



UNIVERSIDAD DE ALMERÍA

Grado en Ingeniería Informática

Introducción a la Programación

2016-2017



Tema 1. Fundamentos de Programación

- Algoritmos y programas
- Introducción a la programación
- Datos y tipos de datos
- Tipos de datos primitivos en Java
- Variables
- Expresiones y sentencias
- Programas
- Estructuras de control
- Uso de subprogramas
- Métodos.

Tema 1. Fundamentos de Programación



Algoritmos y programas



- ☐ Algoritmo
- ☐ Programa
- ☐ Lenguajes de programación
- ☐ Traductores
- ☐ Herramientas de programación
- ☐ Ciclo de vida del software

Tema 1. Fundamentos de Programación

➤ ☐ Algoritmo

Secuencia ordenada de pasos que resuelve un problema concreto.

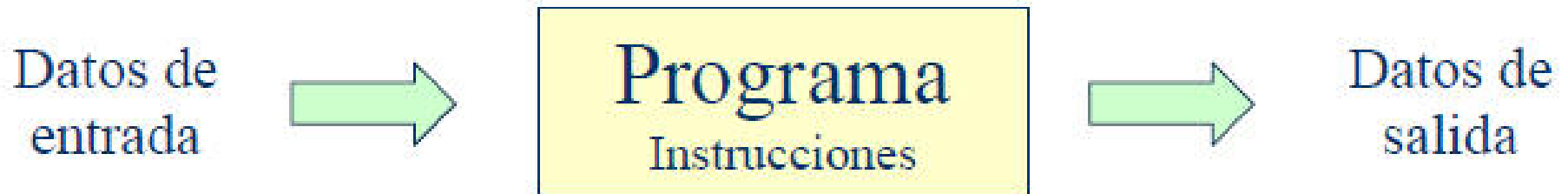
Características

- ✓ Corrección (sin errores).
- ✓ Precisión (ausencia de ambigüedades).
- ✓ Repetitividad (solución genérica de un problema dado).
- ✓ Finitud (número finito de órdenes).
- ✓ Eficiencia (temporal [tiempo necesario] y espacial [memoria utilizada])

Tema 1. Fundamentos de Programación

➤ □ Programa

Implementación de un algoritmo en un lenguaje de programación



Tema 1. Fundamentos de Programación

➤ ☐ Programa

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

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

Tema 1. Fundamentos de Programación

➤ ☐ Lenguajes de programación

Se forman con símbolos tomados de un determinado repertorio (componentes léxicos)

Se construyen siguiendo unas reglas precisas (sintaxis)

- Lenguaje máquina
- Lenguaje ensamblador
- Lenguajes de alto nivel

Tema 1. Fundamentos de Programación

➤ □ Lenguajes de programación

Clasificación de los lenguajes de programación de alto nivel

❖ Lenguajes imperativos:

