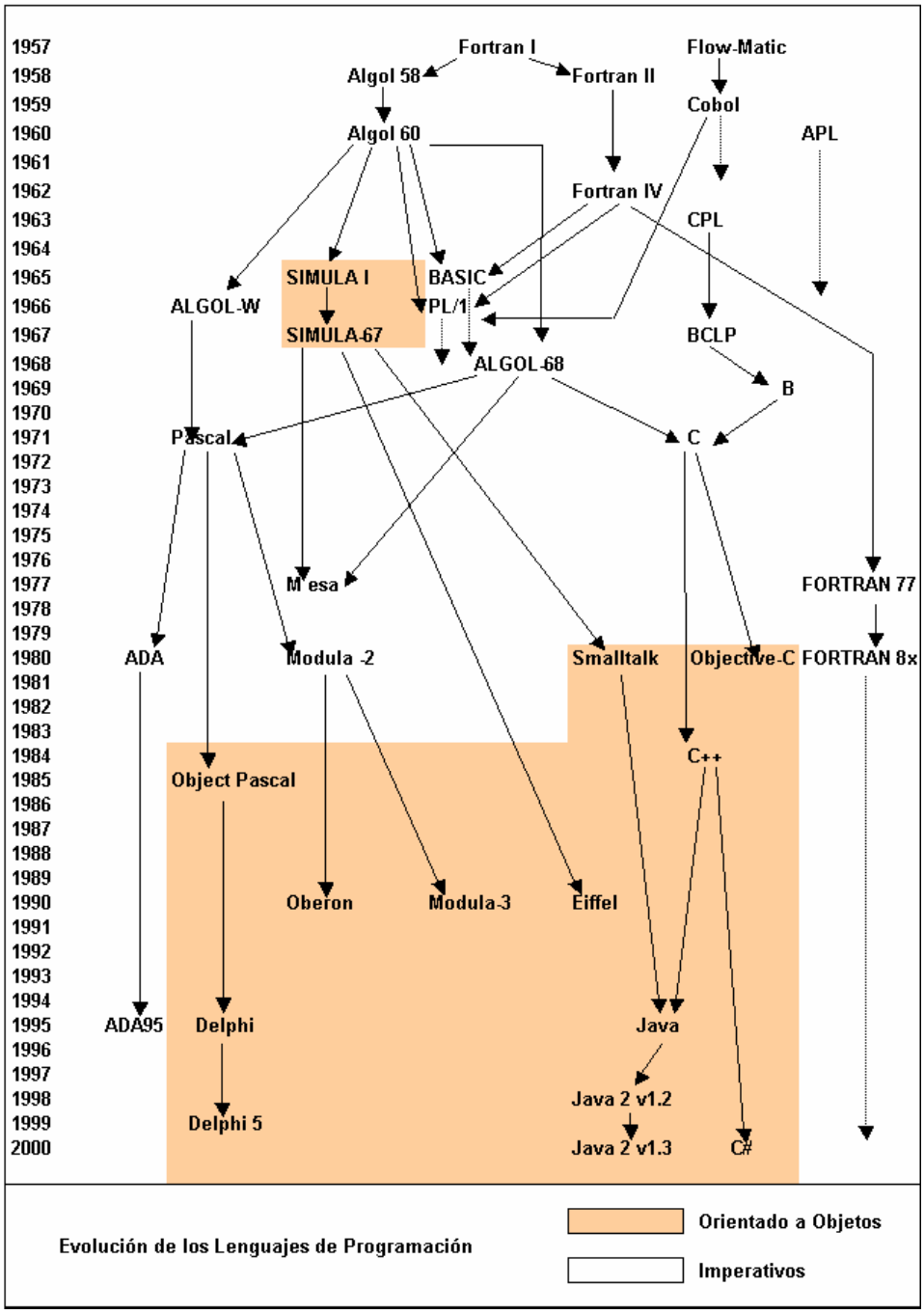
  - P**rogramación orientada a objetos:**
    Estilo de programación que basa la estructura de un programa en módulos deducidos de los tipos de objetos que manipula (en lugar de basarse en las tareas que el sistema debe realizar).

    Ejemplos: Smalltalk, C++, Java, C#...

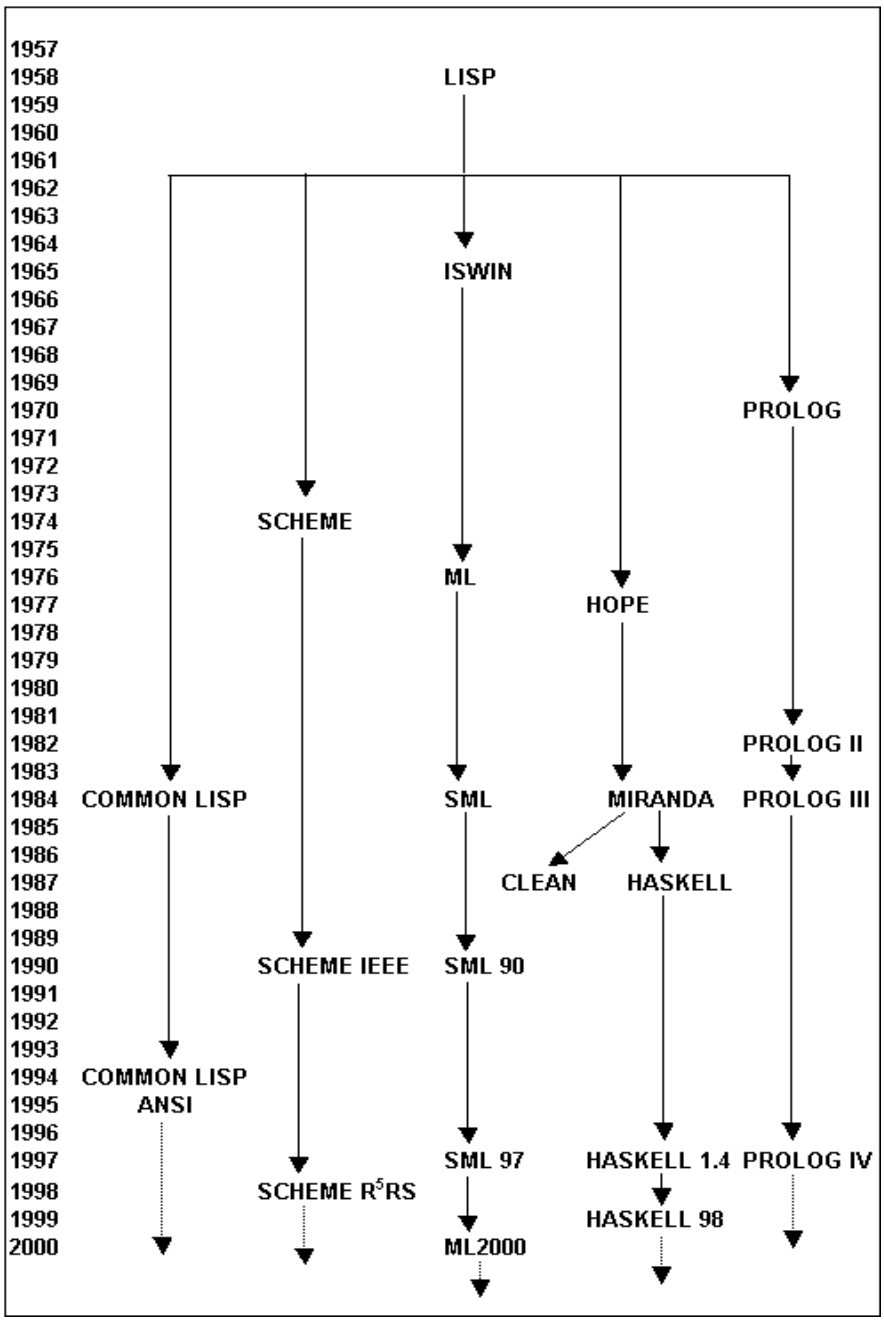
- **Lenguajes declarativos (funcionales y lógicos):**
  Los programas se implementan como conjuntos de funciones (o reglas lógicas) cuya evaluación nos dará el resultado deseado.

  Ejemplos: LISP, PROLOG...

Evolución de los lenguajes de programación: Lenguajes imperativos

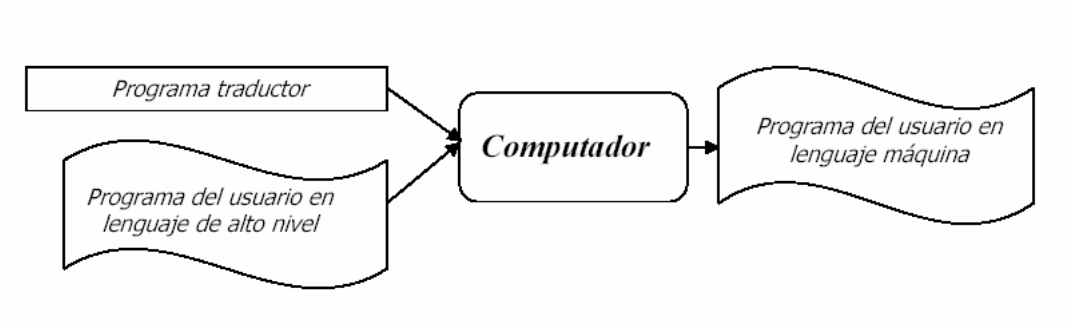


Evolución de los lenguajes de programación: Lenguajes declarativos



Traductores

Los traductores transforman programas escritos en un lenguaje de alto nivel en programas escritos en código máquina:



Tipos de traductores

Compiladores

Generan un programa ejecutable a partir del código fuente

Intérpretes

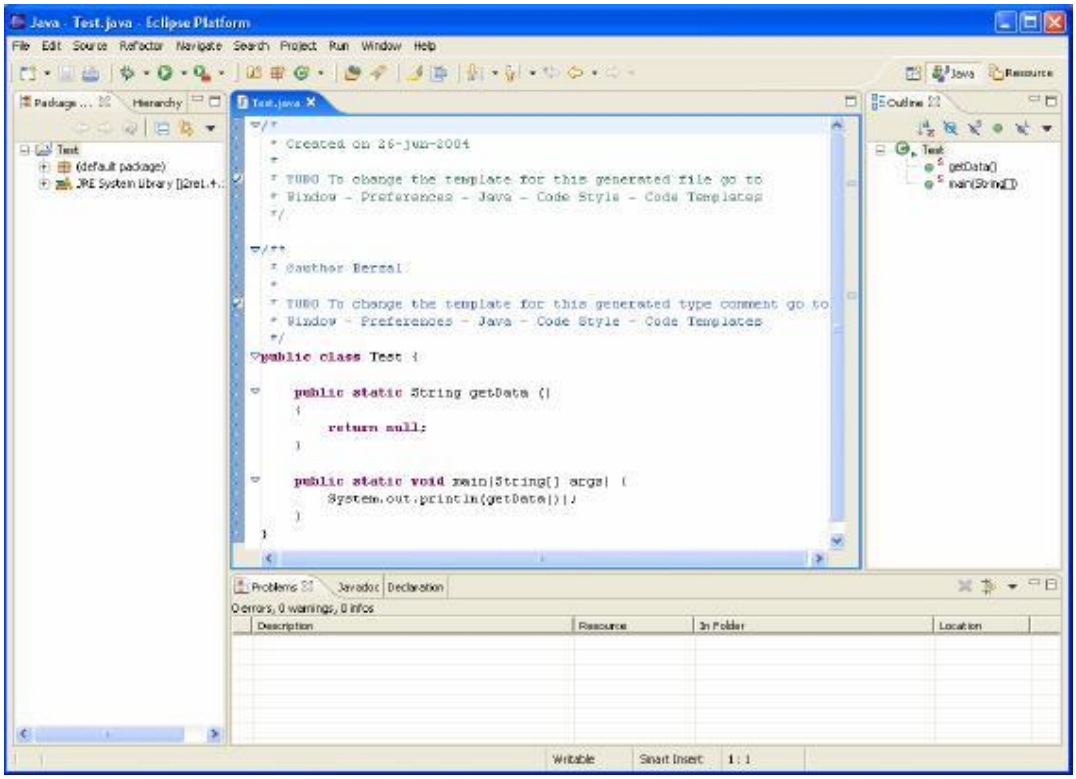
Van analizando, traduciendo y ejecutando las instrucciones del programa una a una. No se traduce una instrucción hasta que la ejecución de la anterior haya finalizado.

Herramientas de programación

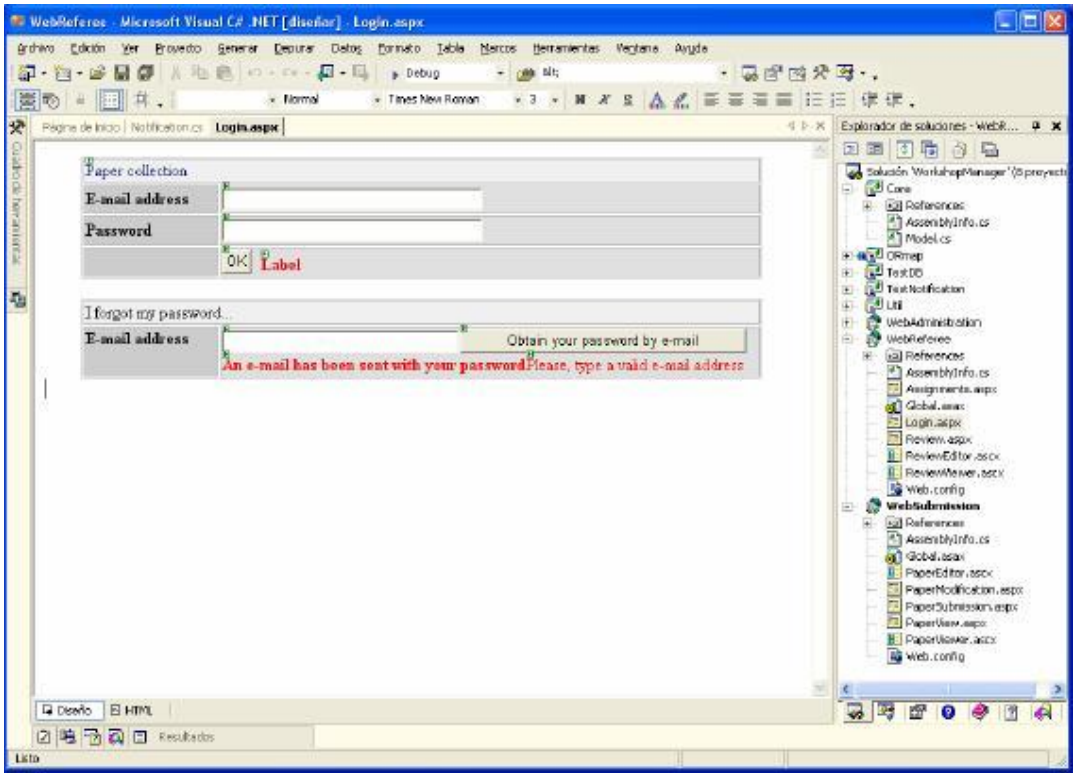
Editores, depuradores, profilers...

↳ IDEs (entornos integrados de desarrollo)

Ejemplos Microsoft Visual Studio .NET
Borland C++Builder/Delphi
Eclipse
…



Eclipse, un IDE para Java (http://www.eclipse.org)



Microsoft Visual Studio .NET, un IDE para la plataforma .NET

Desarrollo de aplicaciones informáticas: Ciclo de vida del software

El ciclo de vida de una aplicación comprende las siguientes etapas:

- ✓ **Planificación**
  Delimitación del ámbito del proyecto, estudio de viabilidad, análisis de riesgos, estimación de costos, planificación temporal y asignación de recursos.

- ✓ **Análisis** (¿qué?): Elicitación de requisitos.
  Descripción clara y completa de qué es lo que se pretende, incluyendo la presentación de los resultados que se desean obtener (formato de las salidas) y la forma en que se va a utilizar la aplicación (interfaz de usuario)

- ✓ **Diseño** (¿cómo?): Estudio de alternativas
  - *Diseño arquitectónico*: Organización de los distintos módulos que compondrán la aplicación (diseño arquitectónico).
  - *Diseño detallado*: Definición de los algoritmos necesarios para implementar la aplicación en lenguaje natural, mediante diagramas de flujo o en pseudocódigo [lenguaje algorítmico].

- ✓ **Implementación**:
  Adquisición de componentes, creación de los módulos de la aplicación en un lenguaje de programación e integración de los recursos necesarios para que el sistema funcione.

- ✓ **Depuración y pruebas**:
  Comprobación del funcionamiento de la aplicación
  Pruebas de unidad y de integración, pruebas alfa, pruebas beta, test de aceptación.
  - *Verificación* (si se está realizando lo que se pretendía)
  - *Validación* (si se realiza lo correcto).

- ✓ **Explotación**: Uso y mantenimiento
  - *Mantenimiento correctivo*: Corrección de defectos o errores.
  - *Mantenimiento adaptativo*: Adaptación de la aplicación a nuevas circunstancias e inclusión de nuevas prestaciones.

Introducción a la programación

Java

La plataforma de programación Java

- Historia
- La máquina virtual Java
- Herramientas de programación en Java

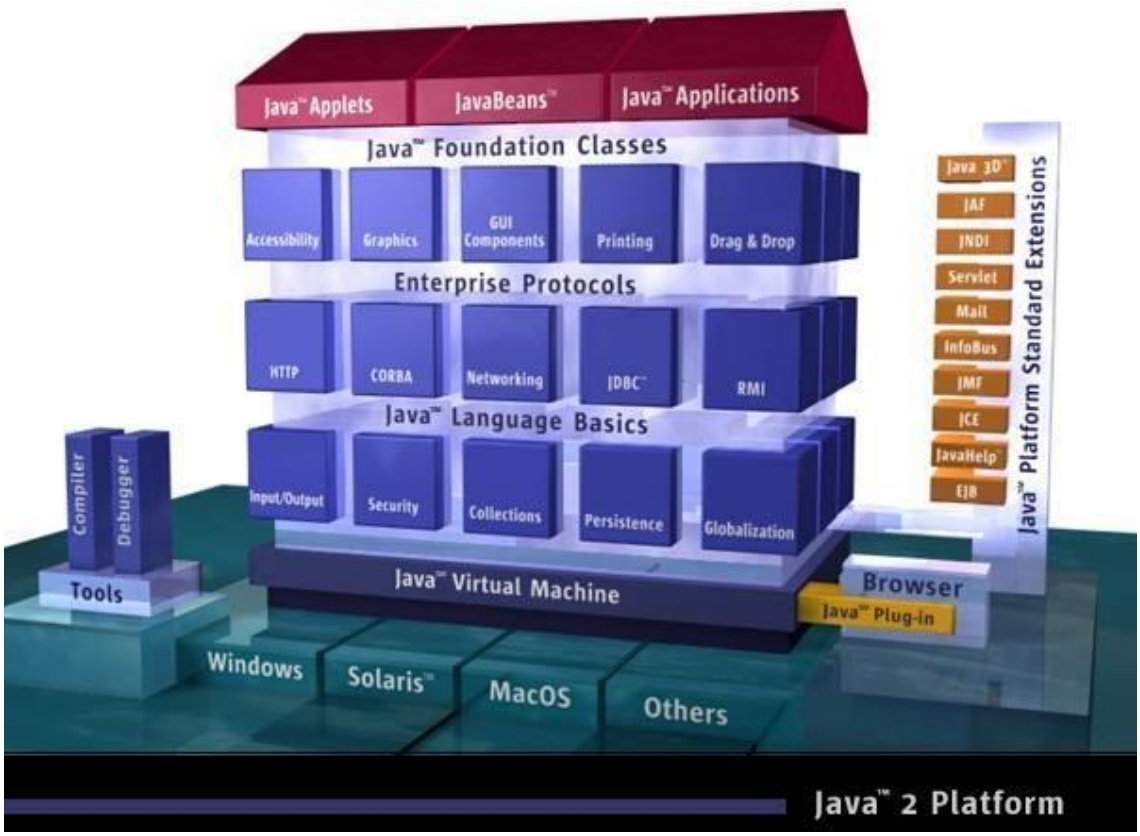
Aplicaciones y applets

- Aplicación de ejemplo
- Applet de ejemplo
- Fases en la creación y ejecución de programas en Java

Características clave de Java

- Mitos y realidades

La plataforma Java



- La **máquina virtual Java**
  (JVM: Java Virtual Machine)

  Imprescindible para poder ejecutar aplicaciones Java.

- Las **bibliotecas** estándar de Java
  (Java Application Programming Interface = Java API)

  Amplia colección de componentes.

- El **lenguaje de programación Java**

  Para escribir aplicaciones.

Historia de Java

Hay versiones distintas sobre el origen, concepción y desarrollo de Java, desde la que dice que éste fue un proyecto que estuvo durante mucho tiempo por distintos departamentos de Sun sin que nadie le prestara atención hasta la más difundida, que presenta a Java como un lenguaje pensado para pequeños electrodomésticos:

Hace algunos años, Sun Microsystems decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Sun decidió crear una filial, denominada FirstPerson Inc..

El mercado inicialmente previsto para los programas de FirstPerson eran los equipos domésticos: microondas, tostadoras y, fundamentalmente, televisores interactivos. En este mercado, dada la falta de pericia de los usuarios, se requerían unos interfaces mucho más cómodos e intuitivos que los sistemas de ventanas del momento.

James Gosling decidió que las ventajas aportadas por la eficiencia de C++ no compensaban el gran coste de la prueba y depuración de aplicaciones C++. Gosling había estado trabajando en un lenguaje de programación que él había llamado **Oak**, el cual, aún partiendo de la sintaxis de C++, intentaba remediar las deficiencias que iba observando.

El primer proyecto en que se aplicó este lenguaje recibió el nombre de proyecto Green y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un hogar.

Para ello se construyó un ordenador experimental denominado ***7** (Star Seven). El sistema presentaba una interfaz basada en la representación de la casa de forma animada y el control se llevaba a cabo mediante una pantalla sensible al tacto. En el sistema aparecía Duke, la mascota de Java.

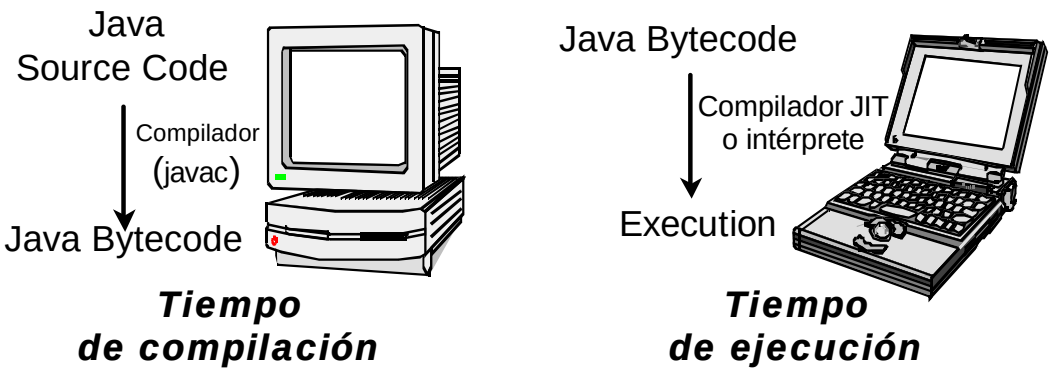
Posteriormente, se aplicó a otro proyecto de VoD (Video On Demand) en el que se empleaba como interfaz para la televisión interactiva. Ninguno de estos proyectos se convirtió nunca en un sistema comercial.

Cuando en Sun se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito, urgieron a FirstPerson a desarrollar con rapidez nuevas estrategias que produjeran beneficios. No lo consiguieron y FirstPerson cerró en la primavera de 1994.

A pesar de este fracaso, Bill Joy, cofundador de Sun y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet podía llegar a ser el terreno adecuado para disputar a Microsoft su primacía casi absoluta en el terreno del software y vio en Oak el instrumento idóneo para llevar a cabo estos planes. Tras un cambio de nombre, al estar Oak ya registrado como marca, el lenguaje Java fue presentado en sociedad en mayo de 1995 (Sun World'95).

http://java.sun.com/features/1998/05/birthday.html

La máquina virtual Java



- El **compilador de Java** genera un código intermedio independiente de la plataforma (bytecodes).

- Los **bytecodes** pueden considerarse como el lenguaje máquina de una máquina virtual, la Máquina Virtual Java (JVM).

- Cuando queremos **ejecutar una aplicación Java**, al cargar el programa en memoria, podemos

  a) Interpretar los bytecodes instrucción por instrucción

  b) Compilar los bytecodes para obtener el código máquina necesario para ejecutar la aplicación en el ordenador (compilador JIT [Just In Time]).

De esta forma, podemos ejecutar un programa escrito en Java sobre distintos sistemas operativos (Windows, Solares, Linux...) sin tener que recompilarlo, como sucedería con programas escritos en lenguajes como C.

Uso típico de Java

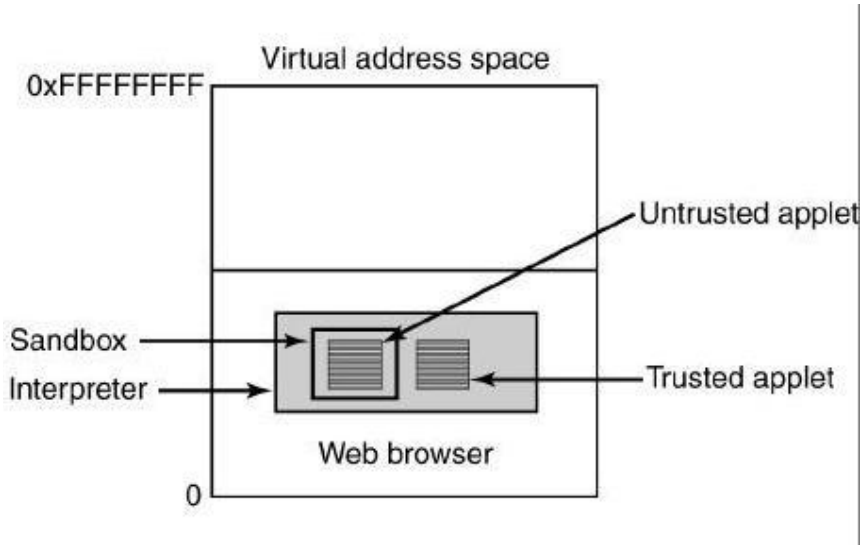
Distribución de aplicaciones a través de Internet

- Aplicaciones (programas independientes)
- Applets (“pequeñas aplicaciones”)

Applets

- Los applets son programas diseñados para ejecutarse como parte de una página web.
- Java impone restricciones de seguridad para que los applets no puedan “dañar” el ordenador en que se ejecutan

  Ejemplos: Acceder a ficheros locales
  Ejecutar otro programa
  Conectarse a otro ordenador desde el nuestro.



- Si los applets se compilasen directamente al código máquina de un ordenador concreto, las personas que accediesen a la página web que contiene el applet desde un ordenador de otro tipo **no podrían ejecutar el applet**.

Herramientas de programación en Java

Java SDK [Software Development Kit]

http://java.sun.com

- Compilación de aplicaciones Java: `javac`
- Ejecución de aplicaciones Java: `java`
- Ejecución de applets: `appletviewer`
- Generación de documentación: `javadoc`
- Creación de archivos de distribución JAR [Java ARchives]: `jar`
- Depuración de aplicaciones Java: `jdb`
- Desensamblador para la máquina virtual Java: `javap`
- Generador de stubs en C: `javah`

...

Versiones

1995 JDK 1.02
1996 JDK 1.1
1998 JDK 1.2 (Java 2 SDK v1.2)
2000 JDK 1.3 (Java 2 SDK v1.3)
2002 JDK 1.4 (Java 2 Platform, Standard Edition v1.4)
2004 JDK 1.5 (Java 2 Platform, Standard Edition 5.0)

Ediciones

J2SE (Standard Edition): Aplicaciones y applets
J2EE (Enterprise Edition): Servidores de aplicaciones
J2ME (Micro Edition): Aplicaciones para dispositivos móviles

Entornos integrados de desarrollo: IDEs
[Integrated Development Environments]

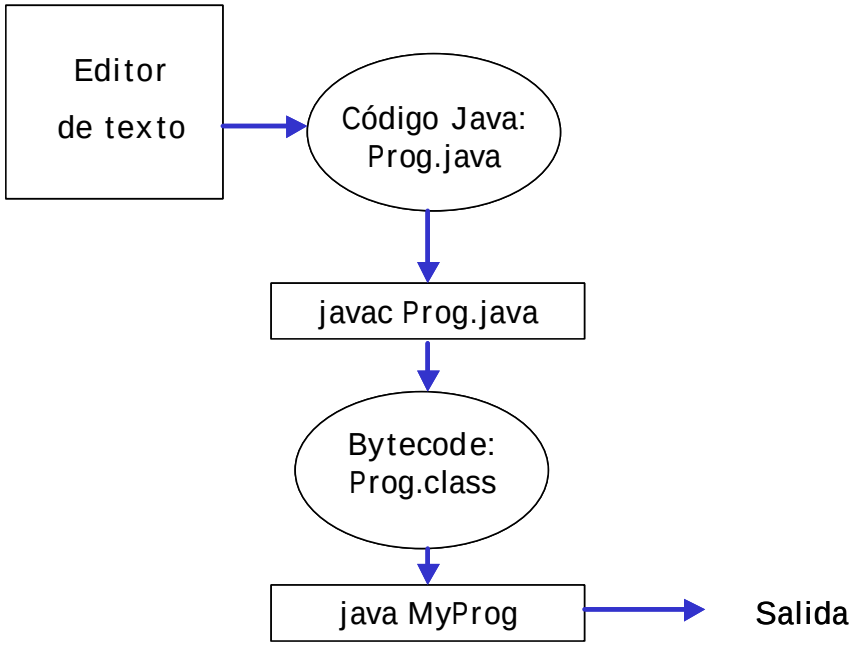
Gratuitos

- Eclipse (http://www.eclipse.org)
- NetBeans (http://java.sun.com)

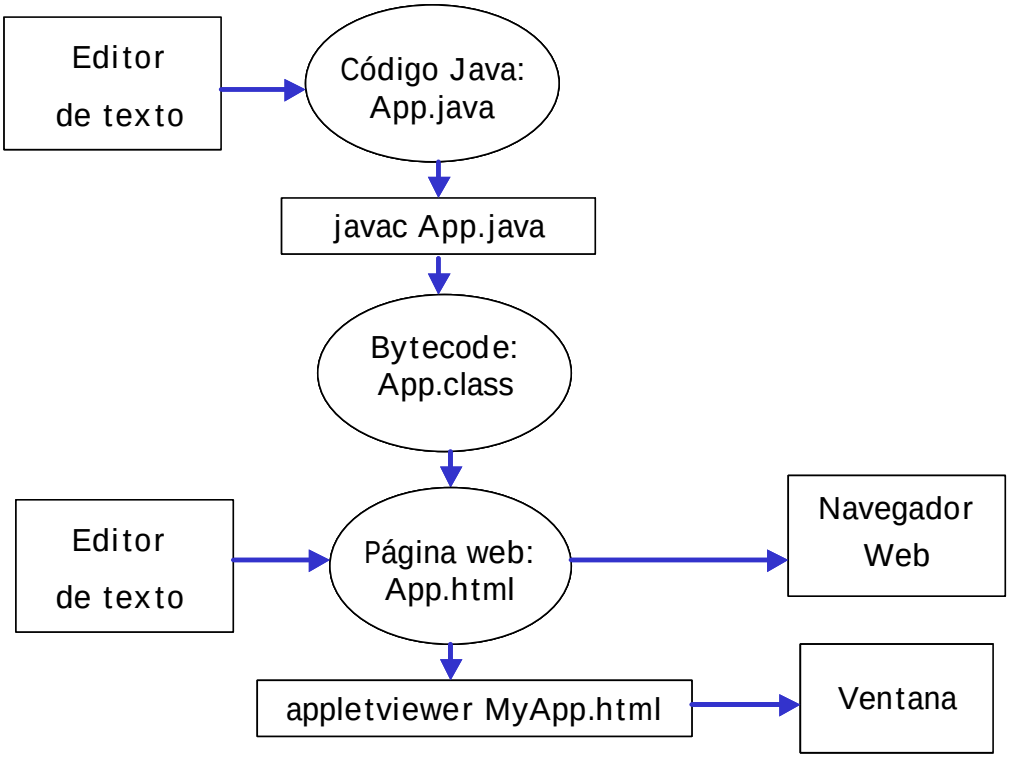
De pago

- Borland JBuilder
- IBM Visual Age for Java
- Sybase PowerJ
- Metrowerks CodeWarrior

Creación y ejecución de aplicaciones Java



Creación y ejecución de applets



Aplicación de ejemplo

Código Java: Fichero Programa.java

```
public class Programa
{
    public static void main (String[] args)
    {
        System.out.println("Hola");
    }
}
```

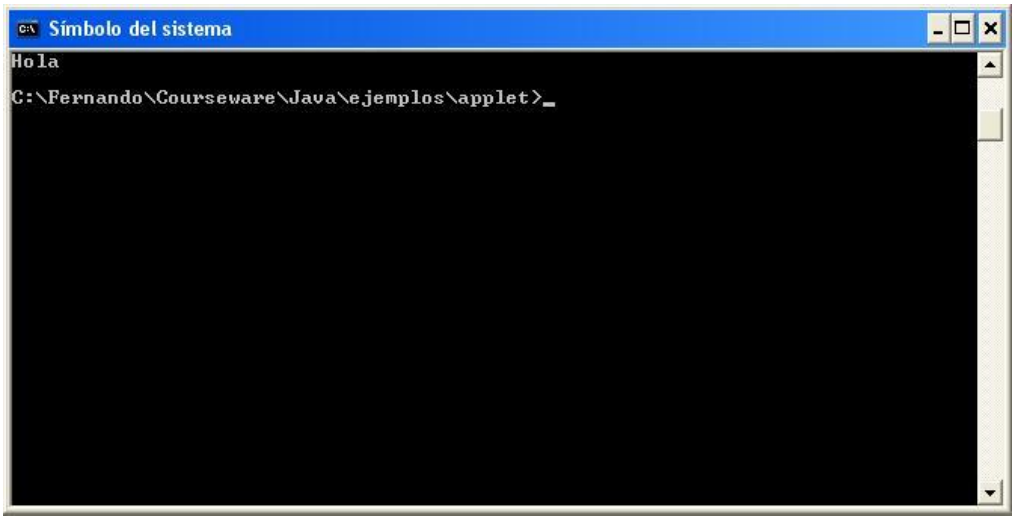
Compilación

```
javac Programa.java
```

Ejecución

```
java Programa
```

Resultado



Applet de ejemplo

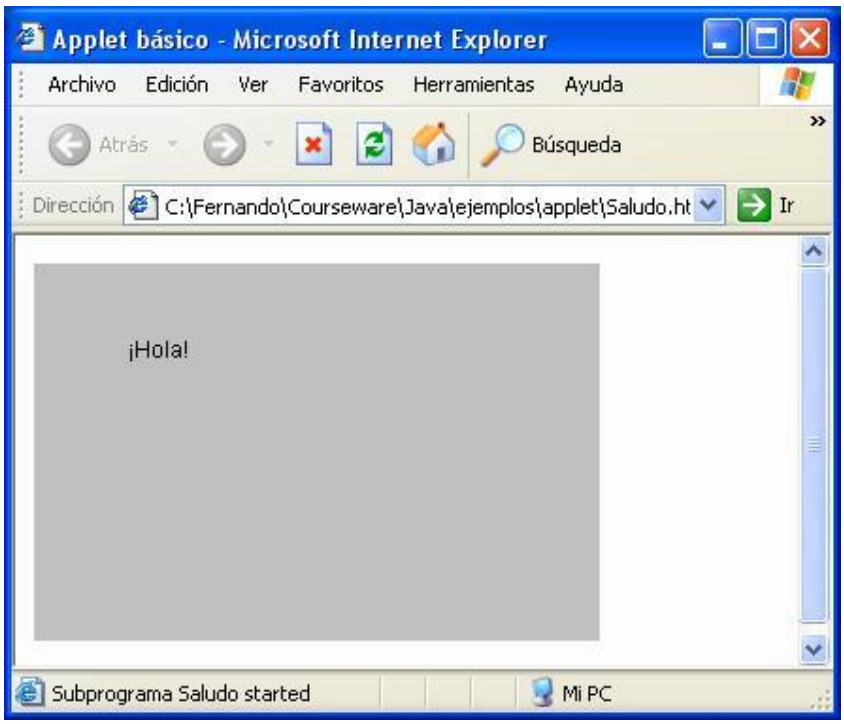
Código Java: Fichero Saludo.java

```
import java.awt.*;
import java.applet.Applet;

public class Saludo extends Applet
{
    public void paint(Graphics g) {
        g.drawString("¡Hola!", 50, 50);
    }
}
```

Página web: Fichero Saludo.html

```
<html>
  <head>
    <title>Applet básico</title>
  </head>
  <body>
    <applet code="Saludo" width=300 height=200>
    </applet>
  </body>
</html>
```

Resultado



Fases en la creación y ejecución de programas en Java

Fase I: Editor

- Se crea un programa con la ayuda de un editor
- Se almacena en un fichero con extensión .java

Fase II: Compilador

- El compilador lee el código Java (fichero .java)
- Si se detectan errores sintácticos, el compilador nos informa de ello.
- Se generan los bytecodes, que se almacenan en ficheros .class

Fase III: Cargador de clases

El cargador de clases lee los bytecodes (ficheros .class):
Los bytecodes pasan de disco a memoria principal.

Fase IV: Verificador de bytecodes

El verificador de bytecodes comprueba que los bytecodes son válidos y no violan las restricciones de seguridad de la máquina virtual Java.

Fase V: Intérprete de bytecodes o compilador JIT

La máquina virtual Java (JVM) lee los bytecodes y los traduce al lenguaje que el ordenador entiende (código máquina).

NOTA: Conforme se ejecuta el programa, se hace uso de la memoria principal para almacenar los datos con los que trabaja la aplicación.

Java es multiplataforma

Los programas escritos en Java se compilan en un bytecode independiente de la máquina y todos los sistemas operativos principales tienen entornos de ejecución de aplicaciones Java [máquinas virtuales].

NOTA: La idea no es nueva (p.ej. UCSD Pascal)

Java es seguro

Pueden forzarse restricciones sobre las operaciones permitidas (los applets no acceden directamente al hardware de la máquina).

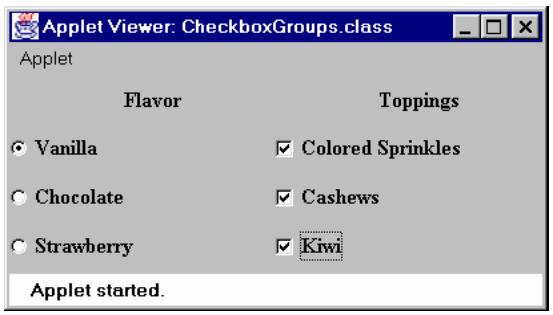
Al cargar un programa en memoria, la máquina virtual Java verifica los bytecodes de la aplicación.

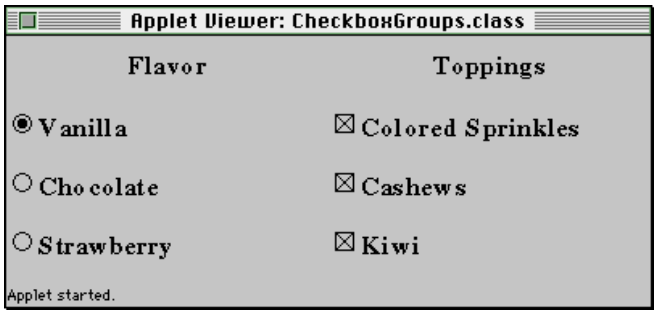
Java tiene un amplio conjunto de bibliotecas estándar

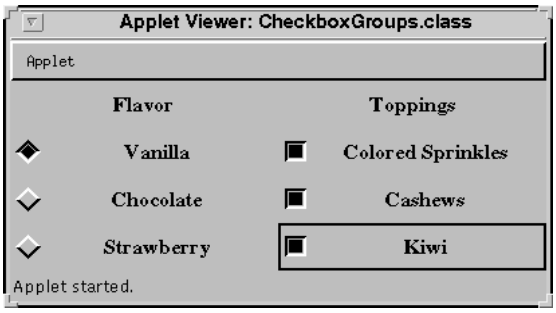
Bibliotecas para trabajar con colecciones y otras estructuras de datos, ficheros, acceso a bases de datos (JDBC), interfaces gráficas de usuario (JFC/Swing), redes de ordenadores (RMI, Jini), aplicaciones distribuidas (EJB), interfaces web (servlets/JSP), hebras, compresión de datos, criptografía...

Java incluye una biblioteca portable para la creación de interfaces gráficas de usuario (AWT en Java 1.0/1.1 y JFC/Swing en Java 2).

"Look & feel" en función del sistema operativo:







Java simplifica algunos aspectos a la hora de programar

- Gestión automática de memoria (recolector de basura).
- Comprobación estricta de tipos
- Sintaxis simplificada con respecto a C++.
  - No se manejan punteros explícitamente (todo son punteros en realidad).
  - No hay que crear makefiles (como en C/C++).
  - No hay que mantener ficheros de cabecera aparte (como en C/C++).
  - No existen macros (#define en C/C++), ya que son propensas a errores.

Mitos y realidades de Java
www.corewebprogramming.com

Mito: Java es un lenguaje de programación para la web.
Realidad: Java es un lenguaje de programación de propósito general.

Uso estimado de Java:
5% applets (clientes web)
45% aplicaciones de escritorio (PCs)
50% aplicaciones en el servidor (servlets/EJB)

Mito: "Write once, run anywhere"
Realidad: Se puede conseguir, aunque se debe comprobar.

Motivos: Las aplicaciones Java pueden ejecutar código local (nativo), las interfaces gráficas pueden comportarse de forma ligeramente distinta en distintas plataformas...

Mito: Java es un lenguaje interpretado.
Realidad: Los compiladores JIT compilan el programa al cargarlo.

Mito: La seguridad y la independencia de la máquina "son gratis".
Realidad: Aplicaciones un 20% más lentas que en C++.

Mito: Java acabará con X (donde X puede ser Microsoft, C++...)
Realidad: Siempre existen ventajas y desventajas.

Microsoft tiene su propia alternativa: la plataforma .NET

Determinadas aplicaciones es mejor escribirlas en otros lenguajes:
- Utilidades simples y eficientes en ANSI C,
- Sistemas complejos de altas prestaciones en C++,
- Aplicaciones para Windows con Visual Basic .NET o C#...

Datos y tipos de datos

Dato

Representación formal de hechos, conceptos o instrucciones adecuada para su comunicación, interpretación y procesamiento por seres humanos o medios automáticos.

Tipo de dato

Especificación de un dominio (rango de valores) y de un conjunto válido de operaciones a los que normalmente los traductores asocian un esquema de representación interna propio.

Clasificación de los tipos de datos

En función de quién los define:

- Tipos de datos estándar
- Tipos de datos definidos por el usuario

En función de su representación interna:

- Tipos de datos escalares o simples
- Tipos de datos estructurados

Codificación de los datos en el ordenador

En el interior del ordenador, los datos se representan en binario.

El sistema binario sólo emplea dos símbolos: 0 y 1

- Un bit nos permite representar 2 símbolos diferentes: 0 y 1
- Dos bits nos permiten codificar 4 símbolos: 00, 01, 10 y 11
- Tres bits nos permiten codificar 8 símbolos distintos: 000, 001, 010, 011, 100, 101, 110 y 111

En general,

con N bits podemos codificar $\mathbf{2^N}$ valores diferentes

N	2^N
1	**2**
2	**4**
3	**8**
4	**16**
5	**32**
6	**64**
7	**128**
8	**256**
9	**512**
10	**1024**
11	**2048**
12	**4096**
13	**8192**
14	**16384**
15	**32768**
16	**65536**

Si queremos representar X valores diferentes, necesitaremos N bits, donde N es el menor entero mayor o igual que $\mathbf{\log_2 X}$

Representación posicional

Un número se representa mediante un conjunto de cifras, cuyo valor depende de la cifra en sí y de la posición que ocupa en el número

NÚMEROS ENTEROS

Ejemplo: Si utilizamos 32 bits para representar números enteros, disponemos de 2^{32} combinaciones diferentes de 0s y 1s:

4 294 967 296 valores.