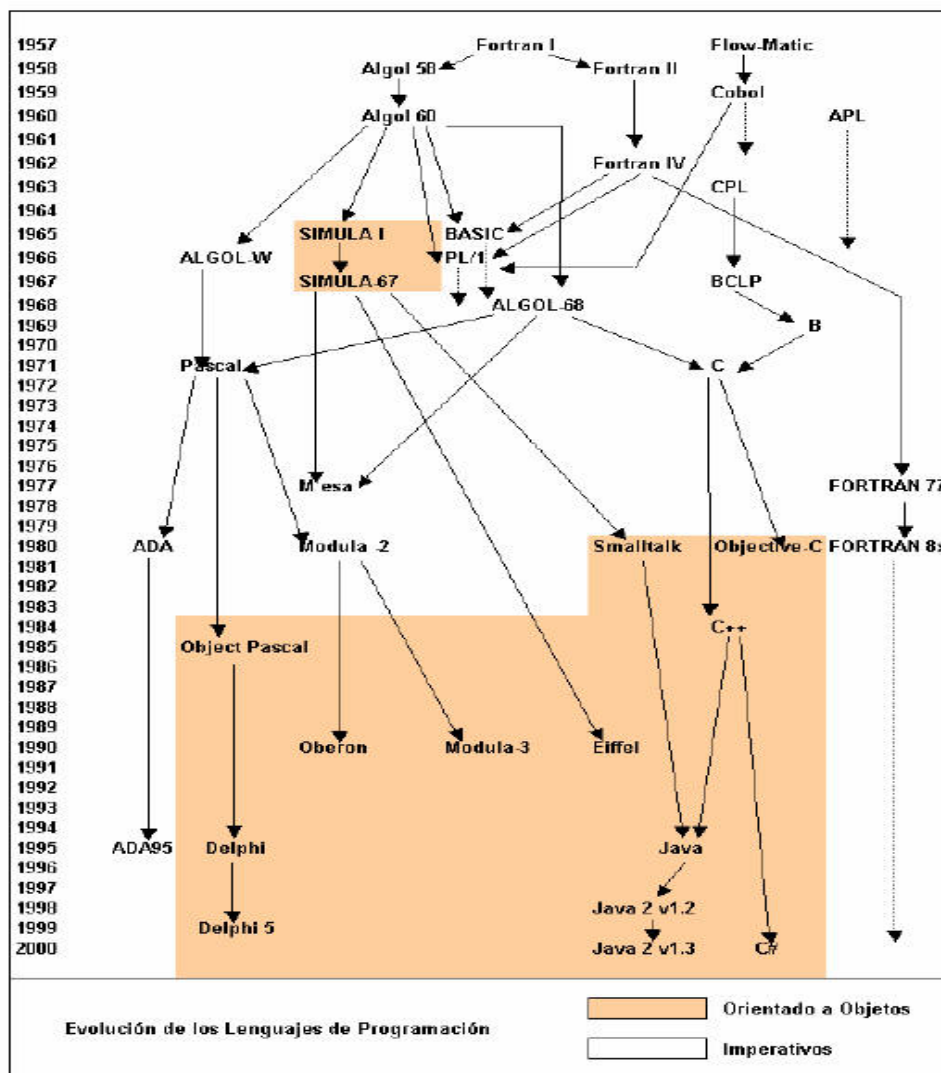
Los programas indican al ordenador de forma inequívoca los pasos a seguir para la resolución de un problema.

- Programación estructurada (C, Pascal, Fortran...)
- Programación orientada a objetos (Smalltalk, C++, Java, C#...)

❖ Lenguajes declarativos (funcionales y lógicos):

Los programas se implementan como conjuntos de funciones (lógicas) cuya evaluación nos dará el resultado deseado.

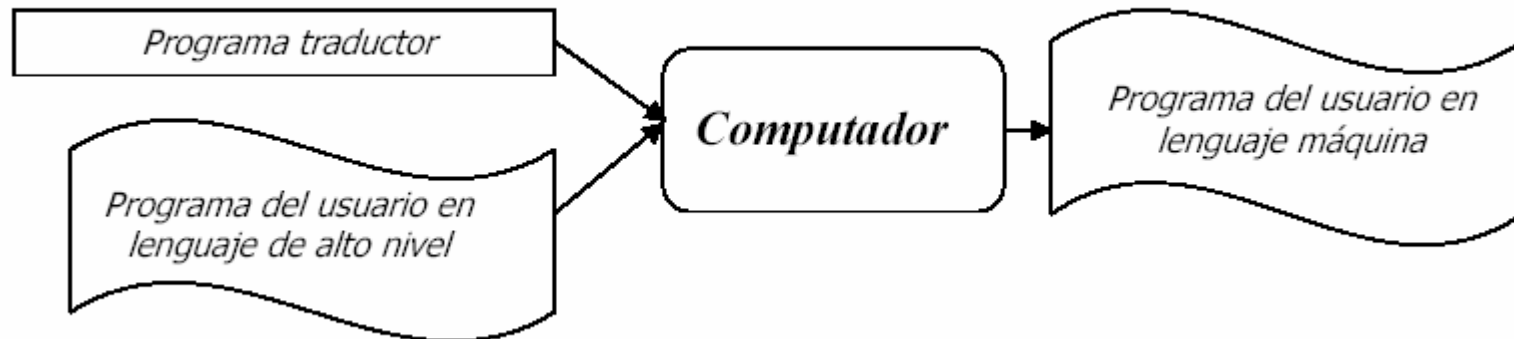
Evolución de los lenguajes de programación: Lenguajes imperativos



Tema 1. Fundamentos de Programación

➤ □ Traductores

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



Tema 1. Fundamentos de Programación

➤ Traductores

Tipos de traductores

- **Compiladores**



Generan un programa ejecutable a partir del código fuente

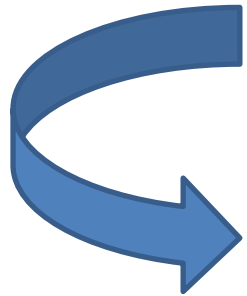
- **Intérpretes**

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.

Tema 1. Fundamentos de Programación

☐ Herramientas de programación

Editores, depuradores,...



IDEs (entornos integrados de desarrollo)

*Ejemplos : Microsoft Visual Studio .NET, Borland C++Builder/Delphi, **Eclipse**, Netbeans, etc.*

Tema 1. Fundamentos de Programación

➤ Herramientas de programación (IDE-Eclipse)

Java - TemariolIntroduccionProgramacion2016/src/org/ip/sesion01/Distancia.java - Eclipse - C:\WORKSPACES\Docencia_2016

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit

org.ip.sesion01 3

- ConvertirCelsiusFahrenheit.java 3
- Distancia.java 3**
- DivisionPorCero.java 3
- EnfriamientoViento.java 3
- EnteroAleatorio.java 3
- EntradaArgumentos.java 3
- Hola.java 3
- OperadoresRelacionales.java 3
- OperadoresUnarios.java 3

org.ip.sesion02 3

- CodigoDefectuoso.java 3
- DiaJuliano.java 3
- DiaSemana.java 3
- Ecuacion2Grado.java 3
- EcuacionRecta.java 3
- OperacionAritmetica.java 3
- TarifaTaxi.java 3

org.ip.sesion03 23

org.ip.sesion04 22

org.ip.sesion05 23

org.ip.sesion06 24

org.ip.sesion07 26

org.ip.sesion08 30

org.ip.sesion09 3

org.ip.sesion10 3

test 31

JRE System Library [JavaSE-1.7]

JUnit 4

Documentos 27

GrupoDocente 25

Distancia.java

```
16
17 public class Distancia {
18     public static void main(String[] args) {
19
20         int x = 2;
21         int y = 1;
22
23         // calculo de la distancia a (0, 0)
24         double dist = Math.sqrt(x * x + y * y);
25
26         System.out.println();
27
28         // salida de la distancia
29         System.out.println("La distancia del punto ("
30             + dist);
31     }
32 }
```

Task List

Find

Connect Mylyn

Connect to your task and ALM tools or create a local task.

Outline

Problems Javadoc Declaration Search Console History

<terminated> Distancia [Java Application] C:\ECLIPSES\GIPP2015EclipseLuna\JDKS\jdk1

La distancia del punto (2, 1) al punto (0, 0) es 2.2360679

org.ip.sesion01.Distancia.java - TemariolIntroduccionProgramacion2016/src

GIPP2016EclipseMars

eclipse MARS.1

Pregúntame cualquier cosa

11:39 24/09/2016

Tema 1. Fundamentos de Programación

➤ ☐ Ciclo de vida del software

Comprende las siguientes etapas:

- **Planificación**
- **Análisis (¿qué?):** Análisis de requisitos
- **Diseño (¿cómo?):** Estudio de alternativas
- **Implementación**
- **Depuración y pruebas**
- **Explotación: Uso y mantenimiento**