Como tenemos que representar números negativos y el cero, el ordenador será capaz de representar

del –2 147 483 648 al +2 147 483 647.

Con 32 bits no podremos representar números más grandes.

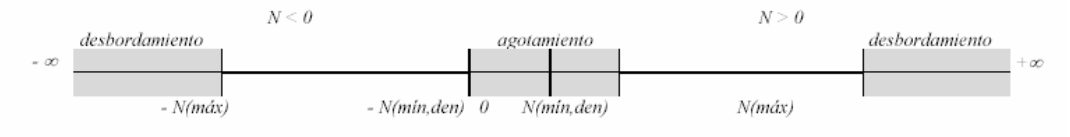
¡¡¡ 2 147 483 647 + 1 = –2 147 483 648 !!!

NÚMEROS REALES (en notación científica)

(+|–) mantisa x $2^{\text{exponente}}$

× El ordenador sólo puede representar un subconjunto de los números reales (números en coma flotante)

× Las operaciones aritméticas con números en coma flotante están sujetas a errores de redondeo.



Estándar IEEE 754

- Precisión sencilla
  (bit de signo + 8 bits exponente + 23 bits mantisa)
- Precisión doble
  (bit de signo + 11 bits exponente + 52 bits mantisa)

Representación de textos

Se escoge un conjunto de caracteres: alfabéticos, numéricos, especiales (separadores y signos de puntuación), gráficos y de control (por ejemplo, retorno de carro).

Se codifica ese conjunto de caracteres utilizando n bits.
Por tanto, se pueden representar hasta **2^n símbolos** distintos.

Ejemplos de códigos normalizados

ASCII (American Standard Code for Information Interchange)

- ANSI X3.4-1968, 7 bits (128 símbolos)
- ISO 8859-1 = Latin-1, 8 bits (256 símbolos)

		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	96	`	a	b	c	d	e	f	g	h	i	j	k	L	m	n	o
70	112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
80	128																
90	144																
A0	160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B0	176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	Ì	í	î	ï
F0	240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	Ü	ý	þ	ÿ

UNICODE, ISO/IEC 10646, 16 bits (65536 símbolos)

Zona	*Códigos*		*Símbolos codificados*	*Nº de caracteres*
A	0000	0000 00FF	Latín-1	256
			otros alfabetos	7.936
		2000	Símbolos generales y caracteres fonéticos chinos, japoneses y coreanos	8.192
I	4000		Ideogramas	24.576
O	A000		Pendiente de asignación	16.384
R	E000 FFFF		Caracteres locales y propios de los usuarios. Compatibilidad con otros códigos	8.192

Tipos de datos primitivos en Java

El lenguaje Java define 8 tipos de datos primitivos:

```
byte     short    int      long
float    double   char     boolean
```

Datos de tipo numérico	
- Números enteros	`byte, short, int, long`
- Números en coma flotante	`float, double`
Datos de tipo carácter	`char`
Datos de tipo booleano	`boolean`

Números enteros
`byte, short, int, long`
4 tipos básicos para representar números enteros (con signo):

Tipo de dato	Espacio en memoria	Valor mínimo	Valor Máximo
`byte`	8 bits	-128	127
`short`	16 bits	-32768	32767
`int`	32 bits	-2147483648	2147483647
`long`	64 bits	-9223372036854775808	9223372036854775807

Literales enteros

Los literales enteros pueden expresarse:

- En decimal (base 10): `255`
- En octal (base 8): `0377` ($3 \cdot 8^2 + 7 \cdot 8^1 + 7 = 255$)
- En hexadecimal (base 16): `0xff` ($15 * 16^1 + 15 = 255$)

Los literales enteros son de tipo `int` por defecto (entre -2^{31} y $2^{31}-1$).

Un literal entero es de tipo `long` si va acompañado del sufijo `l` o `L`:

`1234567890L` es de tipo `long`

NOTA: Se prefiere el uso de `L` porque `l` (L minúscula) puede confundirse con `1` (uno).

Definición
Literal: Especificación de un valor concreto de un tipo de dato.

Operaciones con números enteros

Desbordamiento

Si sobrepasamos el valor máximo que se puede representar con un tipo de dato entero, nadie nos avisa de ello: en la ejecución de nuestro programa obtendremos un resultado incorrecto.

Tipo	Operación	Resultado
`byte`	127 + 1	-128
`short`	32767 + 1	-32768
`int`	2147483647 + 1	-2147483648

Para obtener el resultado correcto, hemos de tener en cuenta el rango de valores de cada tipo de dato, de tal forma que los resultados intermedios de un cálculo siempre puedan representarse correctamente:

Tipo	Operación	Resultado
`int`	1000000 * 1000000	-727379968
`long`	1000000 * 1000000	1000000000000

División por cero

Si dividimos un número entero por cero,
se produce un error en tiempo de ejecución:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at …
```

La ejecución del programa termina de forma brusca al intentar hacer la división por cero.

Números en coma flotante

`float, double`

Según el estándar IEEE 754-1985

Tipo de dato	Espacio en memoria	Mínimo (valor absoluto)	Máximo (valor absoluto)	Dígitos significativos
`float`	32 bits	1.4×10^{-45}	3.4×10^{38}	6
`double`	64 bits	4.9×10^{-324}	1.8×10^{308}	15

Literales reales

- Cadenas de dígitos con un punto decimal

```
123.45   0.0      .001
```

- En notación científica (mantisa·$10^{\text{exponente}}$)

```
123e45   123E+45  1E-6
```

Por defecto, los literales reales representan valores de tipo `double`

Para representar un valor de tipo `float`, hemos de usar el sufijo `f` o `F`:

```
123.45F  0.0f     .001f
```

Operaciones con números en coma flotante

Las operaciones aritméticas en coma flotante no generan excepciones, aunque se realicen operaciones ilegales:

- Cuando el resultado de una operación está fuera de rango, se obtiene `+Infinity` o `–Infinity` (“infinito”).
- Cuando el resultado de una operación está indeterminado, se obtiene `NaN` (“Not a Number”)

Operación	Resultado
1.0 / 0.0	`Infinity`
-1.0 / 0.0	`-Infinity`
0.0 / 0.0	`NaN`

Operadores aritméticos

Java incluye cinco operadores para realizar operaciones aritméticas:

Operador	Operación
`+`	Suma
`-`	Resta o cambio de signo
`*`	Multiplicación
`/`	División
`%`	Módulo (resto de la división)

- Si los operandos son enteros, se realizan operaciones enteras.
- En cuanto uno de los operandos es de tipo `float` o `double`, la operación se realiza en coma flotante.
- No existe un operador de exponenciación: para calcular x^a hay que utilizar la función `Math.pow(x,a)`

División (/)

Operación	Tipo	Resultado
7 / 3	`int`	`2`
7 / 3.0f	`float`	`2.333333333f`
5.0 / 2	`double`	`2.5`
7.0 / 0.0	`double`	`+Infinity`
0.0 / 0.0	`double`	`NaN`

- Si se dividen enteros, el resultado es entero y el resto se pierde.
- Una división entera por cero produce una excepción.
- Una división por cero, en coma flotante, produce `Infinite` o `NaN`.

Módulo (%): Resto de dividir

Operación	Tipo	Resultado
7 % 3	`int`	`1`
4.3 % 2.1	`double`	`~ 0.1`

Expresiones aritméticas
Se pueden combinar literales y operadores
para formar expresiones complejas.

Ejemplo

$$\frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9(\frac{4}{x} + \frac{9+x}{y})$$

En Java se escribiría así:

```
(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)
```

- Las expresiones aritméticas se evalúan **de izquierda a derecha**.

- Los operadores aritméticos mantienen el orden de **precedencia** habitual (multiplicaciones y divisiones antes que sumas y restas).

- Para especificar el orden de evaluación deseado, se utilizan paréntesis.

> NOTA: Es recomendable utilizar paréntesis para eliminar interpretaciones erróneas y posibles ambigüedades

Precisión
Las operaciones en coma flotante no son exactas debido a la forma en que se representan los números reales en el ordenador

Operación	Resultado
1 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1	`0.5000000000000001`
1.0 - 0.9	`0.09999999999999998`

Definición
Expresión: Construcción que se evalúa para devolver un valor.

Caracteres
`char`
http://www.unicode.org/

Tipo de dato	Espacio en memoria	Codificación
`char`	16 bits	UNICODE

Literales de tipo carácter

Valores entre comillas simples

```
'a' 'b' 'c' …   '1' '2' '3' …   '*' …
```

Códigos UNICODE (en hexadecimal): `\u????`

`'\u000a'` (avance de línea)

`'\u000d'` (retorno de carro)

Secuencias de escape para representar caracteres especiales:

Secuencia de escape	Descripción
\t	Tabulador (tab)
\n	Avance de línea (new line)
\r	Retorno de carro (carriage return)
\b	Retroceso (backspace)
\'	Comillas simples
\"	Comillas dobles
\\	Barra invertida

La clase `Character` define funciones (métodos estáticos) para trabajar con caracteres:

```
isDigit(), isLetter(), isLowerCase(), isUpperCase()
toLowerCase(), toUpperCase()
```

Cadenas de caracteres

La clase `String`

- `String` no es un tipo primitivo, sino una clase predefinida
- Una cadena (`String`) es una secuencia de caracteres
- Las cadenas de caracteres, en Java, son inmutables: no se pueden modificar los caracteres individuales de la cadena.

Literales

Texto entra comillas dobles “ ”

```
“Esto es una cadena”
“‘Esto’ también es una cadena”
```

Las secuencias de escape son necesarias para introducir determinados caracteres dentro de una cadena:

```
“\”Esto es una cadena entre comillas\””
```

Concatenación de cadenas de caracteres

El operador + sirve para concatenar cadenas de caracteres

Operación	Resultado
"Total = " + 3 + 4	`Total = 34`
"Total = " + (3+4)	`Total = 7`

Si cualquier operando es un `String`,
toda la operación se convierte en una concatenación de cadenas.

- En Java, cualquier cosa puede convertirse automáticamente en una cadena de caracteres (un objeto de tipo `String`)

Datos de tipo booleano

`boolean`

Representan algo que puede ser verdadero (`true`) o falso (`false`)

	Espacio en memoria	Valores
`boolean`	1 bit	Verdadero o falso

Literales

Literal	Significado
`true`	Verdadero (1)
`false`	Falso (0)

Expresiones de tipo booleano

- Se construyen a partir de expresiones de tipo numérico con **operadores relacionales**.
- Se construyen a partir de otras expresiones booleanas con **operadores lógicos o booleanos**.

Operadores relacionales

- Operadores de comparación válidos para números y caracteres
- Generan un resultado booleano

Operador	Significado
==	Igual
!=	Distinto
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual

Operadores lógicos/booleanos

- Operandos booleanos.
- Tienen menos precedencia que los operadores de comparación.

Operador	Nombre	Significado
!	NOT	Negación lógica
&&	AND	'y' lógico
||	OR	'o' inclusivo
^	XOR	'o' exclusivo

Tablas de verdad

X	!X
`true`	`false`
`False`	`true`

A	B	A&&B	A||B	A^B
`false`	`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`	`True`
`true`	`false`	`false`	`true`	`True`
`true`	`true`	`true`	`True`	`False`

- NOT (!) cambia el valor booleano.
- AND (&&) devuelve `true` si los dos son operandos son `true`.
  No evalúa el segundo operando si el primero es `false`
- OR (||) devuelve `false` si los dos son `false`.
  No evalúa el segundo operando si el primero es `true`
- XOR (^) devuelve `true` si los dos operandos son diferentes.
  Con operandos booleanos es equivalente a `!=`

Ejemplos

Número x entre 0 y 10 `(0 <= x) && (x <= 10)`

Número x fuera del intervalo [0,10] `!((0 <= x) && (x <= 10))`

o bien `(0 > x) || (x > 10)`

Extra: Operadores a nivel de bits

- Se pueden utilizar a nivel de bits con números enteros.
- No se pueden usar con datos de otro tipo (p.ej. reales).

Los operadores NOT (~), AND (&), OR(|) y XOR (^)

Si alguno de los operandos es de tipo `long`, el resultado es `long`.
Si no, el resultado es de tipo `int`.

- NOT (~) realiza el complemento a 1 de un número entero: Cambia los 0s por 1s y viceversa

- AND(&), OR(|) y XOR(^) funcionan a nivel de bits como los operadores booleanos AND (&&), OR(||) y XOR (^), respectivamente.

Operación	A nivel de bits	Resultado
~10	~0…00001010 (1…11110101)	-11
10 & 1	0…00001010 & 0…0001 (0…00000000)	0
10 & 2	0…00001010 & 0…0010 (0…00000010)	2
10 & 3	0…00001010 & 0…0011 (0…00000010)	2
10 | 1	0…00001010 | 0…0001 (0…00001011)	11
10 | 2	0…00001010 | 0…0010 (0…00001010)	10
10 | 3	0…00001010 | 0…0011 (0…00001011)	11
10 ^ 1	0…00001010 ^ 0…0001 (0…00001011)	11
10 ^ 2	0…00001010 ^ 0…0010 (0…00001000)	8
10 ^ 3	0…00001010 ^ 0…0011 (0…00001001)	9

Los operadores de desplazamiento <<, >> y >>>

- El operador de desplazamiento a la izquierda (<<) desplaza los bits del primer operando tantas posiciones a la izquierda como indica el segundo operando. Los nuevos bits se rellenan con ceros.

- El operador de desplazamiento a la derecha con signo (>>) desplaza los bits del primer operando tantas posiciones a la derecha como indica el segundo operando. Los nuevos bits se rellenan con unos (si el primer operando es negativo) y con ceros (si es positivo).

- El operador de desplazamiento a la derecha sin signo (>>>) desplaza los bits del primer operando tantas posiciones a la derecha como indica el segundo operando. Los nuevos bits se rellenan siempre con ceros. Se pierde el signo del número.

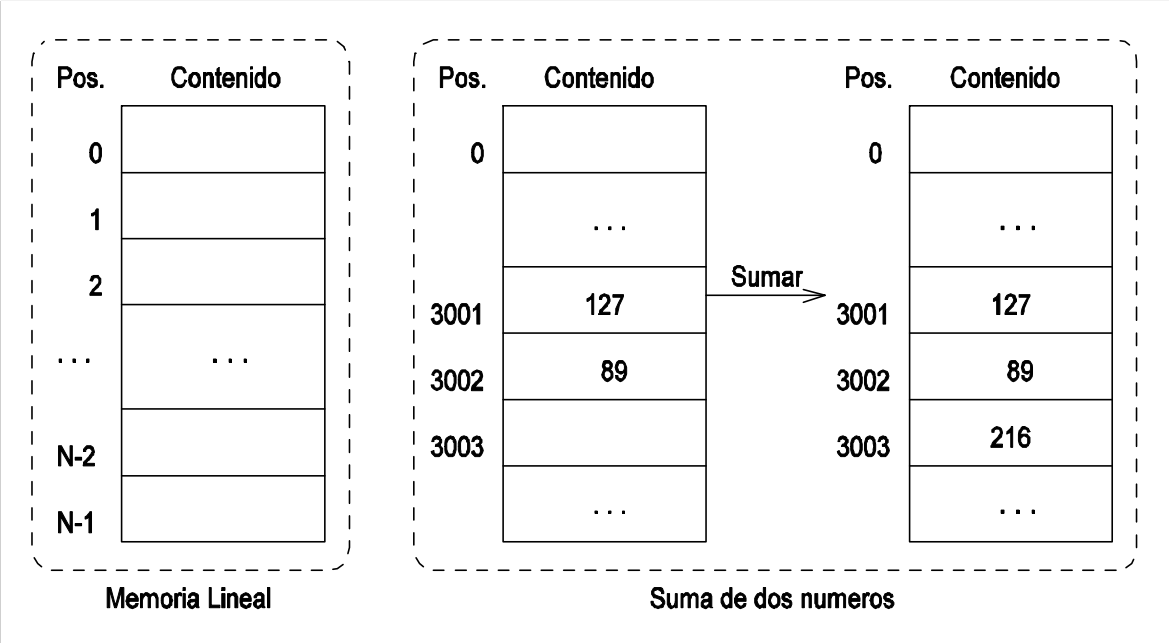
Operación	A nivel de bits	Resultado
10 << 1	`00001010 << 1 (0010100)`	`20 (==10*2)`
7 << 3	`00000111 << 3 (00111000)`	`56 (==7*2^3)`
10 >> 1	`00001010 >> 1 (00000101)`	`5 (==10/2)`
27 >> 3	`00011011 >> 3 (0000011)`	`3 (==27/2^3)`
-50 >> 2	`11001110 >> 2 (11110011)`	`-13 (!=-50/2^2)`
-50 >>> 2	`1…11001110 >>> 2 (001…110011)`	`1073741811`
0xff >>> 4	`11111111 >>> 4 (00001111)`	`15 (==255/2^4)`

`x<<b` es equivalente a multiplicar por 2^b

`x>>b` y `x>>>b` son equivalentes
a realizar una división entera entre 2^b cuando `x` es positivo

Variables

Una variable no es más que un nombre simbólico que identifica una dirección de memoria:



"Suma el contenido de la posición 3001 y la 3002 y lo almacenas en la posición 3003"

vs.

total = cantidad1 + cantidad2

"Suma cantidad1 y cantidad2 y lo almacenas en total"

Declaración de variables

Para usar una variable en un programa hay que declararla.

- El ordenador conoce así cómo codificar la información que se va a almacenar en la posición de memoria correspondiente.
- Al declarar una variable, se reserva el espacio de memoria necesario para almacenar un valor del tipo de la variable.
- El identificador asociado a la variable se puede utilizar para acceder al dato almacenado en memoria y para modificarlo.

Declaración de variables en Java

```
<tipo> identificador;

<tipo> lista de identificadores;
```

- Las variables se han de declarar antes de poder usarlas.
- Los identificadores de las variables son los nombres que utilizaremos para referirnos a ellas.
- Al declarar una variable, hay que definir su tipo: la variable sólo admitirá valores del tipo especificado.
- En una misma declaración se pueden declarar varias variables, siempre que sean del mismo tipo. En este caso, los identificadores de las variables se separan por comas.

Ejemplos

```
// Declaración una variable entera x de tipo int
int x;

// Declaración de una variable real r de tipo double
double r;

// Declaración de una variable c de tipo char
char c;

// Múltiples declaraciones en una sola línea
int i, j, k;
```

Identificadores en Java

- El primer símbolo del identificador será un carácter alfabético (a, …, z, A, …, Z, '_', '$') pero no un dígito. Después de ese primer carácter, podremos poner caracteres alfanuméricos (a, …, z) y (0, 1, …, 9), signos de dólar '$' o guiones de subrayado '_'.

- Los identificadores no pueden coincidir con las palabras reservadas, que ya tienen significado en Java:

```
abstract  continue  for         new       switch
boolean   default   goto        null      synchronized
break     do        if          package   this
byte      double    implements  private   threadsafe
byvalue   else      import      protected throw[s]
case      extends   instanceof  public    transient
catch     false     int         return    true
char      final     interface   short     try
class     finally   long        static    void
const     float     native      super     while

cast      future    generic     inner
operator  outer     rest        var
```

- Las mayúsculas y las minúsculas se consideran diferentes.

- '$' y '_' se interpretan como una letra más.

- '$' no se suele utilizar en la práctica (lo usa el compilador).

Ejemplos válidos

`a, pepe, r456, tu_re_da, AnTeNa, antena, usd$`

Ejemplos no válidos

`345abc, mi variable, Nombre.Largo, cañada, camión`

Java es un lenguaje sensible a mayúsculas/minúsculas.

Convenciones

- Los identificadores deben ser descriptivos: deben hacer referencia al significado de aquello a lo que se refieren.

```
int n1, n2;              // MAL
int anchura, altura;     // BIEN
```

- Los identificadores asociados a las variables se suelen poner en minúsculas.

```
int CoNTaDoR;            // MAL
int contador;            // BIEN
```

- Cuando el identificador está formado por varias palabras, la primera palabra va en minúsculas y el resto de palabras se inician con una letra mayúscula.

```
int mayorvalor;          // MAL
int mayor_valor;         // ACEPTABLE
int mayorValor;          // MEJOR
```

Inicialización de las variables

En una declaración, las variables se pueden inicializar:

```
int i = 0;

float pi = 3.1415927f;

double x = 1.0, y = 1.0;
```

NOTA: La inicialización puede consistir en una expresión compleja que se evalúa cuando se ejecuta el programa.

Definición de constantes en Java

Una constante hace referencia a un valor que no puede modificarse

```
final <tipo> identificador = <valor>;
```

- Las constantes se definen igual que cuando se declara una variable y se inicializa su valor.
- Con la palabra reservada final se impide la modificación del valor almacenado

Convenciones

- Los identificadores asociados a las constantes se suelen poner en mayúsculas.

```
final double PI = 3.141592;
```

- Si el identificador está formado por varias palabras, las distintas palabras se separan con un guión de subrayado

```
final char RETORNO_DE_CARRO = '\n';

final double ELECTRONIC_CHARGE = 1.6E-19;
```

Si intentásemos modificar el valor de una constante, el compilador nos daría un error:

```
RETORNO_DE_CARRO = '\r';      // Ilegal
```

produce el siguiente error

"Cannot assign a value to final variable RETORNO_DE_CARRO"

Expresiones y sentencias

Expresión
Construcción (combinación de tokens)
que se evalúa para devolver un valor.

Sentencia
Representación de una acción o una secuencia de acciones.
En Java, todas las sentencias terminan con un punto y coma [;].

Construcción de expresiones

- Literales y variables son expresiones primarias:

```
1.7      // Literal real de tipo double
sum      // Variable
```

  - Los literales se evalúan a sí mismos.
  - Las variables se evalúan a su valor.

- Los operadores nos permiten combinar expresiones primarias y otras expresiones formadas con operadores:

```
1 + 2 + 3*1.2 + (4 +8)/3.0
```

  Los operadores se caracterizan por:

  - El número de operandos (unarios, binarios o ternarios).
  - El tipo de sus operandos (p.ej. numéricos o booleanos).
  - El tipo del valor que generan como resultado.

Número de operandos

- Operadores unarios

Operador	Descripción
-	Cambio de signo
!	Operador NOT
~	Complemento a 1

- Operadores binarios

Operadores	Descripción
+ - * / %	Operadores aritméticos
== != < > <= >=	Operadores relacionales
&& || ^	Operadores booleanos
& | ^ << >> >>>	Operadores a nivel de bits
+	Concatenación de cadenas

Tipo de los operandos

Operadores	Descripción	Operandos
+ - * / %	Operadores aritméticos	Números
== !=	Operadores de igualdad	Cualesquiera
< > <= >=	Operadores de comparación	Números o caracteres
¡ && || ^	Operadores booleanos	Booleanos
~ & | ^ << >> >>>	Operadores a nivel de bits	Enteros
+	Concatenación de cadenas	Cadenas

Tipo del resultado

Operadores	Descripción	Resultado
+ - * / %	Operadores aritméticos	Número*
== != < > <= >=	Operadores relacionales	Booleano
¡ && || ^	Operadores booleanos	
~ & | ^ << >> >>>	Operadores a nivel de bits	Entero
+	Concatenación de cadenas	Cadena

Sentencias de asignación

Las sentencias de asignación constituyen el ingrediente básico en la construcción de programas con lenguajes imperativos.

Sintaxis:

```
<variable> = <expresión>;
```

Al ejecutar una sentencia de asignación:

1. Se evalúa la expresión que aparece a la derecha del operador de asignación (=).
2. El valor que se obtiene como resultado de evaluar la expresión se almacena en la variable que aparece a la izquierda del operador de asignación (=).

Restricción:

El tipo del valor que se obtiene como resultado de evaluar la expresión ha de ser compatible con el tipo de la variable.

Ejemplos

```
x = x + 1;
int miVariable = 20;          // Declaración con inicialización
otraVariable = miVariable;    // Sentencia de asignación
```

NOTA IMPORTANTE:
Una sentencia de asignación no es una igualdad matemática.

Efectos colaterales

Al evaluar una expresión, algunos operadores provocan efectos colaterales (cambios en el estado del programa; es decir, cambios en el valor de alguna de las variables del programa).

Operadores de incremento (++) *y decremento* (--)

El operador ++ incrementa el valor de una variable.
El operador -- decrementa el valor de una variable.

El resultado obtenido depende de la posición relativa del operando:

Operador	Descripción
`x++`	**Post-incremento** Evalúa primero y después incrementa
`++x`	**Pre-incremento** Primero incrementa y luego evalúa
`x--`	**Post-decremento** Evalúa primero y luego decrementa
`--x`	**Pre-decremento** Primero decrementa y luego evalúa

Ejemplo	Equivalencia	Resultado
`i=0; i++;`	`i=0; i=i+1;`	**`i=1;`**
`i=1; j=++i;`	`i=1; i=i+1; j=i;`	**`j=2;`**
`i=1; j=i++;`	`i=1; j=i; i=i+1;`	**`j=1;`**
`i=3; n=2*(++i);`	`i=3; i=i+1; n=2*i;`	**`n=8;`**
`i=3; n=2*(i++);`	`i=3; n=2*i; i=i+1;`	**`n=6;`**
`i=1; k=++i+i;`	`i=1; i=i+1; k=i+i;`	**`k=4;`**

El uso de operadores de incremento y decremento reduce el tamaño de las expresiones pero las hace más difíciles de interpretar. Es mejor evitar su uso en expresiones que modifican múltiples variables o usan varias veces una misma variable.

*Operadores combinados de asignación (*op=*)*

Java define 11 operadores que combinan el operador de asignación con otros operadores (aritméticos y a nivel de bits):

Operador	Ejemplo	Equivalencia
+=	`i += 1;`	`i++;`
-=	`f -= 4.0;`	`f = f – 4.0;`
***=**	`n *= 2;`	`n = n * 2;`
/=	`n /= 2;`	`n = n / 2;`
%=	`n %= 2;`	`n = n % 2;`
&=	`k &= 0x01;`	`k = k & 0x01;`
|=	`k |= 0x02;`	`k = k & 0x02;`
^=	`k ^= 0x04;`	`k = k & 0x04;`
<<=	`x <<= 1;`	`x = x << 1;`
>>=	`x >>= 2;`	`x = x >> 2;`
>>>=	`x >>>= 3;`	`x = x >>> 3;`

El operador += también se puede utilizar con cadenas de caracteres:

Ejemplo	Resultado
`cadena = “Hola”;` `cadena += “,”`	**`cadena = “Hola,”;`**

`nombre = “Juan”;`
`apellido = “Q.”;`
`nombre += “ ”+apellido;`
	`nombre = “Juan Q.”`

La forma general de los operadores combinados de asignación es

```
variable op= expresión;
```

que pasa a ser

```
variable = variable op (expresión);
```

OJO: `v[i++]+= 2;` y `v[i++]=v[i++]+2;` no son equivalentes.

Conversión de tipos

En determinadas ocasiones, nos interesa convertir el tipo de un dato en otro tipo para poder operar con él.

> *Ejemplo*
>
> Convertir un número entero en un número real
> para poder realizar divisiones en coma flotante

Java permite realizar conversiones entre datos de tipo numérico (enteros y reales), así como trabajar con caracteres como si fuesen números enteros:

- La conversión de un tipo con menos bits a un tipo con más bits es automática (vg. de `int` a `long`, de `float` a `double`), ya que el tipo mayor puede almacenar cualquier valor representable con el tipo menor (además de valores que "no caben" en el tipo menor).

- La conversión de un tipo con más bits a un tipo con menos bits hay que realizarla de forma explícita con "castings". Como se pueden perder datos en la conversión, el compilador nos obliga a ser conscientes de que se está realizando una conversión

```
int i;
byte b;

i = 13;      // No se realiza conversión alguna

b = 13;      // Se permite porque 13 está dentro
             // del rango permitido de valores

b = i;       // No permitido (incluso aunque
             // 13 podría almacenarse en un byte)

b = (byte) i;         // Fuerza la conversión

i = (int) 14.456;     // Almacena 14 en i

i = (int) 14.656;     // Sigue almacenando 14
```

Castings

Para realizar una conversión explícita de tipo (un "casting") basta con poner el nombre del tipo deseado entre paréntesis antes del valor que se desea convertir:

```
char c;
int x;
long k;
double d;
```

Sin conversión de tipo:

```
c = 'A';
x = 100;
k = 100L;
d = 3.0;
```

Conversiones implícitas de tipo (por asignación):

```
x = c;          // char → int
k = 100;        // int → long
k = x;          // int → long
d = 3;          // int → double
```

Conversiones implícitas de tipos (por promoción aritmética)

```
c+1             // char → int
x / 2.0f        // int → float
```

Errores de conversión (detectados por el compilador):

```
x = k;          // long → int
x = 3.0;        // double → int
x = 5 / 2.0f;   // float → int
```

Conversiones explícitas de tipo (castings):

```
x = (int) k;
x = (int) 3.0;
x = (int) (5 / 2.0f);
```

Tabla de conversión de tipos

		Convertir a							
		`boolean`	`byte`	`short`	`char`	`int`	`long`	`Float`	`double`
Convertir desde	`boolean`	-	N	N	N	N	N	N	N
	`byte`	N	-	I	C	I	I	I	I
	`Short`	N	C	-	C	I	I	I	I
	`char`	N	C	C	-	I	I	I	I
	`int`	N	C	C	C	-	I	I*	I
	`long`	N	C	C	C	C	-	I*	I*
	`float`	N	C	C	C	C	C	-	I
	`double`	N	C	C	C	C	C	C	-

N No se puede realizar la conversión
(boolean no se puede convertir a otro tipo)

I Conversión implícita
(se realiza de forma automática)

I* Conversión implícita
(con posible pérdida de dígitos significativos)

C Conversión explícita
(requiere la realización de un casting)

El compilador de Java comprueba siempre los tipos de las expresiones y nos avisa de posibles errores:
"Incompatible types" (N) y *"Possible loss of precision"* (C)

Algunas conversiones de interés

De números en coma flotante a números enteros

Al convertir de número en coma flotante a entero,
el número se trunca (redondeo a cero).

En la clase `Math` existen funciones
que nos permiten realizar el redondeo de otras formas:

`Math.round(x), Math.floor(x), Math.ceil(x)`

Conversión entre caracteres y números enteros

Como cada carácter tiene asociado un código UNICODE,
los caracteres pueden interpretarse como números enteros sin signo

```
int i;
char c;
```

Ejemplo	Equivalencia	Resultado
`I = 'a';`	`i = (int) 'a';`	**`i = 97`**
`c = 97;`	`c = (char) 97;`	**`c = 'a'`**
`c = 'a'+1;`	`c = (char) ((int)'a'+1);`	**`c = 'b'`**
`C = (char)(i+2);`		**`c = 'c'`**

Conversión de cadenas de texto en datos del tipo adecuado

Cuando leemos un dato desde el teclado, obtenemos una cadena de texto (un objeto de tipo `String`) que deberemos convertir en un dato del tipo adecuado utilizando las siguientes funciones auxiliares:

```
b = Byte.parseByte(str);      // String → byte
s = Short.parseShort(str);    // String → short
i = Integer.parseInt(str);    // String → int
n = Long.parseLong(str);      // String → long

f = Float.parseFloat(str);    // String → float
d = Double.parseDouble(str);  // String → double
```

Evaluación de expresiones

- La **precedencia** de los operadores determina el orden de evaluación de una expresión (el orden en que se realizan las operaciones):

  3*4+2 es equivalente a (3*4)+2
  porque el operador * es de mayor precedencia que el operador +

- Cuando en una expresión aparecen dos operadores con el mismo nivel de precedencia, la **asociatividad** de los operadores determina el orden de evaluación.

  a – b + c – d es equivalente a ((a – b) + c) – d
  porque *los operadores aritméticos*
  son asociativos de izquierda a derecha

  a = b += c = 5 es equivalente a a = (b += (c = 5))
  porque *los operadores de asignación*
  son asociativos de derecha a izquierda

- La precedencia y la asociatividad determinan el orden de los operadores, pero no especifican el orden en que se evalúan los **operandos** de un operador binario (un operador con dos operandos):

  En Java, los operandos se evalúan de izquierda a derecha:
  el operando de la izquierda se evalúa primero.

  Si los operandos no tienen efectos colaterales (esto es, no cambian el valor de una variable), el orden de evaluación de los operandos es irrelevante. Sin embargo, las asignaciones `n=x+(++x);` y `n=(++x)+x;` generan resultados diferentes.

NOTA: **Siempre es recomendable el uso de paréntesis.**
Los paréntesis nos permiten especificar el orden de evaluación de una expresión, además de hacer su interpretación más fácil.

Programas

Estructura de un programa simple

Los programas más simples escritos en lenguajes imperativos suelen realizar tres tareas de forma secuencial:

- Entrada de datos
- Procesamiento de los datos
- Salida de resultados

El punto de entrada de un programa en Java es la función `main`:

```
public static void main (String[] args)
{
    Declaraciones y sentencias escritas en Java
}
```

En realidad, Java es un lenguaje de programación orientada a objetos y todo debe estar dentro de una clase, incluida la función `main`, tal como muestra el siguiente programa

```
public class MiPrimerPrograma
{
  public static void main (String args[])
  {
    System.out.println("Mensaje por pantalla”);
  }
}
```

- Las llaves `{}` delimitan bloques en Java (conjuntos de elementos de un programa).

- La máquina virtual Java ejecuta el programa invocando a la función `main`.

Operaciones básicas de entrada/salida

Mostrar resultados con la función System.out.println

La función `System.out.println` nos permite mostrar una línea de texto en la pantalla cuando ejecutamos el programa:

```
int edad = 25;
System.out.println("Tengo " + edad + " años");

final double pi = 3.1415927;
System.out.println("Valor de PI = " + pi);
```

- En la función `println` se suele utilizar el operador + para concatenar cadenas de caracteres.
- Cualquier cosa en Java se puede convertir en una cadena.

Obtener datos de entrada a través de los parámetros de main

La función `main` puede recibir parámetros:

```
public class Eco
{
  public static void main (String[] args)
  {
    System.out.println(args[0]);
    System.out.println(args[1]);
  }
}
```

Si ejecutamos el programa `Eco`

```
java Eco dato1 dato2
```

obtenemos como salida

```
dato1
dato2
```

- Los parámetros de la función main son de tipo `String`, por lo que deberemos convertir las cadenas en datos del tipo deseado.
- Con `args.length` podemos saber cuántos parámetros se le han pasado a nuestro programa.

Leer datos desde el teclado en Java

En Java, la realización de operaciones de entrada de datos no es inmediata, por lo que utilizaremos una clase auxiliar que se encargará de realizar las operaciones necesarias:

```
b = TextIO.getByte();    // byte
s = TextIO.getShort();   // short
i = TextIO.getInt();     // int
k = TextIO.getLong();    // long

x = TextIO.getFloat();   // float
y = TextIO.getDouble();  // double

a = TextIO.getBoolean(); // boolean

c = TextIO.getChar();    // char

s = TextIO.getln();      // String
```

Internamente, la implementación de la clase auxiliar `TextIO` realiza algo similar a lo siguiente:

```
InputStreamReader input;
BufferedReader lector;
String cadena;

// Secuencia de bytes → Secuencia de caracteres
input = new InputStreamReader(System.in);
// Secuencia de caracteres → Secuencia de líneas
lector = new BufferedReader(input);

try {
    cadena = lector.readLine();
} catch (Exception e) {
    cadena = "";
}

// Y ahora hacemos lo que queramos con la cadena
```

Cuando estudiemos el uso de ficheros en Java comprenderemos exactamente qué es lo que se hace al leer datos desde el teclado.

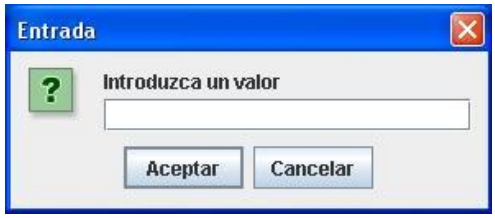
E/S con interfaces gráficas de usuario (GUIs): JOptionPane

La biblioteca de clases estándar de Java incluye una amplia gama de componentes para la construcción de interfaces gráficas de usuario.

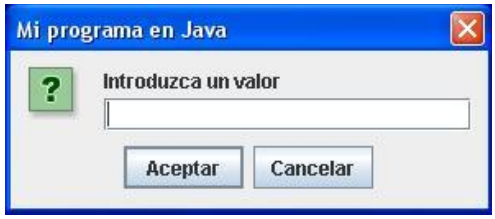
El componente `javax.swing.JOptionPane` se puede emplear para obtener datos de entrada y mostrar mensajes de salida:

Entrada de datos con showInputDialog

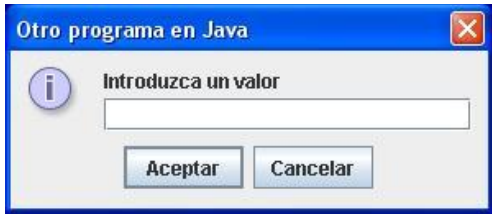
```
String entrada;
entrada = JOptionPane.showInputDialog
          ( "Introduzca un valor" );
```




```
entrada = JOptionPane.showInputDialog ( null,
            "Introduzca un valor",
            "Mi programa en Java",
            JOptionPane.QUESTION_MESSAGE );
```




```
entrada = JOptionPane.showInputDialog ( null,
            "Introduzca un valor",
            "Otro programa en Java",
            JOptionPane.INFORMATION_MESSAGE );
```



Salida de datos con showMessageDialog

```
JOptionPane.showMessageDialog ( null,
            “¡Bienvenido a Java!” );
```


```
JOptionPane.showMessageDialog ( null,
            “¡Bienvenido a Java!”,
            “Programa de ejemplo”,
            JOptionPane.INFORMATION_MESSAGE );
```


```
JOptionPane.showMessageDialog ( null,
            “¡Bienvenido a Java!”,
            “Programa de ejemplo”,
            JOptionPane.WARNING_MESSAGE );
```


```
JOptionPane.showMessageDialog ( null,
            “¡Bienvenido a Java!”,
            “Programa de ejemplo”,
            JOptionPane.ERROR_MESSAGE );
```

Cuando se emplea `JOptionPane` es aconsejable llamar a `System.exit(0);` para terminar la ejecución del programa.

Ejemplos

Calificación final de la asignatura

```
public class Asignatura
{
  public static void main (String args[])
  {
    // Declaraciones
    // Constante
    final double PORCENTAJE_LABORATORIO = 0.5;
    // Variables
    String nombre;
    double notaExamen;
    double notaLaboratorio;
    double notaFinal;
    // Entrada de datos
    nombre = args[0];
    notaExamen = Double.parseDouble(args[1]);
    notaLaboratorio = Double.parseDouble(args[2]);
    // Cálculo
    notaFinal =
        (1-PORCENTAJE_LABORATORIO)*notaExamen
        + PORCENTAJE_LABORATORIO*notaLaboratorio;
    // Salida de resultados
    System.out.println (nombre
        + " ha obtenido una nota final de "
        + notaFinal);
  }
}
```

Ejecución del programa

```
java Asignatura Pepe 6 9
```

```
Pepe ha obtenido una nota final de 7.5
```

```
java Asignatura “Juan Nadie” 8 9.5
```

```
Juan Nadie ha obtenido una nota final de 8.75
```

Ejemplos

Año bisiesto

Programa para comprobar si un año es bisiesto o no:
Un año es bisiesto si es divisible por 4 pero no por 100,
o bien es divisible por 400.

```
public class Bisiesto
{
  public static void main (String args[])
  {
    // Declaración de variables

    int     year;
    boolean bisiesto;

    // Entrada de datos

    year = Integer.parseInt(args[0]);

    // Cálculos

    bisiesto = ((year%4==0) && (year%100!=0))
             || (year%400==0);

    // Salida de resultados

    System.out.println(bisiesto);
  }
}
```

Ejecución del programa

```
java Bisiesto 2004
    true
java Bisiesto 2005
    false
java Bisiesto 2000
    true
java Bisiesto 2100
    false
```

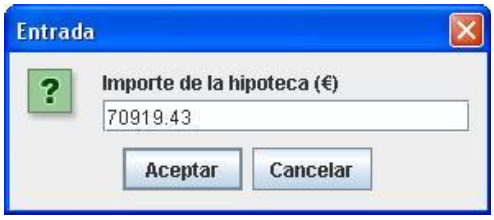
Ejemplos

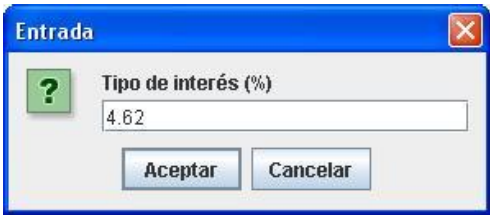
Cuota de una hipoteca

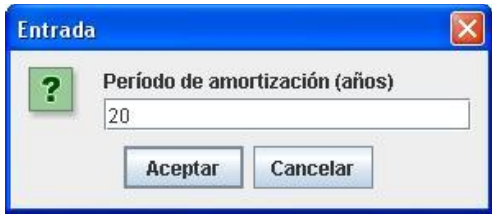
```
import javax.swing.JOptionPane;

public class Hipoteca
{
  public static void main (String args[])
  {
    double cantidad; // en euros
    double interes;  // en porcentaje (anual)
    int    tiempo;   // en años
    double cuota;    // en euros (mensual)
    double interesMensual; // en tanto por uno
    String entrada;        // variable auxiliar

    // Entrada de datos

    entrada = JOptionPane.showInputDialog
                ("Importe de la hipoteca (€)");
    cantidad = Double.parseDouble(entrada);

    entrada = JOptionPane.showInputDialog
                ("Tipo de interés (%)");
    interes = Double.parseDouble(entrada);

    entrada = JOptionPane.showInputDialog
                ("Período de amortización (años)");
    tiempo = Integer.parseInt(entrada);

    // Cálculo de la cuota mensual

    interesMensual = interes/(12*100);

    cuota = (cantidad*interesMensual)
      / (1.0–1.0/Math.pow(1+interesMensual,tiempo*12));

    cuota = Math.round(cuota*100)/100.0;

    // Resultado

    JOptionPane.showMessageDialog
       (null, "Cuota mensual = "+cuota+"€" );

    System.exit(0);
  }
}
```

Ejecución del programa

1. Datos de entrada







2. Cálculo de la cuota mensual

$$interésMensual\ (\delta) = interésAnual\ /\ 12$$

$$cuotaMensual = \frac{cantidad \times \delta}{1 - \frac{1}{(1+\delta)^{años \times 12}}}$$

Redondeo

Operación	Descripción	Resultado
`100*cuota`	Cuota en céntimos de euro	`double`
`Math.round(…)`	Redondeo al entero más cercano	`int`
`… / 100.0`	Cuota en euros (con céntimos)	`double`

3. Resultado final

Comentarios

Los comentarios sirven para incluir aclaraciones en el código.

Java permite dos tipos de comentarios:

```
// Comentarios de una línea
/* Comentarios de varias líneas */
```

- Es bueno incluir comentarios que expliquen lo que hace el programa y sus características claves (p.ej. autor, fecha, algoritmos utilizados, estructuras de datos, peculiaridades…).

```
// Cálculo del MCD
// usando el algoritmo de Euclides
// © Fernando Berzal, 2004
```

- Los comentarios nunca han de limitarse a decir en lenguaje natural lo que ya está escrito en el código: Jamás se utilizarán para "parafrasear" el código y repetir lo que es obvio.

```
✗ i++;       // Incrementa el contador
```

- Los comentarios han de aclarar; esto es, ayudar al lector en las partes difíciles (y no confundirle). Si es posible, escriba código fácil de entender por sí mismo: cuanto mejor lo haga, menos comentarios necesitará.