Tema 1. Fundamentos de Programación



Introducción a la programación

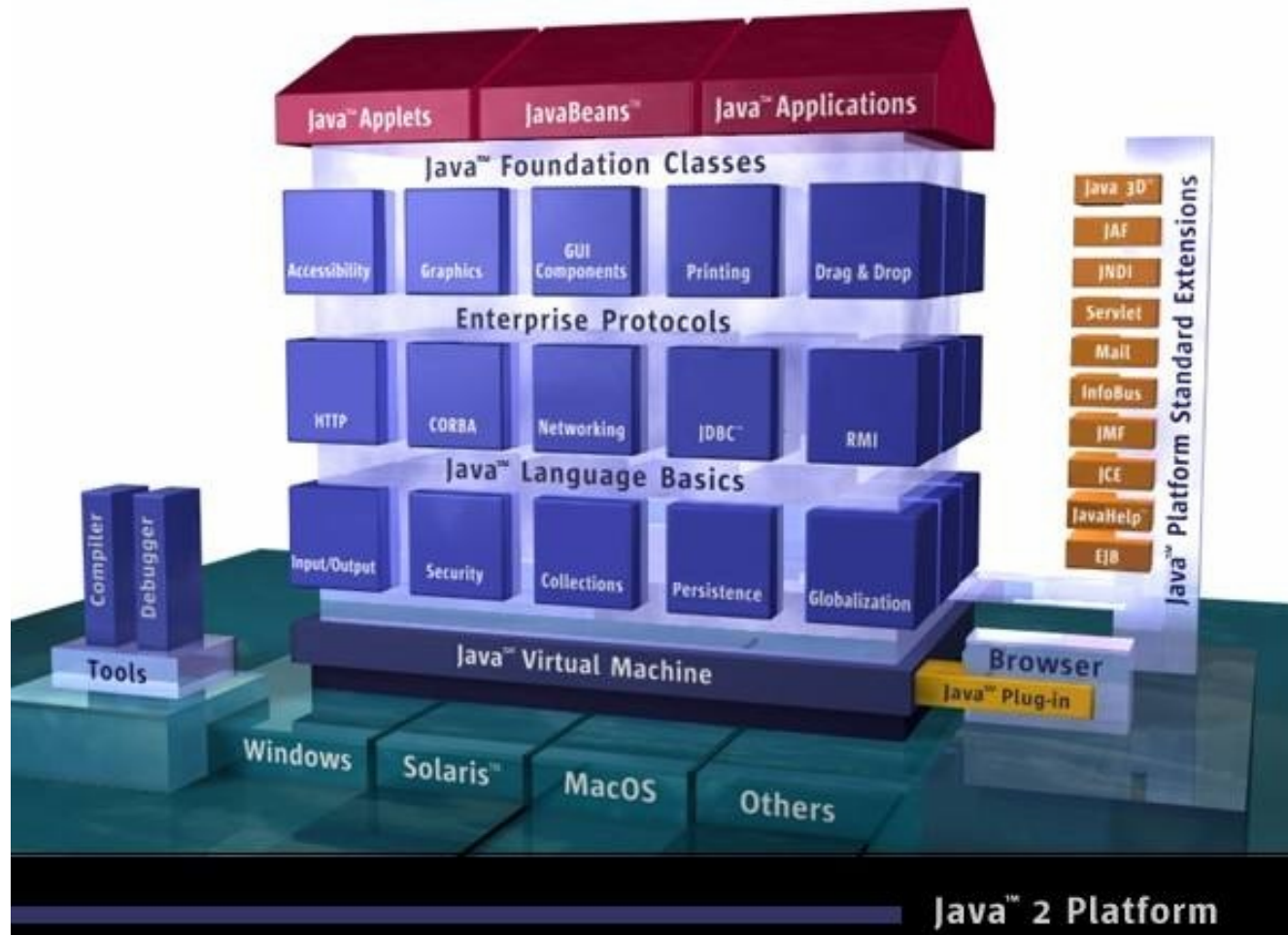


- ☐ La plataforma de programación Java
- ☐ Aplicaciones y applets
- ☐ Características de Java

Tema 1. Fundamentos de Programación



□ La plataforma de programación Java



Tema 1. Fundamentos de Programación



□ La plataforma de programación 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.

Tema 1. Fundamentos de Programación

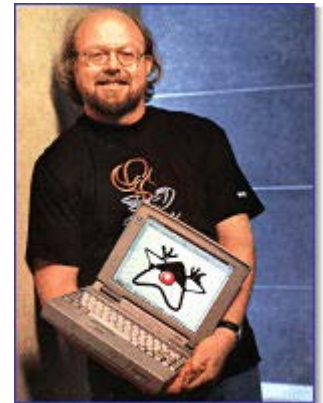
➤ □ La plataforma de programación Java

Historia de Java

La versión más difundida sobre el origen es la que presenta a **Java** como un lenguaje pensado para pequeños electrodomésticos. Sun Microsystems decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos.