```
✗ int mes;   // Mes
✓ int mes;   // Mes del año (1..12)
```

```
✗ ax = 0x723;   /* RIP L.v.B. */
```

NB: Beethoven murió en 1827 (0x723 en hexadecimal).

Sangrías

Conviene utilizar espacios en blanco o separadores para delimitar el ámbito de las estructuras de control de nuestros programas.

Líneas en blanco

Para delimitar claramente los distintos segmentos de código en nuestros programas dejaremos líneas en blanco entre ellos.

Identificadores

Los identificadores deben ser descriptivos (reflejar su significado).

- ✗ `p, i, s...`
- ✓ `precio, izquierda, suma...`

Declaraciones

- Usualmente, declararemos una única variable por línea.
- Nunca mezclaremos en una misma línea la declaración de variables que sean de distintos tipos o que se utilicen en el programa para distintos fines.

Constantes

- Se considera una mala costumbre incluir literales de tipo numérico (“números mágicos”) en medio del código. Se prefiere la definición de constantes simbólicas (declaraciones con `final`).

Expresiones

- Uso de paréntesis: Aunque las normas de precedencia de los operadores vienen definidas en el lenguaje, no abusaremos de ellas. Siempre resulta más fácil interpretar una expresión si ésta tiene los paréntesis apropiados. Además, éstos eliminan cualquier tipo de ambigüedad.

- Uso de espacios en blanco: Resulta más fácil leer una expresión con espacios que separen los distintos operadores y operandos involucrados en la expresión.

```
a%x*c/b-1    →   ( (a%x) * c ) / b - 1
```

- Expresiones booleanas: Es aconsejable escribirlas como se dirían en voz alta.

```
!(bloque<actual)   →   ( bloque >= actual )
```

- Expresiones complejas:
  Es aconsejable dividirlas para mejorar su legibilidad

- Claridad:
  Siempre buscaremos la forma más simple de escribir una expresión.

```
✗ key = key >> ( bits – ((bits>>3)<<3));

✓ key >>= bits & 0x7;
```

- Conversiones de tipo (castings):
  Evitaremos las conversiones implícitas de tipo. Cuando queramos realizar una conversión de tipo, lo indicaremos explícitamente.

```
i = (int) f;
```

- Siempre se han de evitar los efectos colaterales (modificaciones no deseadas pueden afectar a la ejecución del programa sin que nos demos cuenta de ello).

IDEA CLAVE

Escribimos código para que lo puedan leer otras personas, no sólo para que lo traduzca el compilador (si no fuese así, podríamos seguir escribiendo nuestros programas en binario).

- No comente el código "malo" (uso de construcciones extrañas, expresiones confusas, sentencias poco legibles...): Reescríbalo.

- No contradiga al código: Los comentarios suelen coincidir con el código cuando se escriben, pero a medida que se corrigen errores y el programa evoluciona, los comentarios suelen dejarse en su forma original y aparecen discrepancias. Si cambia el código, asegúrese de que los comentarios sigan siendo correctos.

El código bien escrito
es más fácil de leer, entender y mantener
(además, seguramente tiene menos errores)

Errores de programación

Errores sintácticos

Errores detectados por el compilador en tiempo de compilación.

Errores semánticos

Sólo se detectan en tiempo de ejecución: Causan que el programa finalice inesperadamente su ejecución (p.ej. división por cero) o que el programa proporcione resultados incorrectos.

Elementos léxicos del lenguaje de programación

Java

Elementos léxicos del lenguaje de programación Java

- Palabras reservadas
- Identificadores
- Literales
- Operadores
- Delimitadores
- Comentarios

Apéndices

- Operadores de Java
- Sintaxis de Java

Elementos léxicos de Java

Token

Componente léxico de un lenguaje de programación

Palabras reservadas

Palabras que tiene un significado concreto en el lenguaje de programación, sin necesidad de que se lo asignemos nosotros.

```
abstract  continue  for         new       switch
boolean   default   goto        null      synchronized
break     do        if          package   this
byte      double    implements  private   threadsafe
byvalue   else      import      protected throw[s]
case      extends   instanceof  public    transient
catch     false     int         return    true
char      final     interface   short     try
class     finally   long        static    void
const     float     native      super     while

cast      future    generic     inner
operator  outer     rest        var
```

Identificadores

Palabras que podemos utilizar para denominar algo en el lenguaje.

Identificadores en Java

- El primer símbolo del identificador será un carácter alfabético (a, …, z, A, …, Z, '_', '$') pero no un dígito. Después de ese primer carácter, podremos poner caracteres alfanuméricos (a, …, z) y (0, 1, …, 9), signos de dólar '$' o guiones de subrayado '_'.
- Los identificadores no pueden coincidir con las palabras reservadas.
- Las mayúsculas y las minúsculas se consideran diferentes.
- El signo de dólar y el guión de subrayado se interpretan como una letra más.

  Ejemplos válidos

  a, pepe, r456, tu_re_da, AnTeNa, antena, usd$

  Ejemplos no válidos

  345abc, mi variable, Nombre.Largo, cañada, camión

Literal

Especificación de un valor concreto de un tipo de dato.

Números enteros
21 (int), 21L (long), 077 (en octal), 0xDC00 (en hexadecimal)

Números reales
3.14 (double), 3.14f (float), 3.14d (double), 2e12, 3.1E12

Valores booleanos
true (verdadero), false (falso)

Caracteres
'p', '\u????' (código UNICODE en hexadecimal), '\t' (tabulador)…

Cadenas de caracteres
"mensaje", "" (cadena vacía)

Operadores

Igual que en Matemáticas, realizan una acción específica:

- Suelen estar definidos en el núcleo del compilador
  (aunque también pueden estar definidos en bibliotecas externas)
- Suelen representarse con tokens formados por símbolos.
- Suelen utilizar notación infija.
- Pueden aplicarse a uno o varios operandos (argumentos).
- Suelen devolver un valor.

Operadores en Java, por orden de precedencia

```
. [ ] ( )
++ --
! ~ instanceof
new
* / %
+ -
<< >> >>>
< > <= >= == !=
& ^ |
&& ||
? :
= op= ,
```

Delimitadores

Símbolos utilizados como separadores de las distintas construcciones de un lenguaje de programación (esto es, los signos de puntuación de un lenguaje de programación.

`( )` PARÉNTESIS: Listas de parámetros en la definición y llamada a métodos, precedencia en expresiones, expresiones para control de flujo y conversiones de tipo.

`{ }` LLAVES: Inicialización de arrays, bloques de código, clases, métodos y ámbitos locales.

`[ ]` CORCHETES: Arrays.

`;` PUNTO Y COMA: Separador de sentencias.

`,` COMA: Identificadores consecutivos en una declaración de variables y sentencias encadenadas dentro de una sentencia for.

`.` PUNTO: Separador de nombres de paquetes, subpaquetes y clases; separador entre variables y métodos/miembros.

Comentarios

Aclaración que el programador incluye en el texto del programa para mejorar su inteligibilidad.

En Java hay tres tipos de comentarios:

```
// Comentario de una sola línea

/* Comentario de una o más líneas */

/** Comentario de documentación, una o más líneas */
```

> La herramienta `javadoc` genera automáticamente en HTML la documentación del código a partir de los comentarios `/** … */`

Apéndices

P	A	Operador	Operando(s)	Operación
15	I	. [ ] (args) ++, - -	Objeto, método (miembro) Array (indice) Método, lista de argumentos variable	Acceso a un miembro del objeto Acceso a un elemento de un array Llamada a un método Post incremento, post decremento
14	D	++,- - +,- ~ !	Variable Número Entero Booleano	Pre incremento, Pre decremento Cambio de signo (-) Complemento a nivel de bit NOT booleano
13	D	new (type)	Clase, lista de argumentos Tipo, cualquier tipo	Creación de objetos Cast (conversión de tipos)
12	I	* ,/, %	Número, número	Multplicación, división, módulo. Válido tb para fp
11	I	+,- +	Numero, número String, cualquiera	Suma, resta Concatenación de cadenas
10	I	<< >> >>>	Entero, entero Entero, entero Entero, entero	Desplazamiento a izquierda Desplazamiento a derecha con signo Desplazamiento a derecha con ceros
9	I	<, <= >, >= instance of	Número, Número Número, Número Referencia, tipo	Menor que, menor igual que Mayor que, mayor igual que Comparación de tipo
8	I	== != == !=	Primitiva , primitiva Primitiva , primitiva Referencia, referencia Referencia, referencia	Igual (tiene el mismo valor) No igual (diferente valor) Igual (mismo objeto) No igual (diferente objeto)
7	I	& &	Entero, entero Booleano, booleano	And booleano a nivel de bits And Booleano
6	I	^ ^	Entero, Entero Booleano, Booleano	XOR booleano a nivel de bits XOR Booleano
5	I	| |	Entero, Entero Booleano, Booleano	OR booleano a nivel de bits OR Booleano
4	I	&&	Booleano, Booleano	AND Condicional
3	I	||	Booleano, Booleano	OR Condicional
2	D	?:	Booleano, otro, otro	Operador condicional (if)
1	D	= *=,/=,%= +=,-= ,<<=,>>= >>>=, &=,^=,|=	Variable, otro	Asignación con operación

P: Precedencia
A : Asociatividad (I=Izquierda, D=Derecha)

Elemento	Objetivo	Sintaxis
Asignación	Evaluación de una expresión y almacenamiento del valor obtenido como resultado	var **=** expr**;** expr**++;**
Llamada	Llamada a un método	method**();**
Instanciación	Creación de un objeto	**new** Type**( )**
Secuencia	Grupo de instrucciones	**{** instrucciones **}**
Vacía	No hacer nada	**;**
Etiqueta	Etiquetado de una instrucción	etiqueta**:** instrucción
Variable	Declaración de una variable	[final] tipo nombre [=valor] [, nombre [=valor]]…;
if	Condicional	**if** (expr) instrucción [**else** instrucción]

switch	Condicional	**switch** (expr) {
		[**case** expr **:** instrucciones]…
		[**default :** instrucciones]
		}

while	Bucle	**while** (expr) instrucción
do	Bucle	**do** instrucción **while** (expr);
for	Bucle	**for** (init; test; increment) instrucción
break	Salir de un bloque	**break** [etiqueta] ;
continue	Reiniciar un bucle	**continue** [etiqueta];
return	Resultado de un método	**return** [expr];
synchronized	Sección crítica	**synchronized** (expr) {instrucciones}
throw	Lanzamiento de excepciones	**throw** expr;
try	Manejo de excepciones	**try** {instrucciones} [**catch** (tipo) {instrucciones}]… [**finally** {instrucciones}]

Introducción a la programación
Relación de ejercicios

Conceptos básicos

1. Escriba (en lenguaje natural) un algoritmo adecuado para la elaboración de su receta de cocina favorita. Analice las características de su algoritmo (especialmente, su precisión).

2. Elabore, en lenguaje natural, dos algoritmos que permitan calcular el máximo común divisor de dos números enteros positivos. Analice las características de los algoritmos propuestos (precisión, finitud y eficiencia).

   NOTA: Busque información sobre el algoritmo de Euclides.

Datos, tipos de datos y expresiones

3. Traduzca las siguientes fórmulas a expresiones escritas en Java, declarando para ello las variables que considere necesarias:

   a. $F = \frac{9}{5}C + 32$

   b. $f(x,y) = \frac{1+\frac{x^2}{y}}{\frac{x^3}{1+y}}$

   c. $\sqrt{1+\left(\frac{e^x}{x^2}\right)^2}$

4. ¿Cuál es el resultado de evaluar las siguientes expresiones si suponemos que, inicialmente, `x` vale `1`?

   ```
   a. (x > 1) & (x++ < 10)

   b. (1 > x) && ( 1 > x++)

   c. (1 == x) | (10 > x++)

   d. (1 == x) || (10 > x++)

   e. (++x) + x;

   f. x + (++x)
   ```

Programas

5. Diseñe un programa que lea los coeficientes de una ecuación de segundo grado $ax^2+bx+c=0$ y calcule sus dos soluciones. Se supone que la ecuación tiene soluciones reales.

6. Diseñe un programa que lea los coeficientes de un sistema de dos ecuaciones lineales con dos incógnitas y calcule su solución. Se supone que el sistema de ecuaciones es compatible determinado.

$$\left.\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}\right\}$$

7. Implemente un programa que, dados los tres vértices de un triángulo, calcule el área del mismo. Puede aplicar la siguiente fórmula:

$$S = \sqrt{T(T - S_1)(T - S_2)(T - S_3)}$$

donde S_1, S_2 y S_3 son las longitudes de los tres lados del triángulo y T es la mitad de su perímetro.

8. Dada una medida de tiempo expresada en horas, minutos y segundos con valores arbitrarios, elabore un programa que transforme dicha medida en una expresión correcta. Por ejemplo, dada la medida *3h 118m 195s,* el programa deberá obtener como resultado *5h 1m 15s.*

9. Escriba un programa en Java que nos calcule el cambio que debe dar la caja de un supermercado: Dado un precio y una cantidad de dinero, el programa nos dirá cuántas monedas deben darse como cambio de tal forma que el número total de monedas sea mínimo.

10. El precio final de un producto para un comprador es la suma total del costo del producto, un porcentaje de beneficios que obtiene el vendedor y el I.V.A. Diseñar un algoritmo para obtener el precio final de un producto sabiendo su costo, el porcentaje de beneficios y el I.V.A. aplicable. Obtener el resultado redondeando a los cinco céntimos (p.ej. 5.94€ $\rightarrow$ 5.95€).

11. Un banco recibe todos los días del Banco Mundial una lista de cómo está el cambio de las divisas del mundo respecto del dólar americano (USD). Diseñar un algoritmo que, a partir de una cantidad de dólares que deseamos comprar, nos devuelva la cantidad en euros (y en pesetas) que nos costarían esos dólares. Suponga que el banco obtiene en el cambio un tanto por ciento variable de beneficios.

NOTA: 1 euro = 166.386 pesetas

Introducción a la programación
Relación de ejercicios

Datos, tipos de datos y expresiones

1. Traduzca las siguientes fórmulas a expresiones escritas en Java, declarando para ello las variables que considere necesarias:

   a. $F = \frac{9}{5}C + 32$

   ```
   int c, f;
   ```
   Solución **`f = 9*c/5 + 32;`**
   Error común `f = 9/5*c + 32; // == 1*c + 32`

   ```
   float c, f;
   ```
   Solución **`f = 9*c/5 + 32;`**
   Error común `f = (9/5)*c + 32;  // == 1*c + 32`
   Solución alternativa **`f = (9.0/5.0)*c + 32;`**

   b. $f(x,y) = \dfrac{1 + \frac{x^2}{y}}{\frac{x^3}{1+y}}$

   ```
   double x,y;          // Tanto x como y
   double f;            // han de ser != 0

   f = ( 1 + x*x/y ) / ( x*x*x / (1+y) );

   // ERROR: Tal como está escrita la expresión,
   //        y debería tener un valor distinto de -1

   f = ( 1 + x*x/y ) * (1+y) / ( x*x*x );
   ```

   c. $\sqrt{1 + \left(\frac{e^x}{x^2}\right)^2}$

   ```
   double x;

   Math.sqrt ( 1 + Math.pow( Math.exp(x)/(x*x), 2) )

   Math.sqrt ( 1 + Math.exp(2*x) / Math.pow(x,4) )

   Math.sqrt ( 1 + Math.exp(2*x) / (x*x*x*x) )
   ```

2. ¿Cuál es el resultado de evaluar las siguientes expresiones si suponemos que, inicialmente, `x` vale `1`?

a. `(x > 1) & (x++ < 10)`

false

```
(1 > 1) & (1 < 10) == false & true == false
x = 2;
```

b. `(1 > x) && ( 1 > x++)`

false

```
(1 > 1) && (1 > 1) == (1 > 1) == false
x = 2;
```

c. `(1 == x) | (10 > x++)`

true

```
(1 == 1) | (10 > 1) == true | true == true
x = 2;
```

d. `(1 == x) || (10 > x++)`

true

```
(1 == 1) || (10 > 1) == true || true == true
x = 2;
```

e. `(++x) + x;`

4

```
x = 1
(++x) + x
x = 2;
x + x == 2 + 2 == 4
```

f. `x + (++x)`

3

```
x = 1;
1 + (++x)
x = 2;
1 + x == 1 + 2 == 3
```

Estructuras de control

Programación estructurada

Estructuras condicionales

- La sentencia `if`
- La cláusula `else`
- Encadenamiento y anidamiento
- El operador condicional `?:`
- La sentencia `switch`

Estructuras repetitivas/iterativas

- El bucle `while`
- El bucle `for`
- El bucle `do...while`
- Bucles anidados

Cuestiones de estilo

Las estructuras de control
controlan la ejecución de las instrucciones de un programa
(especifican el orden en el que se realizan las acciones)

Programación estructurada

IDEA CENTRAL:
Las estructuras de control de un programa
sólo deben tener **un punto de entrada y un punto de salida**.

La programación estructurada...

mejora la productividad de los programadores.

mejora la legibilidad del código resultante.

La ejecución de un programa estructurado progresa **disciplinadamente**, en vez de saltar de un sitio a otro de forma impredecible

Gracias a ello, los programas…

resultan más fáciles de probar

se pueden depurar más fácilmente

se pueden modificar con mayor comodidad

En programación estructurada sólo se emplean tres construcciones:

Secuencia
Conjunto de sentencias que se ejecutan en orden

Ejemplos:
Sentencias de asignación y llamadas a rutinas.

Selección
Elige qué sentencias se ejecutan en función de una condición.

Ejemplos:
Estructuras de control condicional `if-then-else` y `case/switch`

Iteración
Las estructuras de control repetitivas repiten conjuntos de instrucciones.

Ejemplos:
Bucles `while`, `do...while` y `for`.

Teorema de Böhm y Jacopini (1966):
Cualquier programa de ordenador
puede diseñarse e implementarse
utilizando únicamente las tres construcciones estructuradas
(secuencia, selección e iteración; esto es, sin sentencias `goto`).

Böhm, C. & Jacopini, G.: "Flow diagrams, Turing machines, and languages only with two formation rules". Communications of the ACM, 1966, Vol. 9, No. 5, pp. 366-371

Dijkstra, E.W.: "Goto statement considered harmful". Communications of the ACM, 1968, Vol. 11, No. 3, pp. 147-148

Estructuras de control condicionales

Por defecto,
las instrucciones de un programa se ejecutan secuencialmente:

> El orden secuencial de ejecución no altera el flujo de control del programa respecto al orden de escritura de las instrucciones.

Sin embargo, al describir la resolución de un problema, es normal que tengamos que tener en cuenta condiciones que influyen sobre la secuencia de pasos que hay que dar para resolver el problema:

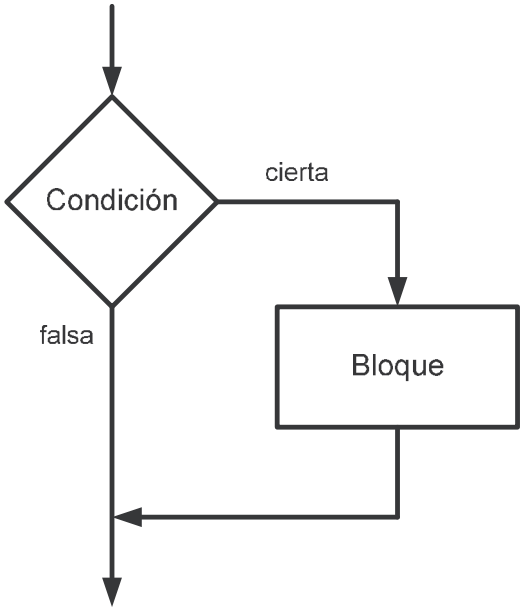
> Según se cumplan o no determinadas condiciones,
> la secuencia de pasos involucrada en la realización de una tarea será diferente

Las estructuras de control condicionales o selectivas nos permiten decidir qué ejecutar y qué no en un programa.

Ejemplo típico
Realizar una división sólo si el divisor es distinto de cero.

La estructura de control condicional `if`

La sentencia `if` nos permite elegir
si se ejecuta o no un bloque de instrucciones.



Sintaxis

```
if (condición)
   sentencia;


if (condición) {
   bloque
}
```

donde `bloque` representa un bloque de instrucciones.

Bloque de instrucciones:
Secuencia de instrucciones encerradas entre dos llaves {....}

Consideraciones acerca del uso de la sentencia `if`

- Olvidar los paréntesis al poner la condición del `if` es un error sintáctico (los paréntesis son necesarios)

- No hay que confundir el operador de comparación `==` con el operador de asignación `=`

- Los operadores de comparación `==`, `!=`, `<=` y `>=` han de escribirse sin espacios.

- `=>` y `=<` no son operadores válidos en Java.

- El fragmento de código afectado por la condición del `if` debe sangrarse para que visualmente se interprete correctamente el ámbito de la sentencia `if`:

```
if (condición) {
   // Aquí se incluye el código
   // que ha de ejecutarse
   // cuando se cumple la condición del if
}
```

- Aunque el uso de llaves no sea obligatorio cuando el `if` sólo afecta a una sentencia, es recomendable ponerlas siempre para delimitar explícitamente el ámbito de la sentencia `if`.

Error común:

```
if (condición);
   sentencia;
```

es interpretado como

```
if (condición)
   ;   // Sentencia vacía
sentencia;
```

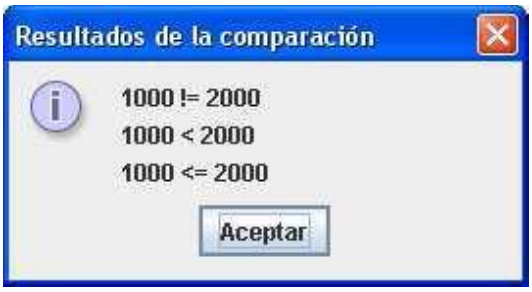
¡¡¡La sentencia siempre se ejecutaría!!!

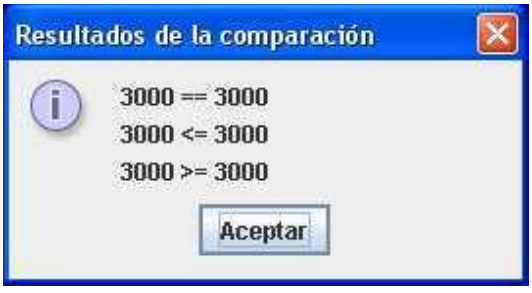
Ejemplo
Comparación de números (Deitel & Deitel)

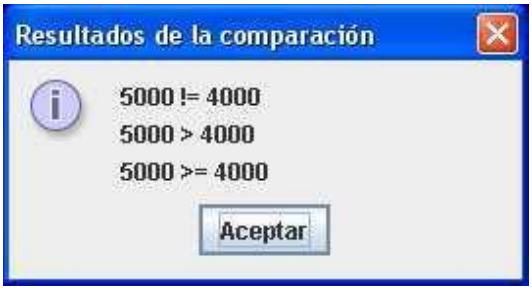
Operador	Significado
==	Igual
!=	Distinto
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual

```
import javax.swing.JOptionPane;

public class Comparison
{
  public static void main( String args[] )
  {
    // Declaración de variables

    String primerDato, segundoDato;
    String resultado;
    int    dato1, dato2;


    primerDato  = JOptionPane.showInputDialog
                  ( "Primer dato:" );
    segundoDato = JOptionPane.showInputDialog
                  ( "Segundo dato:" );

    dato1 = Integer.parseInt( primerDato );
    dato2 = Integer.parseInt( segundoDato );

    resultado = "";

    if ( dato1 == dato2 )
       resultado += dato1 + " == " + dato2;

    if ( dato1 != dato2 )
       resultado += dato1 + " != " + dato2;
```

```
      if ( dato1 < dato2 )
         resultado += "\n" + dato1 + " < " + dato2;

      if ( dato1 > dato2 )
         resultado += "\n" + dato1 + " > " + dato2;

      if ( dato1 <= dato2 )
         resultado += "\n" + dato1 + " <= " + dato2;

      if ( dato1 >= dato2 )
         resultado += "\n" + dato1 + " >= " + dato2;

      JOptionPane.showMessageDialog
        ( null, resultado,
          "Resultados de la comparación",
          JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   }
}
```

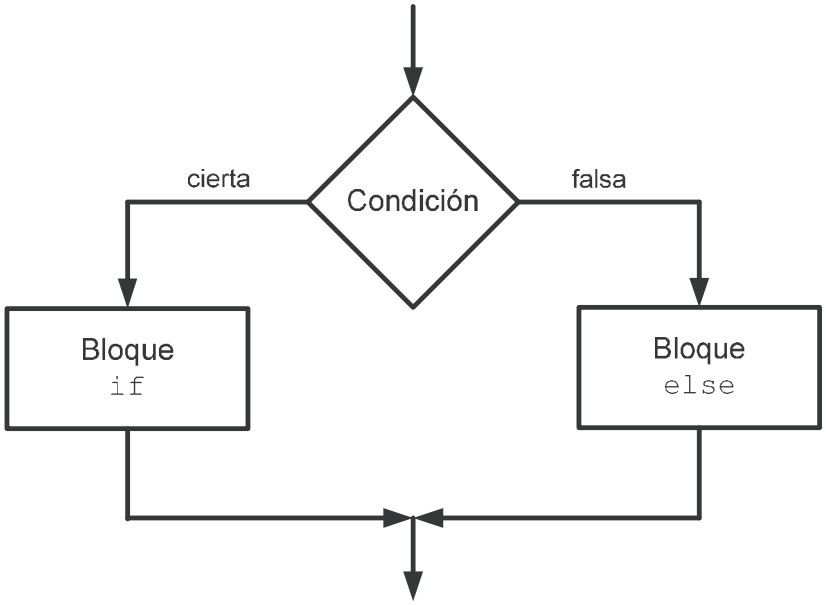






La cláusula `else`

Una sentencia `if`, cuando incluye la cláusula `else`, permite ejecutar un bloque de código si se cumple la condición y otro bloque de código diferente si la condición no se cumple.



Sintaxis

```
if (condición)
   sentencia1;
else
   sentencia2;
```

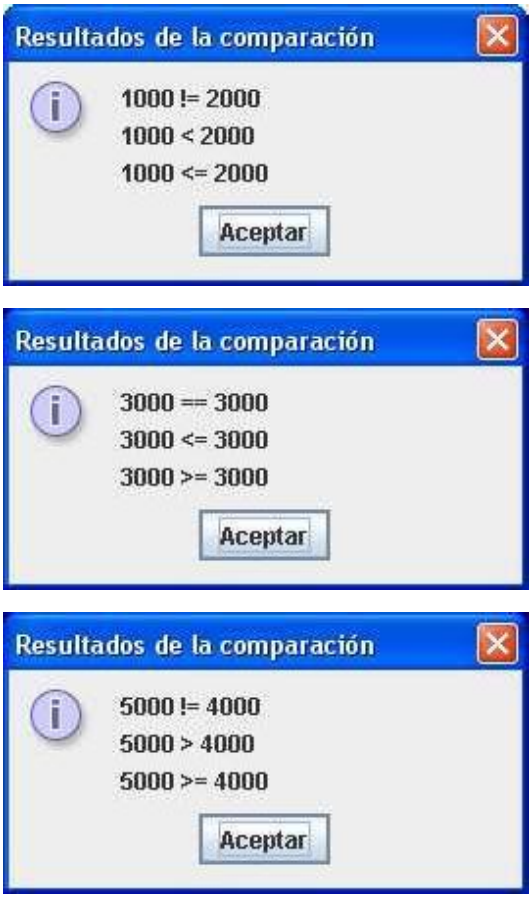
```
if (condición) {
   bloque1
} else {
   bloque2
}
```

Los bloques de código especificados
representan dos alternativas complementarias y excluyentes

Ejemplo

```
import javax.swing.JOptionPane;

public class IfElse
{
  public static void main( String args[] )
  {
    String primerDato, segundoDato;
    String resultado;
    int    dato1, dato2;

    primerDato  = JOptionPane.showInputDialog
                  ( "Primer dato:" );
    segundoDato = JOptionPane.showInputDialog
                  ( "Segundo dato:" );

    dato1 = Integer.parseInt( primerDato );
    dato2 = Integer.parseInt( segundoDato );

    resultado = "";

    if ( dato1 == dato2 )
       resultado += dato1 + " == " + dato2;
    else
       resultado += dato1 + " != " + dato2;

    if ( dato1 < dato2 )
       resultado += "\n" + dato1 + " < " + dato2;
    else
       resultado += "\n" + dato1 + " >= " + dato2;

    if ( dato1 > dato2 )
       resultado += "\n" + dato1 + " > " + dato2;
    else
       resultado += "\n" + dato1 + " <= " + dato2;

    JOptionPane.showMessageDialog
      ( null, resultado,
        "Resultados de la comparación",
        JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
  }
}
```

Con esta versión del programa de comparación de números
obtenemos los mismos resultados que antes,
si bien nos ahorramos tres comparaciones:



La sentencia

```
if (condición)
   sentencia1;
else
   sentencia2;
```

es equivalente a

```
if (condición)
   sentencia1;

if (!condición)
   sentencia2;
```

Nota acerca de la comparación de objetos

Cuando el operador == se utiliza para comparar objetos,
lo que se compara son las referencias a los objetos
y no el estado de los objetos en sí.

Ejemplo

```
public class Complex
{
  private double real;
  private double imag;

  // Constructor

  public Complex (double real, double imag)
  {
    this.real = real;
    this.imag = imag;
  }

  // equals se suele definir para comparar objetos

  public boolean equals (Complex c)
  {
    return (this.real == c.real)
        && (this.imag == c.imag);
  }
}

...
Complex c1 = new Complex(2,1);
Complex c2 = new Complex(2,1);

if (c1 == c2)
   System.out.println("Las referencias son iguales");
else
   System.out.println("Las referencias no son iguales");

if (c1.equals(c2))
   System.out.println("Los objetos son iguales");
else
   System.out.println("Los objetos no son iguales");
```

Encadenamiento

Las sentencias `if` se suelen encadenar:

if … else if …

```
import javax.swing.JOptionPane;

public class IfChain
{
  public static void main( String args[] )
  {
    String entrada;
    String resultado;
    float  nota;


    entrada = JOptionPane.showInputDialog
              ( "Calificación numérica:" );

    nota = Float.parseFloat( entrada );


    if ( nota >= 9 )
       resultado = "Sobresaliente";
    else if ( nota >= 7 )
       resultado = "Notable";
    else if ( nota >= 5 )
       resultado = "Aprobado";
    else
       resultado = "Suspenso";


    JOptionPane.showMessageDialog
      ( null, resultado,
        "Calificación final",
        JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
  }
}
```

El `if` encadenado anterior equivale a:

```
import javax.swing.JOptionPane;

public class IfChain2
{
  public static void main( String args[] )
  {
    String entrada;
    String resultado;
    float  nota;


    entrada = JOptionPane.showInputDialog
              ( "Calificación numérica:" );

    nota = Float.parseFloat( entrada );


    resultado = “”;

    if ( nota >= 9 )
       resultado = "Sobresaliente";

    if ( (nota>=7) && (nota<9) )
       resultado = "Notable";

    if ( (nota>=5) && (nota<7) )
       resultado = "Aprobado";

    if ( nota < 5 )
       resultado = "Suspenso";


    JOptionPane.showMessageDialog
      ( null, resultado,
        "Calificación final",
        JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
  }
}
```

Anidamiento

Las sentencias `if` también se pueden anidar unas dentro de otras.

Ejemplo: Resolución de una ecuación de primer grado **`ax+b=0`**

```
import javax.swing.JOptionPane;

public class IfNested
{
  public static void main( String args[] )
  {
    String entrada;
    String resultado;
    float  a,b,x;

    entrada = JOptionPane.showInputDialog
              ( "Coeficiente a" );
    a = Float.parseFloat( entrada );

    entrada = JOptionPane.showInputDialog
              ( "Coeficiente b" );
    b = Float.parseFloat( entrada );

    if (a!=0) {
       x = -b/a;
       resultado = "La solución es " + x;
    } else {
       if (b!=0) {
          resultado = "No tiene solución.";
       } else {
          resultado = "Solución indeterminada.";
       }
    }

    JOptionPane.showMessageDialog
      ( null, resultado,
        "Solución de la ecuación de primer grado",
        JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
  }
}
```

El `if` anidado anterior equivale a …

```
import javax.swing.JOptionPane;

public class IfNested2
{
  public static void main( String args[] )
  {
    String entrada;
    String resultado;
    float  a,b,x;

    entrada = JOptionPane.showInputDialog
              ( "Coeficiente a" );
    a = Float.parseFloat( entrada );

    entrada = JOptionPane.showInputDialog
              ( "Coeficiente b" );
    b = Float.parseFloat( entrada );

    resultado = "";

    if (a!=0) {
       x = -b/a;
       resultado = "La solución es " + x;
    }

    if ( (a==0) && (b!=0) ) {
       resultado = "No tiene solución.";
    }

    if ( (a==0) && (b==0) ) {
       resultado = "Solución indeterminada.";
    }

    JOptionPane.showMessageDialog
      ( null, resultado,
        "Solución de la ecuación de primer grado",
        JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
  }
}
```

En este caso, se realizan 5 comparaciones en vez de 2.

El operador condicional **`?:`**

Java proporciona una forma de abreviar una sentencia `if`

> El operador condicional `?:`
> permite incluir una condición dentro de una expresión.

Sintaxis

```
variable = condición? expresión1: expresión2;
```

"equivale" a

```
if (condición)
   variable = expresión1;
else
   variable = expresión2;
```

Sólo se evalúa una de las expresiones,
por lo que deberemos ser cuidadosos con los efectos colaterales.

Ejemplos

```
max = (x>y)? x : y;

min = (x<y)? x : y;

med = (x<y)? ((y<z)? y: ((z<x)? x: z)):
             ((x<z)? x: ((z<y)? y: z));

nick = (nombre!=null)? nombre : "desconocido";
```

Selección múltiple con la sentencia **`switch`**
Permite seleccionar entre varias alternativas posibles

Sintaxis

```
switch (expresión) {

  case expr_cte1:
       bloque1;
       break;

  case expr_cte2:
       bloque2;
       break;
  ...
  case expr_cteN:
       bloqueN;
       break;

  default:
       bloque_por_defecto;
}
```

- Se selecciona a partir de la evaluación de una única expresión.
- La expresión del `switch` ha de ser de tipo entero (`int`).
- Los valores de cada caso del `switch` han de ser constantes.
- En Java, cada bloque de código de los que acompañan a un posible valor de la expresión entera ha de terminar con una sentencia `break;`
- La etiqueta `default` marca el bloque de código que se ejecuta por defecto (cuando al evaluar la expresión se obtiene un valor no especificado por los casos anteriores del `switch`).
- En Java, se pueden poner varias etiquetas seguidas acompañando a un único fragmento de código si el fragmento de código que ha de ejecutarse es el mismo para varios valores de la expresión entera que gobierna la ejecución del `switch`.

Ejemplo

```
public class Switch
{
  public static void main( String args[] )
  {
    String resultado;
    int    nota;

    // Entrada de datos
    …

    switch (nota) {
      case 0:
      case 1:
      case 2:
      case 3:
      case 4:
           resultado = "Suspenso";
           break;

      case 5:
      case 6:
           resultado = "Aprobado";
           break;

      case 7:
      case 8:
           resultado = "Notable";
           break;

      case 9:
      case 10:
           resultado = "Sobresaliente";
           break;

      default:
           resultado = "Error";
    }

    // Salida de resultados
    …
  }
}
```

Si tuviésemos que trabajar con datos de tipo real, no podríamos usar `switch` (usaríamos `if`s encadenados).

Estructuras de control repetitivas/iterativas

A menudo es necesario ejecutar
una instrucción o un bloque de instrucciones más de una vez.

Ejemplo

Implementar un programa que calcule
la suma de N números leídos desde teclado.

Podríamos escribir un programa en el que apareciese repetido el código que deseamos que se ejecute varias veces, pero…

- Nuestro programa podría ser demasiado largo.
- Gran parte del código del programa estaría duplicado, lo que dificultaría su mantenimiento en caso de que tuviésemos que hacer cualquier cambio, por trivial que fuese éste.
- Una vez escrito el programa para un número determinado de repeticiones (p.ej. sumar matrices 3x3), el mismo programa no podríamos reutilizarlo si necesitásemos realizar un número distinto de operaciones (p.ej. sumar matrices 4x4).

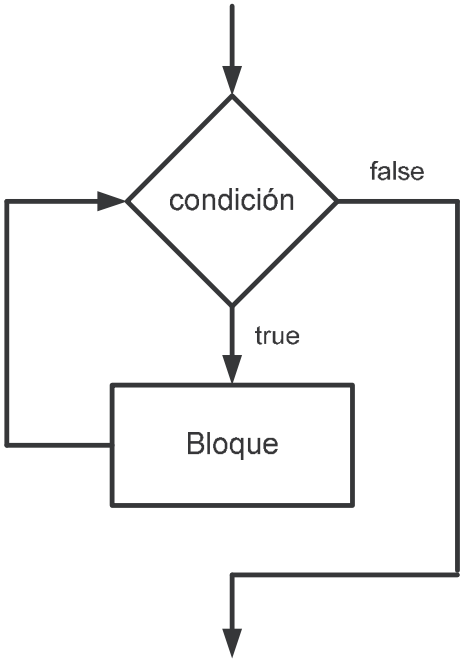
Las **estructuras de control repetitivas o iterativas**, también conocidas como "**bucles**", nos permiten resolver de forma elegante este tipo de problemas. Algunas podemos usarlas cuando conocemos el número de veces que deben repetirse las operaciones. Otras nos permiten repetir un conjunto de operaciones mientras se cumpla una condición.

Iteración: Cada repetición de las instrucciones de un bucle.

El bucle **`while`**

Permite repetir la ejecución de un conjunto de sentencias mientras se cumpla una condición:

```
while (condición)
      sentencia;

while (condición) {
      bloque
}
```



El bucle `while` terminará su ejecución
cuando deje de verificarse la condición que controla su ejecución.

Si, inicialmente, no se cumple la condición,
el cuerpo del bucle no llegará a ejecutarse.

MUY IMPORTANTE

En el cuerpo del bucle debe existir algo que haga variar el valor asociado a la condición que gobierna la ejecución del bucle.

Ejemplo
Tabla de multiplicar de un número

```
public class While1
{
  public static void main( String args[] )
  {
    int n; // Número
    int i; // Contador

    n = Integer.parseInt( args[0] );
    i = 0;

    while (i<=10) {
      System.out.println (n+" x "+i+" = "+(n*i));
      i++;
    }
  }
}
```

Ejemplo
Divisores de un número

```
public class While2
{
  public static void main( String args[] )
  {
    int n;
    int divisor;

    n = Integer.parseInt( args[0] );

    System.out.println("Los divisores son:");

    divisor = n;

    while (divisor>0) {

      if ((n%divisor) == 0)
         System.out.println(divisor);

      divisor--;
    }
  }
}
```

En los ejemplos anteriores,
se conoce de antemano el número de iteraciones
que han de realizarse (cuántas veces se debe ejecutar el bucle):

La expresión del `while` se convierte en una simple comprobación del valor de una variable contador.

El contador es una variable que se incrementa o decrementa en cada iteración y nos permite saber la iteración en la que nos encontramos en cada momento.

En el cuerpo del bucle, siempre se incluye una sentencia

```
contador++;
```

o bien

```
contador--;
```

para que, eventualmente,
la condición del `while` deje de cumplirse.

En otras ocasiones, puede que no conozcamos de antemano cuántas iteraciones se han de realizar.

La condición del `while` puede que tenga un aspecto diferente pero, en el cuerpo del bucle, deberá seguir existiendo algo que modifique el resultado de evaluar la condición.

Ejemplo
Sumar una serie de números
hasta que el usuario introduzca un cero

```
import javax.swing.JOptionPane;

public class While3
{
  public static void main( String args[] )
  {
    float  valor;
    float  suma;

    suma = 0;
    valor = leerValor();

    while (valor!=0) {
          suma += valor;
          valor = leerValor();
    }

    mostrarValor("Suma de los datos", suma);
    System.exit(0);
  }

  private static float leerValor ()
  {
    String entrada;

    entrada = JOptionPane.showInputDialog
              ( "Introduzca un dato:" );

    return Float.parseFloat(entrada);
  }

  private static void mostrarValor
                      (String mensaje, float valor)
  {
    JOptionPane.showMessageDialog
      ( null, valor, mensaje,
        JOptionPane.INFORMATION_MESSAGE );
  }
}
```

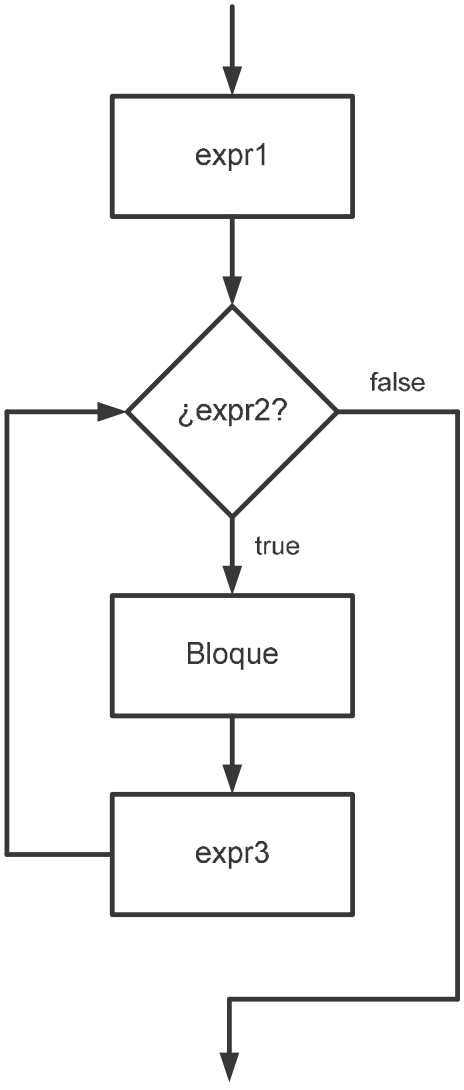
El valor introducido determina en cada iteración
si se termina o no la ejecución del bucle.

El bucle `for`

Se suele emplear en sustitución del bucle `while`
cuando se conoce el número de iteraciones que hay que realizar.

Sintaxis

```
for (expr1; expr2; expr3) {
    bloque;
}
```



Equivalencia entre `for` y `while`

Un fragmento de código
como el que aparecía antes con un bucle `while`:

```
i = 0;
while (i<=10) {
  System.out.println (n+" x "+i+" = "+(n*i));
  i++;
}
```

puede abreviarse si utilizamos un bucle `for`:

```
for (i=0; i<=10; i++) {
  System.out.println (n+" x "+i+" = "+(n*i));
}
```

Como este tipo de estructuras de control es muy común,
el lenguaje nos ofrece una forma más compacta
de representar un bucle cuando sabemos
cuántas veces ha de ejecutarse el cuerpo del bucle.

En general,

```
for (expr1; expr2; expr3) {
    bloque;
}
```

equivale a

```
expr1;
while (expr2) {
      bloque;
      expr3;
}
```

Ejemplo
Cálculo del factorial de un número

Bucle **for**