James Gosling



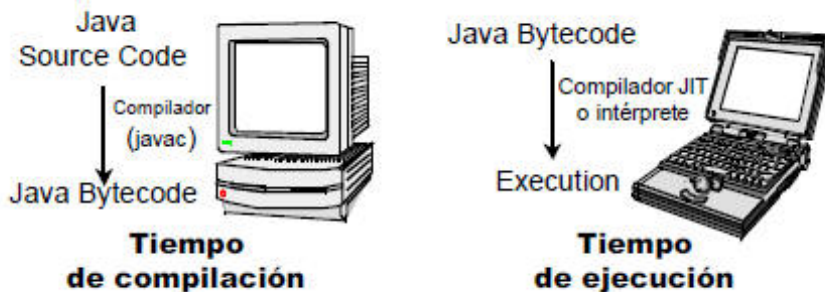
Tema 1. Fundamentos de Programación

➤ □ La plataforma de programación Java



La máquina virtual de 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]).



IMP!!

Information Module Profile



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.

Tema 1. Fundamentos de Programación



□ La plataforma de programación Java



Uso típico de Java

- Aplicaciones (programas independientes)
- Applets (“pequeñas aplicaciones”) son programas diseñados para ejecutarse como parte de una página web.

Herramientas de programación en Java

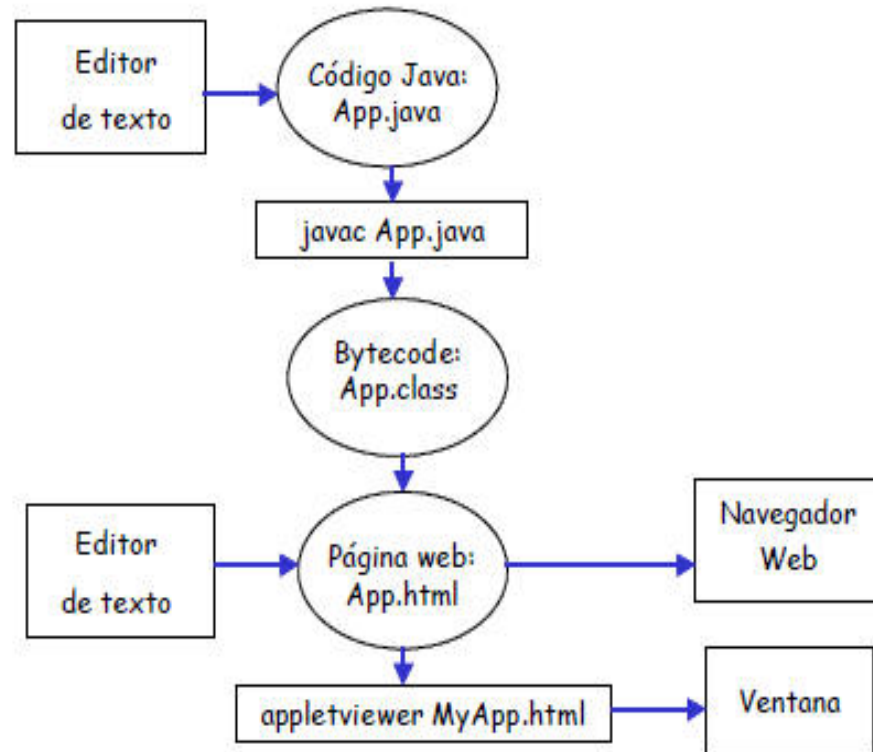
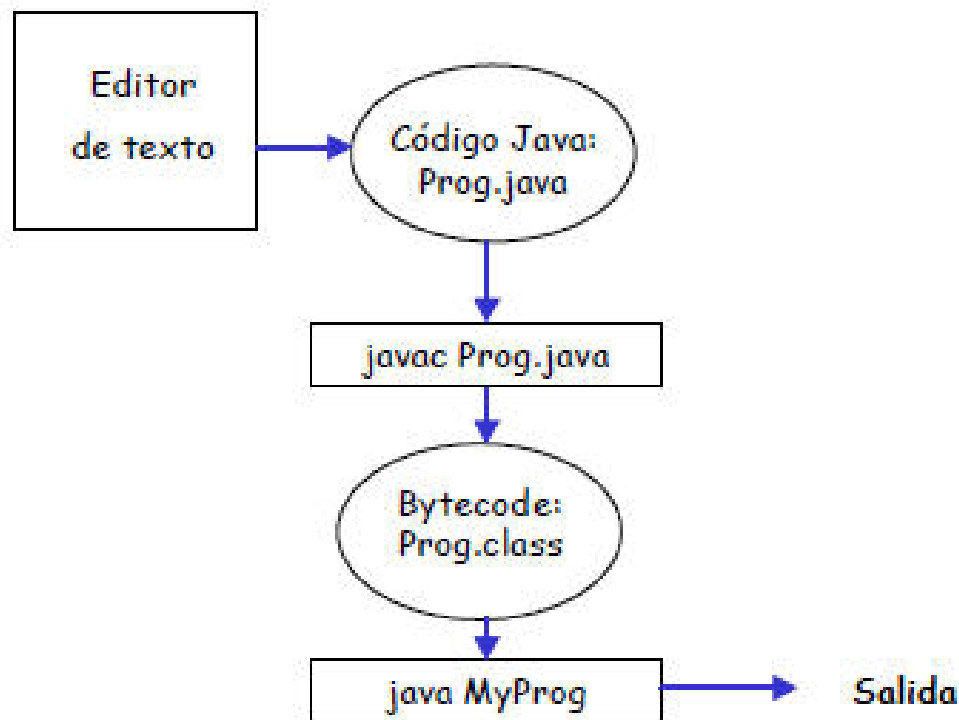
- **Java SDK [Software Development Kit]** <https://java.com>
 - ✓ Compilación de aplicaciones Java: javac
 - ✓ Ejecución de aplicaciones Java: java
 - ✓ Ejecución de applets: appletviewer
 - ✓ Generación de documentación: javadoc
 - ✓ Creación de archivos de distribución JAR [Java ARchives]: jar
 - ✓ Depuración de aplicaciones Java: jdb
 - ✓ etc.
- **Entornos integrados de desarrollo: IDEs** → **Eclipse** (<https://eclipse.org>) y **NetBeans** (<https://netbeans.org>)