```
public class FactorialFor
{
  public static void main( String args[] )
  {
    long i,n,factorial;

    n = Integer.parseInt( args[0] );

    factorial = 1;

    for (i=1; i<=n; i++) {
        factorial *= i;
    }

    System.out.println ( "f("+n+") = " + factorial);
  }
}
```

Bucle **while**

```
public class FactorialWhile
{
  public static void main( String args[] )
  {
    long i,n,factorial;

    n = Integer.parseInt( args[0] );

    factorial = 1;
    i = 1;

    while (i<=n) {
        factorial *= i;
        i++;
    }

    System.out.println ( "f("+n+") = " + factorial);
  }
}
```

```
for (expr1; expr2; expr3) {
    bloque;
}
```

En un bucle `for`

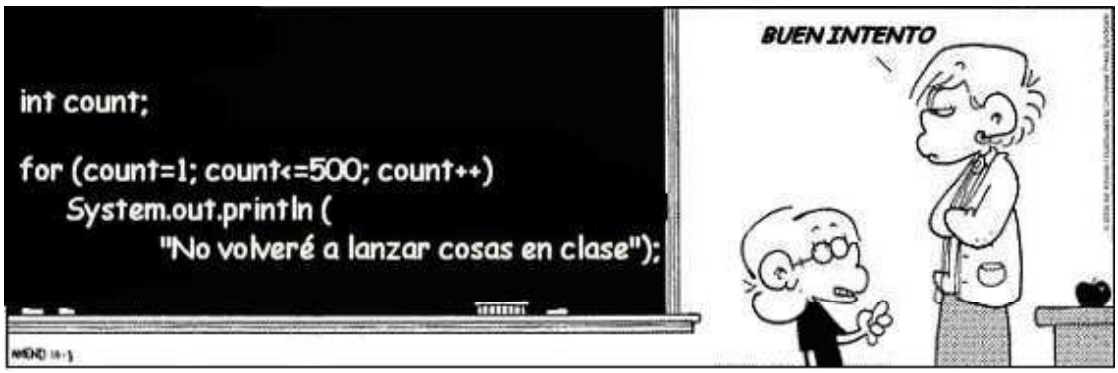
La primera expresión, `expr1`, suele contener inicializaciones de variables separadas por comas. En particular, siempre aparecerá la inicialización de la variable que hace de contador.

- Las instrucciones que se encuentran en esta parte del `for` sólo se ejecutarán una vez antes de la primera ejecución del cuerpo del bucle (`bloque`).

La segunda expresión, `expr2`, es la que contiene una expresión booleana (la que aparecería en la condición del bucle `while` equivalente para controlar la ejecución del cuerpo del bucle).

La tercera expresión, `expr3`, contiene las instrucciones, separadas por comas, que se deben ejecutar al finalizar cada iteración del bucle (p.ej. el incremento/decremento de la variable contador).

El bloque de instrucciones `bloque` es el ámbito del bucle (el bloque de instrucciones que se ejecuta en cada iteración).



Cabecera del bucle	Número de iteraciones
`for (i=0; i<N; i++)`	N
`for (i=0; i<=N; i++)`	N+1
`for (i=k; i<N; i++)`	N-k
`for (i=N; i>0; i--)`	N
`for (i=N; i>=0; i--)`	N+1
`for (i=N; i>k; i--)`	N-k
`for (i=0; i<N; i+=k)`	N/k
`for (i=j; i<N; i+=k)`	(N-j)/k
`for (i=1; i<N; i*=2)`	$\log_2 N$
`for (i=0; i<N; i*=2)`	∞
`for (i=N; i>0; i/=2)`	$\log_2 N + 1$

suponiendo que N y k sean enteros positivos

Bucles infinitos

Un bucle infinito es un bucle que se repite "infinitas" veces:

```
for (;;)          /*bucle infinito*/

while (true)      /*bucle infinito*/
```

Si nunca deja de cumplirse la condición del bucle, nuestro programa se quedará indefinidamente ejecutando el cuerpo del bucle, sin llegar a salir de él.

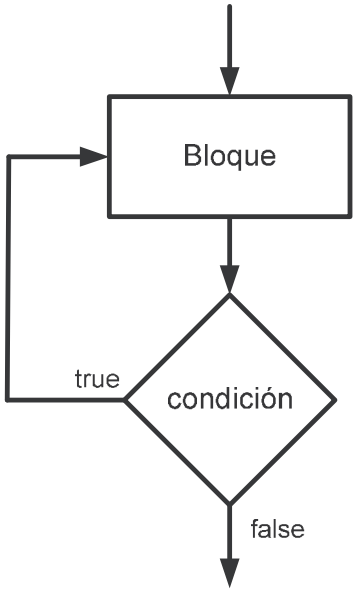
El bucle `do while`

Tipo de bucle, similar al `while`, que realiza la comprobación de la condición después de ejecutar el cuerpo del bucle.

Sintaxis

```
do
  sentencia;
while (condición);
```

```
do {
  bloque
} while (condición);
```



- El bloque de instrucciones se ejecuta, al menos, una vez.

- El bucle `do while` resulta especialmente indicado para validar datos de entrada (comprobar que los valores de entrada obtenidos están dentro del rango de valores que el programa espera).

En todos nuestros programas debemos asegurarnos de que se obtienen datos de entrada válidos antes de realizar cualquier tipo de operación con ellos.

Ejemplo
Cálculo del factorial
comprobando el valor del dato de entrada

```
import javax.swing.JOptionPane;

public class FactorialDoWhile
{
  public static void main( String args[] )
  {
    long n;

    do {
       n = leerEntero(0,20);
    } while ((n<0) || (n>20));

    mostrarMensaje ( "f("+n+") = "+ factorial(n));
    System.exit(0);
  }

  private static long factorial (long n)
  {
    long i;
    long factorial = 1;

    for (i=1; i<=n; i++) {
        factorial *= i;
    }

    return factorial;
  }

  private static int leerEntero (int min, int max)
  {
    String entrada = JOptionPane.showInputDialog
                     ( "Introduzca un valor entero”
                     + “ (entre "+min+" y "+max+"):” );

    return Integer.parseInt(entrada);
  }

  private static void mostrarMensaje (String mensaje)
  {
    JOptionPane.showMessageDialog(null,mensaje);
  }
}
```

Ejemplo
Cálculo de la raíz cuadrada de un número

```
import javax.swing.JOptionPane;

public class Sqrt
{
  public static void main( String args[] )
  {
    double n;
    do {
       n = leerReal ("Introduzca un número positivo");
    } while (n<0);
    mostrarMensaje( "La raíz cuadrada de "+n
                  + " es aproximadamente "+raiz(n));
    System.exit(0);
  }

  private static double raiz (double n)
  {
    double r;     // Raíz cuadrada del número
    double prev;  // Aproximación previa de la raíz

    r = n/2;
    do {
       prev = r;
       r = (r+n/r)/2;
    } while (Math.abs(r-prev) > 1e-6);
    return r;
  }

  private static double leerReal (String mensaje)
  {
    String entrada;
    entrada = JOptionPane.showInputDialog(mensaje);
    return Double.parseDouble(entrada);
  }

  private static void mostrarMensaje (String mensaje)
  {
    JOptionPane.showMessageDialog(null,mensaje);
  }
}
```

Bucles anidados

Los bucles también se pueden anidar:

```
for (i=0; i<N;i++) {
    for (j=0; j<N; j++) {
        printf("(%d,%d) ",i,j);
    }
}
```

genera como resultado:

```
(0,0) (0,1) (0,2) … (0,N)
(1,0) (1,1) (1,2) … (1,N)
                …
(N,0) (N,1) (N,2) … (N,N)
```

Ejemplo

```
int n,i,k;

…
n = 0;                              // Paso 1
for (i=1; i<=2; i++) {              // Paso 2
    for (k=5; k>=1; k-=2) {         // Paso 3
        n = n + i + k;              // Paso 4
    }                               // Paso 5
}                                   // Paso 6
…                                   // Paso 7
```

Paso	1	2	3	4	5	3	4	5	3	4	5	3	6	2	3	4	5	3	4	5	3	6	2	7
N	0			6			10			12						19			24			27					
i	?	1											2										3			
k	?		5		3			1			−1				5		3			1			−1				

Cuestiones de estilo

Escribimos código para que lo puedan leer otras personas,
no sólo para que lo traduzca el compilador.

Identificadores

- Los identificadores deben ser **descriptivos**
  - `p, i, s...`

    `precio, izquierda, suma...`

- En ocasiones, se permite el uso de nombres cortos para variables locales cuyo significado es evidente (p.ej. bucles controlados por contador)

```
for (elemento=0; elemento<N; elemento++ )…

for (i=0; i<N; i++) …
```

Constantes

- Se considera una mala costumbre incluir literales de tipo numérico (**"números mágicos"**) en medio del código. Se prefiere la definición de constantes simbólicas (con `final`).
  - `for (i=0; i<79; i++) …`

    `for (i=0; i<columnas-1; i++) …`

Expresiones

- *Expresiones booleanas:*
  Es aconsejable escribirlas como se dirían en voz alta.

  ß
  ```
  if ( !(bloque<actual) ) …

  if ( bloque >= actual ) …
  ```

- *Expresiones complejas:*
  Es aconsejable dividirlas para mejorar su legibilidad

  ß
  ```
  x += ( xp = ( 2*k<(n-m) ? c+k: d-k ));

  if ( 2*k < n-m )
     xp = c+k;
  else
     xp = d-k;

  x += xp;


  max = (a > b) ? a : b;
  ```

Comentarios

- Comentarios descriptivos: Los comentarios deben comunicar algo. Jamás se utilizarán para "parafrasear" el código y repetir lo que es obvio.

  ß
  ```
  i++;     /* Incrementa el contador */


  /* Recorrido secuencial de los datos*/

  for (i=0; i<N; i++) ...
  ```

- Sangrías:
  Conviene utilizar espacios en blanco o separadores para delimitar el ámbito de las estructuras de control de nuestros programas.

- Líneas en blanco:
  Para delimitar claramente los distintos bloques de código en nuestros programas dejaremos líneas en blanco entre ellos.

- Salvo en la cabecera de los bucles `for`, sólo incluiremos una sentencia por línea de código.

- Sean cuales sean las convenciones utilizadas al escribir código (p.ej. uso de sangrías y llaves), hay que ser consistente en su utilización.

```
while (…) {
  …
}

for (…;…;…) {
  …
}

if (…) {
   …
}
```

```
while (…)
{
   …
}

for (…;…;…)
{
   …
}

if (…)
{
   …
}
```

El código bien escrito es más fácil de leer, entender y mantener (además, seguramente tiene menos errores)

Estructuras de control
Relación de ejercicios

1. ¿Cuántas veces se ejecutaría el cuerpo de los siguientes bucles `for`?

   ```
   for (i=1; i<10; i++) …

   for (i=30; i>1; i-=2) …

   for (i=30; i<1; i+=2) …

   for (i=0; i<30; i+=4) …
   ```

2. Ejecute paso a paso el siguiente bucle:

   ```
   c = 5;

   for (a=1; a<5; a++)
       c = c – 1;
   ```

3. Escriba un programa que lea una serie de N datos y nos muestre: el número de datos introducidos, la suma de los valores de los datos, la media del conjunto de datos, el máximo, el mínimo, la varianza y la desviación típica.

   PISTA: La varianza se puede calcular a partir de la suma de los cuadrados de los datos.

4. Diseñe un programa que lea los coeficientes de una ecuación de segundo grado $ax^2+bx+c=0$ y calcule sus dos soluciones. El programa debe responder de forma adecuada para cualquier caso que se pueda presentar.

5. Diseñe un programa que lea los coeficientes de un sistema de dos ecuaciones lineales con dos incógnitas y calcule su solución. El programa debe responder de forma adecuada cuando el sistema de ecuaciones no sea compatible determinado.

$$\left.\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}\right\}$$

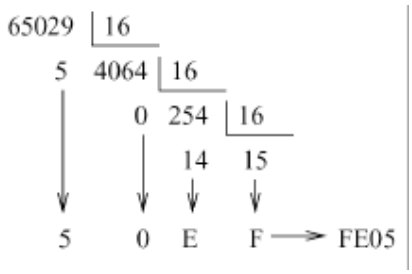
6. Dada una medida de tiempo expresada en horas, minutos y segundos con valores arbitrarios, elabore un programa que transforme dicha medida en una expresión correcta. Por ejemplo, dada la medida *3h 118m 195s*, el programa deberá obtener como resultado *5h 1m 15s*. Realice el programa sin utilizar los operadores de división entera (`/` y `%`).

7. Escriba un programa en C que nos calcule el cambio que debe dar la caja de un supermercado: Dado un precio y una cantidad de dinero, el programa nos dirá cuántas monedas deben darse como cambio de tal forma que el número total de monedas sea mínimo. Realice el programa sin utilizar los operadores de división entera (`/` y `%`).

8. Implemente un programa que lea un número decimal y lo muestre en pantalla en hexadecimal (base 16). El cambio de base se realiza mediante divisiones sucesivas por 16 en las cuales los restos determinan los dígitos hexadecimales del número según la siguiente correspondencia:

Resto	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Dígito	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Por ejemplo:



$65029|_{10} = FE05|_{16}$

9. Escriba una función (un método) que obtenga la letra del DNI a partir del número. Para ello debe obtener el resto de dividir el número entre 23. La letra asociada al número vendrá dada por este resto en función de la siguiente tabla:

0 → T	1 → R	2 → W	3 → A	4 → G	5 → M
6 → Y	7 → F	8 → P	9 → D	10 → X	11 → B
12 → N	13 → J	14 → Z	15 → S	16 → Q	17 → V
18 → H	19 → L	20 → C	21 → K	22 → E	23 → T

10. Escriba una función que, a partir de los dígitos de un ISBN, calcule el carácter de control con el que termina todo ISBN. Para calcular el carácter de control, debe multiplicar cada dígito por su posición (siendo el dígito de la izquierda el que ocupa la posición 1), sumar los resultados obtenidos y hallar el resto de dividir por 11. El resultado será el carácter de control, teniendo en cuenta que el carácter de control es ‘X’ cuando el resto vale 10.

11. Implemente un programa que calcule la suma de los 100 primeros términos de las siguientes sucesiones:

$$a_n = a_{n-1} + n$$

$$a_n = \frac{a_{n-1}}{n}$$

$$a_n = (-1)^n \frac{n^2 - 1}{2n + 1}$$

12. Realice un programa que calcule los valores de la función:

$$f(x, y) = \frac{\sqrt{x}}{y^2 - 1}$$

para los valores de *(x,y)* con *x=-50,-48…0…48,50* e *y=-40,-39…0…39,40*

13. Implemente funciones que nos permitan calcular x^n, $n!$ y $\binom{n}{m}$

14. Escriba un programa que muestre en pantalla todos los números primos entre 1 y n, donde n es un número positivo que recibe el programa como parámetro.

15. Escriba una función que, dados dos números enteros, nos diga si cualquiera de ellos divide o no al otro.

16. Implemente un programa que calcule los divisores de un número entero.

17. Implemente una función que nos devuelva el máximo común divisor de dos números enteros.

18. Implemente una función que nos devuelva el mínimo común múltiplo de dos números enteros.

19. Diseñe e implemente un programa que realice la descomposición en números primos de un número entero.

20. Escriba un programa que lea números enteros hasta que se introduzcan 10 números o se introduzca un valor negativo. El programa mostrará entonces el valor medio de los números introducidos (sin contar el número negativo en caso de que éste se haya indicado).

21. Implemente una función que nos diga si un número ha conseguido o no el reintegro en el sorteo de la ONCE. Un número de cinco cifras consigue el reintegro si su primera o última cifra coincide con la primera o última cifra del número agraciado en el sorteo.

22. Diseñe un programa para jugar a adivinar un número entre 0 y 100. El programa irá dando pistas al jugador indicándole si el número introducido por el jugador es menor o mayor que el número que tiene que adivinar. El juego termina cuando el jugador adivina el número o decide terminar de jugar (por ejemplo, escribiendo un número negativo).

23. Amplíe el programa del ejercicio anterior permitiendo que el jugador juegue tantas veces como desee. El programa deberá mantener las estadísticas del jugador y mostrárselas al final de cada partida (número medio de intentos para adivinar el número, número de veces que el jugador abandona, mejor partida y peor partida).

24. Aplicar el método de Newton-Raphson a los siguientes problemas:

    a. Calcular la raíz cuadrada de un número.

    b. Calcular la raíz cúbica de un número.

    c. Calcular la raíz n-ésima de un número

    El método de Newton-Raphson en un método general para la obtención de los ceros de una función. Para ello se van generando los términos de la sucesión

$$x_{n+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

    donde *f(x)* es la función cuyo cero deseamos obtener y *f'(x)* es la derivada de la función. Por ejemplo, para calcular la raíz cuadrada de un número *n*, hemos de obtener un cero de la función $f(x) = x^2 - n$

    Partiendo de un valor inicial x0 cualquiera, se van generando términos de la sucesión hasta que la diferencia entre dos términos consecutivos de la sucesión sea inferior a una precisión especificada de antemano (p.ej. 10^{-6}) .

25. Uso y manipulación de fechas:

    a. Diseñe una clase para representar fechas.

    b. Escriba un método estático que nos diga el número de días de un mes (¡ojo con los años bisiestos!).

    c. Añada a su clase un método que nos indique el número de días del mes al que pertenece la fecha.

    d. Incluya, en su clase `Fecha`, un método que nos diga el número de días que hay desde una fecha determinada hasta otra.

    e. Implemente un método que nos diga el día de la semana correspondiente a una fecha concreta (p.ej. el 1 de diciembre de 2004 fue miércoles).

    f. Escriba un programa que muestre el calendario de un mes concreto.

NOTA: Compruebe el correcto funcionamiento de todos los programas con varios valores para sus entradas. En el caso de los ejercicios en que se pide la creación de un método o función, escriba programas auxiliares que hagan uso del método creado.

Vectores y matrices

Arrays

- Declaración
- Creación
- Acceso a los elementos de un array
- Manipulación de vectores y matrices

Algoritmos de ordenación

- Ordenación por selección
- Ordenación por inserción
- Ordenación por intercambio directo (método de la burbuja)
- Ordenación rápida (QuickSort)

Algoritmos de búsqueda

- Búsqueda lineal
- Búsqueda binaria

Apéndice: Cadenas de caracteres

Arrays

Un array es una estructura de datos
que contiene una colección de datos del mismo tipo

Ejemplos

Temperaturas mínimas de los últimos treinta días

Valor de las acciones de una empresa durante la última semana

…

Propiedades de los arrays

- Los arrays se utilizan como **contenedores** para almacenar datos relacionados (en vez de declarar variables por separado para cada uno de los elementos del array).

- Todos los **datos** incluidos en el array son **del mismo tipo**. Se pueden crear arrays de enteros de tipo `int` o de reales de tipo `float`, pero en un mismo array no se pueden mezclar datos de tipo `int` y datos de tipo `float`.

- El tamaño del array se establece cuando se crea el array (con el operador `new`, igual que cualquier otro objeto).

- A los elementos del array se accederá a través de la posición que ocupan dentro del conjunto de elementos del array.

Terminología

Los arrays unidimensionales se conocen con el nombre de **vectores**.

Los arrays bidimensionales se conocen con el nombre de **matrices**.

Declaración

Para declarar un array,
se utilizan corchetes para indicar que se trata de un array
y no de una simple variable del tipo especificado.

Vector (array unidimensional):

```
tipo identificador[];
```

o bien

```
tipo[] identificador;
```

donde

`tipo` es el tipo de dato de los elementos del vector

`identificador` es el identificador de la variable.

Matriz (array bidimensional):

```
tipo identificador[][];
```

o bien

```
tipo[][] identificador;
```

NOTA: No es una buena idea que el identificador del array termine en un dígito, p.ej. vector3

Creación

Los arrays se crean con el operador `new`.

Vector (array unidimensional):

```
vector = new tipo[elementos];
```

Entre corchetes se indica el tamaño del vector.

`tipo` debe coincidir con el tipo con el que se haya declarado el vector.

`vector` debe ser una variable declarada como `tipo[]`

Ejemplos

```
float[] notas = new float[ALUMNOS];
int[] temperaturas = new int[7];
```

Matriz (array bidimensional):

```
matriz = new tipo[filas][columnas];
```

Ejemplo

```
int[][] temperaturas = new int[12][31];
```

Uso

Para acceder a los elementos de un array,
utilizamos índices
(para indicar la posición del elemento dentro del array)

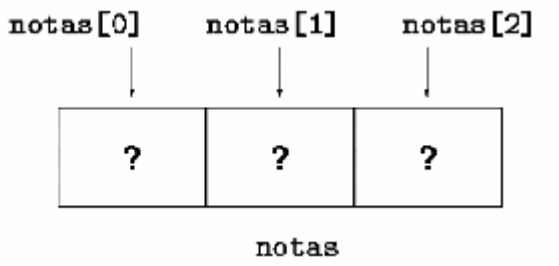
Vector (array unidimensional):

```
vector[índice]
```

- En Java, el índice de la primera componente de un vector es siempre 0.
- El tamaño del array puede obtenerse utilizando la propiedad `vector.length`
- Por tanto, el índice de la última componente es `vector.length-1`

Ejemplo

```
float[] notas = new float[3];
```



Matriz (array bidimensional):

```
matriz[índice1][índice2]
```

Una matriz, en realidad, es un vector de vectores:

- En Java, el índice de la primera componente de un vector es siempre 0, por lo que `matriz[0][0]` será el primer elemento de la matriz.

- El tamaño del array puede obtenerse utilizando la propiedad `array.length`:

  - `matriz.length` nos da el número de filas

  - `matriz[0].length` nos da el número de columnas

- Por tanto, el último elemento de la matriz es `matriz[matriz.length-1][matriz[0].length-1]`

Inicialización en la declaración
Podemos asignarle un valor inicial
a los elementos de un array en la propia declaración

```
int vector[] = {1, 2, 3, 5, 7};
int matriz[][] = { {1,2,3}, {4,5,6} };
```

El compilador deduce automáticamente las dimensiones del array.

Manipulación de vectores y matrices

Las operaciones se realizan componente a componente

Ejemplo: Suma de los elementos de un vector

```
static float media (float datos[])
{
  int   i;
  int   n = datos.length;
  float suma = 0;

  for (i=0; i<n; i++)
      suma = suma + datos[i];

  return suma/n;
}
```

No es necesario utilizar todos los elementos de un vector, por lo que, al trabajar con ellos, se puede utilizar una variable entera adicional que nos indique el número de datos que realmente estamos utilizando:

El tamaño del vector nos dice cuánta memoria se ha reservado para almacenar datos del mismo tipo, no cuántos datos del mismo tipo tenemos realmente en el vector.

Ejemplo: Suma de los n primeros elementos de un vector

```
static float media (float datos[], int n)
{
  int   i;
  float suma = 0;

  for (i=0; i<n; i++)
      suma = suma + datos[i];

  return suma/n;
}
```

Ejemplo

```
public class Vectores
{
  public static void main (String[] args)
  {
    int pares[] = { 2, 4, 6, 8, 10 };
    int impares[] = { 1, 3, 5, 7, 9 };

    mostrarVector(pares);
    System.out.println("MEDIA="+media(pares));

    mostrarVector(impares);
    System.out.println("MEDIA="+media(impares));
  }

  static void mostrarVector (int datos[])
  {
    int   i;

    for (i=0; i<datos.length; i++)
        System.out.println(datos[i]);
  }

  static float media (int datos[])
  {
    int i;
    int n = datos.length;
    int suma = 0;

    for (i=0; i<n; i++)
        suma = suma + datos[i];

    return suma/n;
  }
}
```

```
static int[] leerVector (int datos)
{
  int   i;
  int[] vector = new int[datos];

  for (i=0; i<datos; i++)
      vector[i] = leerValor();

  return vector;
}
```

IMPORTANTE:

Cuando se pasa un array como parámetro,
se copia una referencia al array y no el conjunto de valores en sí.

Por tanto, tenemos que tener cuidado con los efectos colaterales
que se producen si, dentro de un módulo,
modificamos un vector que recibimos como parámetro.

Ejemplo
El siguiente método lee los elementos de un vector ya creado