Tema 1. Fundamentos de Programación

➤ □ Aplicaciones y applets

Creación y ejecución de aplicaciones Java

Creación y ejecución de applets



Tema 1. Fundamentos de Programación



□ Aplicaciones y applets

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).

Tema 1. Fundamentos de Programación

➤ □ Aplicaciones y applets

Características clave de Java

- Java es multiplataforma
- Java es seguro
- Java tiene un amplio conjunto de bibliotecas estándar
- Java incluye una biblioteca portable para la creación de interfaces gráficas de usuario (AWT y JFC/Swing)
- Java simplifica algunos aspectos a la hora de programar

Tema 1. Fundamentos de Programación



Datos y tipos de datos

- 
- ☐ Dato
 - ☐ Tipo de dato
 - ☐ Codificación de los datos en el ordenador

Tema 1. Fundamentos de Programación

➤ ☐ 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.

Tema 1. Fundamentos de Programación

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

Tema 1. Fundamentos de Programación

➤ ☐ Codificación de los datos en el ordenador

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

Un bit nos permite representar 2 símbolos diferentes: 0 y 1

En general,

| 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 |

Tema 1. Fundamentos de Programación

➤ ☐ Codificación de los datos en el ordenador

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.

!!! $2\,147\,483\,647 + 1 = -2\,147\,483\,648$!!!

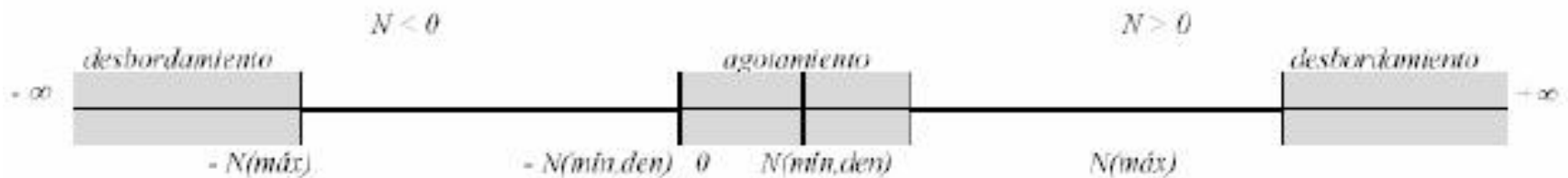
Tema 1. Fundamentos de Programación

➤ □ Codificación de los datos en el ordenador

NÚMEROS REALES (en notación científica)

(+|-) mantisa $\times 2^{\text{exponente}}$

- 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.