```
static void leerVector (int[] datos)
{
  int   i;

  for (i=0; i<datos.length; i++)
      datos[i] = leerValor();
}
```

Copia de arrays

La siguiente asignación sólo copia las referencias,
no crea un nuevo array:

```
int[] datos = pares;
```

Para copiar los elementos de un array,
hemos de crear un nuevo array y copiar los elementos uno a uno

```
int[] datos = new int[pares.length];

for (i=0; i<pares.length; i++)
    datos[i] = pares[i]
```

También podemos utilizar una función predefinida
en la biblioteca de clases estándar de Java:

```
System.arraycopy(from,fromIndex,to,toIndex,n);
```

```
int[] datos = new int[pares.length];
System.arraycopy(pares,0,datos,0,pares.length);
```

EXTRA:
La biblioteca de clases de Java incluye una clase auxiliar llamada **`java.util.Arrays`** que incluye como métodos algunas de las tareas que se realizan más a menudo con vectores:

- `Arrays.sort(v)` ordena los elementos del vector.
- `Arrays.equals(v1,v2)` comprueba si dos vectores son iguales.
- `Arrays.fill(v,val)` rellena el vector `v` con el valor `val`.
- `Arrays.toString(v)` devuelve una cadena que representa el contenido del vector.
- `Arrays.binarySearch(v, k)` busca el valor `k` dentro del vector `v` (que previamente ha de estar ordenado).

Ejemplos

Un programa que muestra los parámetros
que le indicamos en la línea de comandos:

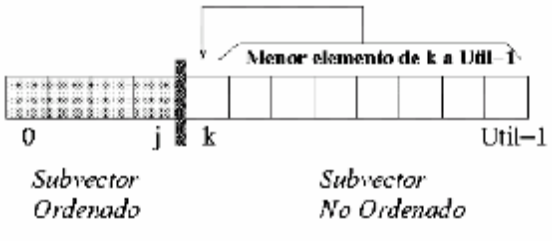
```
public class Eco
{
  public static void main(String args[])
  {
    int i;

    for (i=0; i<args.length; i++)
        System.out.println(args[i]);
  }
}
```

Un método que muestra el contenido de una matriz:

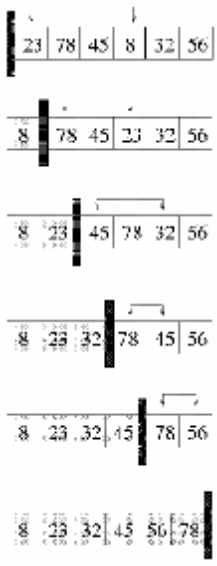
```
public static void mostrarMatriz (double matriz[][])
{
  int i,j;
  int filas = matriz.length;
  int columnas = matriz[0].length;

  // Recorrido de las filas de la matriz

  for (i=0; i<filas; i++) {

      // Recorrido de las celdas de una fila

      for (j=0; j<columnas; j++) {

          System.out.println ( "matriz["+i+"]["+j+"]="
                               + matriz[i][j] );
      }
  }
}
```

Algoritmos de ordenación

Ordenación por selección

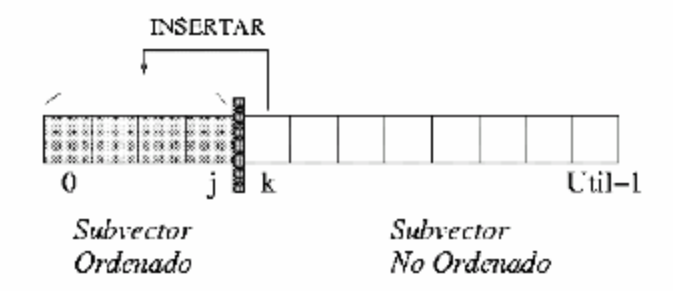



```
static void ordenarSeleccion (double v[])
{
  double tmp;
  int i, j, pos_min;
  int N = v.length;

  for (i=0; i<N-1; i++) {

      // Menor elemento del vector v[i..N-1]

      pos_min = i;

      for (j=i+1; j<N; j++)
          if (v[j]<v[pos_min])
             pos_min = j;

      // Coloca el mínimo en v[i]

      tmp = v[i];
      v[i] = v[pos_min];
      v[pos_min] = tmp; =
  }
}
```

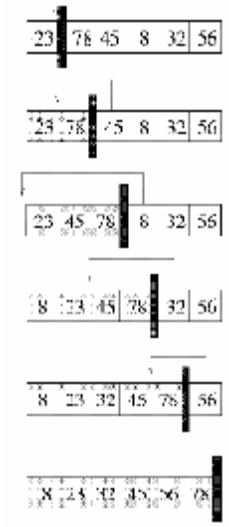


En cada iteración, se selecciona el menor elemento del subvector no ordenado y se intercambia con el primer elemento de este subvector.

Ordenación por inserción

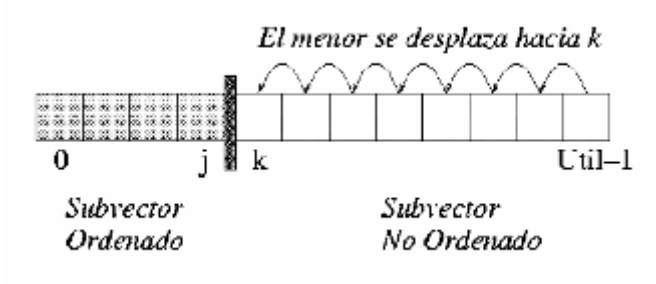



```
static void ordenarInsercion (double v[])
{
  double tmp;
  int i, j;
  int N = v.length;

  for (i=1; i<N; i++) {

      tmp = v[i];

      for (j=i; (j>0) && (tmp<v[j-1]); j--)
          v[j] = v[j-1];

      v[j] = tmp;
  }
}
```



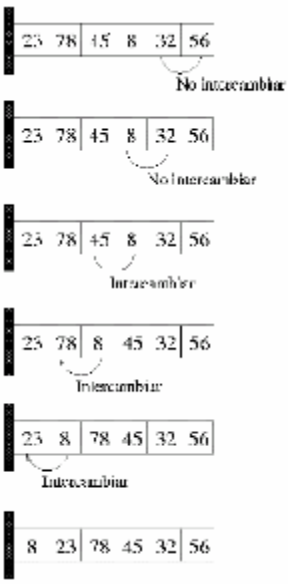
En cada iteración, se inserta un elemento del subvector no ordenado en la posición correcta dentro del subvector ordenado.

Ordenación por intercambio directo (método de la burbuja)

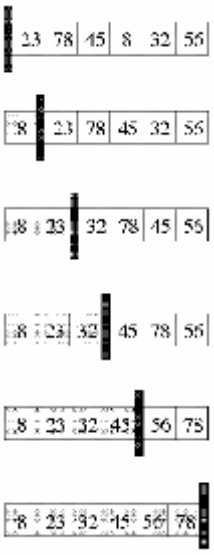



```
static void ordenarBurbuja (double v[])
{
  double tmp;
  int i, j;
  int N = v.length;

  for (i=1; i<N; i++)
      for (j=N-1; j>=i; j--)
          if (v[j] < v[j-1]) {
             tmp = v[j];
             v[j] = v[j-1];
             v[j-1] = tmp;
          }
}
```

En cada iteración



Estado del vector tras cada iteración:



Ordenación rápida (QuickSort)

1. Se toma un elemento arbitrario del vector, al que denominaremos pivote (p).

2. Se divide el vector de tal forma que todos los elementos a la izquierda del pivote sean menores que él, mientras que los que quedan a la derecha son mayores que él.

3. Ordenamos, por separado, las dos zonas delimitadas por el pivote.

```
static void quicksort
            (double v[], int izda, int dcha)
{
  int pivote;  // Posición del pivote

  if (izda<dcha) {

     pivote = partir (v, izda, dcha);

     quicksort (v, izda, pivote-1);

     quicksort (v, pivote+1, dcha);
  }
}
```

Uso:

```
quicksort (vector, 0, vector.length-1);
```

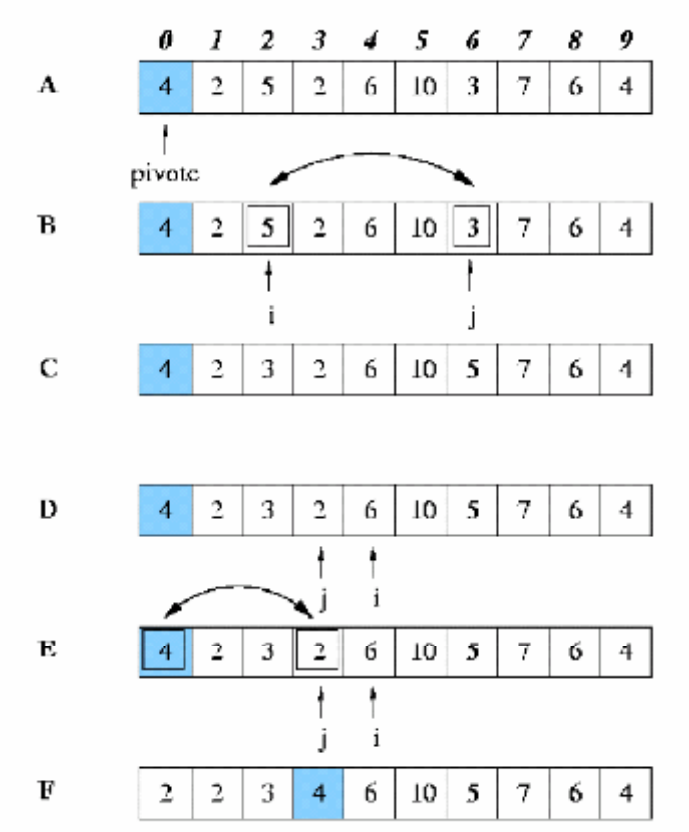
Obtención del pivote

Mientras queden elementos mal colocados respecto al pivote:

a. Se recorre el vector, de izquierda a derecha, hasta encontrar un elemento situado en una posición i tal que v[i] > p.

b. Se recorre el vector, de derecha a izquierda, hasta encontrar otro elemento situado en una posición j tal que v[j] < p.

c. Se intercambian los elementos situados en las casillas i y j (de modo que, ahora, v[i] < p < v[j]).



```
  /**
   * División del vector en dos partes
   * @see quicksort
   *
   * @param primero Índice del primer elemento
   * @param ultimo Índice del último elemento
   * @return Posición del pivote
   *
   * @pre (primero>=0)
   *      && (primero<=ultimo)
   *      && (ultimo<v.length)
   */

private static int partir
        (double v[], int primero, int ultimo)
{
  double pivote = v[primero]; // Valor del pivote
  double temporal;            // Variable auxiliar
  int izda = primero+1;
  int dcha = ultimo;

  do {  // Pivotear…

    while ((izda<=dcha) && (v[izda]<=pivote))
          izda++;

    while ((izda<=dcha) && (v[dcha]>pivote))
          dcha--;

    if (izda < dcha) {
       temporal = v[izda];
       v[izda] = v[dcha];
       v[dcha] = temporal;
       dcha--;
       izda++;
    }

  } while (izda <= dcha);

  // Colocar el pivote en su sitio
  temporal = v[primero];
  v[primero] = v[dcha];
  v[dcha] = temporal;

  return dcha; // Posición del pivote
}
```

Algoritmos de búsqueda

Búsqueda lineal = Búsqueda secuencial

```
// Búsqueda lineal de un elemento en un vector
// - Devuelve la posición de “dato” en el vector
// - Si “dato” no está en el vector, devuelve -1

static int buscar (double vector[], double dato)
{
  int i;
  int N = vector.length;
  int pos = -1;

  for (i=0; i<N; i++)
      if (vector[i]==dato)
         pos = i;

  return pos;
}
```

Versión mejorada

```
// Búsqueda lineal de un elemento en un vector
// - Devuelve la posición de “dato” en el vector
// - Si “dato” no está en el vector, devuelve -1

static int buscar (double vector[], double dato)
{
  int i;
  int N = vector.length;
  int pos = -1;

  for (i=0; (i<N) && (pos==-1); i++)
      if (vector[i]==dato)
         pos = i;

  return pos;
}
```

Búsqueda binaria

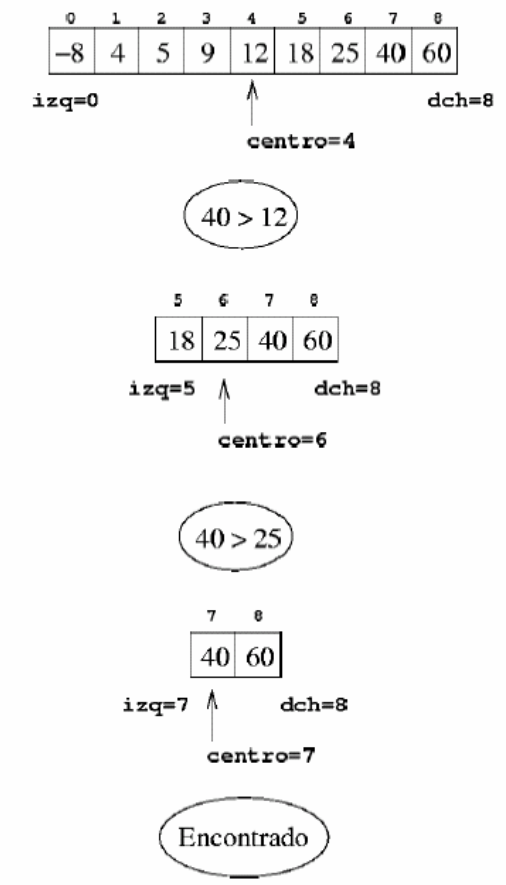
Precondición

El vector ha de estar ordenado

Algoritmo

Se compara el dato buscado
con el elemento en el centro del vector:

- Si coinciden, hemos encontrado el dato buscado.
- Si el dato es mayor que el elemento central del vector, tenemos que buscar el dato en segunda mitad del vector.
- Si el dato es menor que el elemento central del vector, tenemos que buscar el dato en la primera mitad del vector.



```
// Búsqueda binaria de un elemento en un vector
// - Devuelve la posición de “dato” en el vector
// - Si “dato” no está en el vector, devuelve -1

// Implementación recursiva
// Uso: binSearch(vector,0,vector.length-1,dato)

static int binSearch
    (double v[], int izq, int der, double buscado)
{
  int centro = (izq+der)/2;

  if (izq>der)
     return -1;
  else if (buscado==v[centro])
     return centro;
  else if (buscado<v[centro])
     return binSearch(v, izq, centro-1, buscado);
  else
     return binSearch(v, centro+1, der, buscado);
}

// Implementación iterativa
// Uso: binSearch (vector, dato)

static int binSearch (double v[], double buscado)
{
  int izq = 0;
  int der = v.length-1;
  int centro = (izq+der)/2;

  while ((izq<=der) && (v[centro]!=buscado)) {

    if (buscado<v[centro])
       der = centro – 1;
    else
       izq = centro + 1;

    centro = (izq+der)/2;
  }

  if (izq>der)
      return -1;
  else
     return centro;
}
```

Apéndice: Cadenas de caracteres

Una cadena de caracteres no es más que un vector de caracteres.

La clase `java.lang.String`,
que se emplea para representar cadenas de caracteres en Java,
incluye distintos métodos que nos facilitan algunas de las operaciones que se suelen realizar con cadenas de caracteres:

- El método `substring` nos permite obtener una subcadena:

```
String java="Java";
String s = java.substring(0,3);
System.out.println(s);            // Jav
```

- El método `charAt(n)` nos devuelve el carácter que se encuentra en la posición n de la cadena:

```
String java="Java";
char c = java.atChar(2);
System.out.println(c);            // v
```

- El método `indexOf(s)` nos devuelve la posición de una subcadena dentro de la cadena:

```
String java="Java";
int p = java.indexOf("av");
System.out.println(p);            // 1
```

- El método `replace(old,new)` reemplaza subcadenas:

```
String java="Java";
java.replace("ava","ini");
System.out.println(java);         // Jini
```

- El método `equals(s)` se usa para comprobar si dos cadenas son iguales:

```
if (s.equals("Hola")) {
   …
}
```

RECORDATORIO: el operador == no debe utilizarse para comparar objetos.

- El método `startsWith(s)` nos dice si una cadena empieza con un prefijo determinado:

```
if (s.startsWith("get")) {
   …
}
```

- El método `endsWith(s)` nos dice si una cadena termina con un sufijo determinado:

```
if (s.endsWith(".html")) {
   …
}
```

- El método `length()` devuelve la longitud de la cadena.

…

La clase `java.lang.String` incluye decenas de métodos.

La lista completa de métodos y los detalles de utilización de cada método se pueden consultar en la ayuda del JDK.

Vectores y matrices
Relación de ejercicios

1. Dado un vector de números reales:

    a. Escriba un método `max` que nos devuelva el máximo de los valores incluidos en el vector.

    b. Escriba un método `min` que nos devuelva el mínimo de los valores incluidos en el vector.

    c. Escriba un método `media` que nos devuelva la media de los valores incluidos en el vector.

    d. Escriba un método `varianza` que nos devuelva la varianza de los valores incluidos en el vector.

    e. Escriba un método `mediana` que nos devuelva la mediana de los valores incluidos en el vector.

    f. Escriba un método `moda` que nos devuelva la moda de los valores incluidos en el vector

    g. Escriba un método `percentil(n)` que nos delvuelva el valor correspondiente al percentil n en el conjunto de valores del vector.

2. Implemente una clase en Java, llamada `Serie`, que encapsule un vector de números reales e incluya métodos (no estáticos) que nos permitan calcular todos los valores mencionados en el ejercicio anterior a partir de los datos encapsulados por un objeto de tipo `Serie`.

3. Dado un vector de números reales, escriba un método que nos devuelva el máximo y el mínimo de los valores incluidos en el vector.

4. Dado un vector, implemente un método que inserte un elemento en una posición dada del vector.

    NOTA: Insertar un elemento en el vector desplaza una posición hacia la derecha a los elementos del vector que han de quedar detrás del elemento insertado. Además, la inserción ocasiona la “desaparición” del último elemento del vector.

5. Implemente un método llamado secuencia que realice la búsqueda de la secuencia en orden creciente más larga dentro de un vector de enteros. El método ha de devolver tanto la posición de la primera componente de la secuencia como el tamaño de la misma.

6. Una cadena de ADN se representa como una secuencia circular de bases (adenina, timina, citosina y guanina) que es única para cada ser vivo, por ejemplo:

A	T	G
T		C
A	T	G

   Dicha cadena se puede representar como un vector de caracteres recorriéndola en sentido horario desde la parte superior izquierda:

A	T	G	C	G	T	A	T

   Se pide diseñar una clase que represente una secuencia de ADN e incluya un método booleano que nos devuelva `true` si dos cadenas de ADN coinciden.

   MUY IMPORTANTE: La secuencia de ADN es cíclica, por lo que puede comenzar en cualquier posición. Por ejemplo, las dos secuencias siguientes coinciden:

A	T	G	C	G	T	A	T

A	T	A	T	G	C	G	T

7. Dado un vector de números reales, escriba un método que ordene los elementos del vector **de mayor a menor**.

8. Dado un vector de números reales, escriba un método que ordene los elementos del vector de tal forma que los números pares aparezcan antes que los números impares. Además, los números pares deberán estar ordenados de forma ascendente, mientras que los números impares deberán estar ordenados de forma descendente. Esto es, el vector {1,2,3,4,5,6} quedará como {2,4,6,5,3,1}.

9. Crear una clase `Matriz` para manipular matrices que encapsule un array bidimensional de números reales.

   a. Incluya en la clase métodos que nos permitan acceder y modificar de forma segura los elementos de la matriz (esto es, las variables de instancia deben ser privadas y los métodos han de comprobar la validez de sus parámetros).

   b. Escriba un método que nos permita sumar matrices.

   c. Implemente un método que nos permita multiplicar matrices.

   d. Cree un método con el que se obtenga la traspuesta de una matriz.

10. En Java, para generar números pseudoaleatorios, se puede utilizar la función `Math.random()` definida en la clase `java.lang.Math`. Dicha función genera una secuencia de números pseudoaleatorios que se supone sigue una distribución uniforme (esto es, todos los valores aparecerán con la misma probabilidad). Escriba un programa que compruebe si el generador de números pseudoaleatorios de Java genera realmente números aleatorios con una distribución uniforme.

    Sugerencia: Genere un gran número de números aleatorios (entre 0 y 100, por ejemplo) y compruebe que la distribución resultante del número de veces que aparece cada número es (más o menos) uniforme. Por ejemplo, mida la dispersión de la distribución resultante utilizando una medida como la varianza (y reutilice la clase `Serie` del ejercicio 2).

11. Realizar una simulación de Monte Carlo para aproximar el valor del área bajo una curva f(x), en $x \in [a, b]$.

    Algoritmo: Generar puntos aleatorios en el rectángulo de extremos (a, 0) y (b, m), con $m = \max_{a \le x \le b} f(x)$, y contar el número de puntos que caen por debajo de la curva.

    NOTA: Este método permite generar números pseudoaleatorios que sigan cualquier distribución que nosotros deseemos. Por ejemplo, probar con una distribución normal.

12. Crear un programa modular para jugar a las **7 y media**. Se trata de un juego de cartas (con baraja española) en el que el objetivo es alcanzar una puntuación de 7.5. Cada carta del 1 al 7 tiene su valor nominal y cada figura (sota, caballo y rey) vale 0.5 puntos.

    NOTA: Para barajar, mezcle los elementos de un vector de cartas intercambiando en repetidas ocasiones cartas elegidas al azar con la ayuda de la función `Math.random()`