Tema 1. Fundamentos de Programación

➤ □ Codificación de los datos en el ordenador

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.

Tema 1. Fundamentos de Programación

➤ Codificación de los datos en el ordenador

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 | 10 | 11 | 12 | 13 | 14 | 15 |
|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|-----|-----|----|----|----|-----|
| 00 | 0 | NUL | SOH | STX | ETX | EOT | ENO | 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 | é | ê | ë | ì | í | î | ï | |
| F0 | 240 | ð | ñ | ò | ó | ô | õ | ö | ÷ | ø | ù | ú | û | ü | ý | þ | ÿ |

Tema 1. Fundamentos de Programación

➤ □ Codificación de los datos en el ordenador


Ejemplos de códigos normalizados

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

| <i>Zona</i> | <i>Códigos</i> | | <i>Símbolos codificados</i> | <i>Nº de caracteres</i> |
|-------------|----------------|--------------|---|-------------------------|
| A | 0000 | 0000 00FF | Latin-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 |

Tema 1. Fundamentos de Programación

Tipos de datos primitivos en Java

-  ☐ Números enteros
- ☐ Números en coma flotante
- ☐ Caracteres
- ☐ Cadenas de caracteres
- ☐ Booleanos

Tema 1. Fundamentos de Programación

Tipos de datos primitivos en Java

El lenguaje Java define 8 tipos de datos primitivos:

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**

Tema 1. Fundamentos de Programación

➤ □ Números enteros

| 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 |

Operaciones con números enteros

Desbordamiento

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

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

Tema 1. Fundamentos de Programación

➤ ☐ Números enteros

División por cero

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

Exception in thread "main"

java.lang.ArithmeticException: / by zero

at ...

Tema 1. Fundamentos de Programación

➤ ☐ Números en coma flotante

| 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 |

Operaciones con números en coma flotante

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



Not a Number

Tema 1. Fundamentos de Programación

Operadores aritméticos

| 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)**

Tema 1. Fundamentos de Programación

Operadores aritméticos

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 |

Módulo (%): Resto de dividir

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

Tema 1. Fundamentos de Programación

Expresiones aritméticas

Expresión: Construcción que se evalúa para devolver un valor. Se pueden combinar literales y operadores para formar expresiones complejas. Por ejemplo,

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

En Java se escribiría:

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

Tema 1. Fundamentos de Programación



❑ Caracteres

| 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' ... '*' ...

Secuencias de escape para representar caracteres especiales

La clase **Character** define funciones (métodos estáticos)

para trabajar con caracteres:

isDigit(), **isLetter()**, **isLowerCase()**, **isUpperCase()**

toLowerCase(), **toUpperCase()**

Tema 1. Fundamentos de Programación



□ Caracteres

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

Tema 1. Fundamentos de Programación

➤ □ 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”

Tema 1. Fundamentos de Programación



□ Cadenas de caracteres

Concatenación de cadenas de caracteres

El operador + sirve para concatenar cadenas de caracteres

| Operación | Resultado |
|---------------------------------|-------------------------|
| <code>"Total = " + 3 + 4</code> | <code>Total = 34</code> |
| <code>"Total = " + (3+4)</code> | <code>Total = 7</code> |

Tema 1. Fundamentos de Programación

➤ □ Booleanos

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

| Espacio en memoria | | Valores |
|----------------------|-------|-------------------|
| <code>boolean</code> | 1 bit | Verdadero o falso |

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**.

Tema 1. Fundamentos de Programación

Operadores relacionales

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

Operadores lógicos/booleanos

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

| X | !X |
|-------|-------|
| true | false |
| False | true |

Tablas de verdad



| 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 |

Tema 1. Fundamentos de Programación



Variables

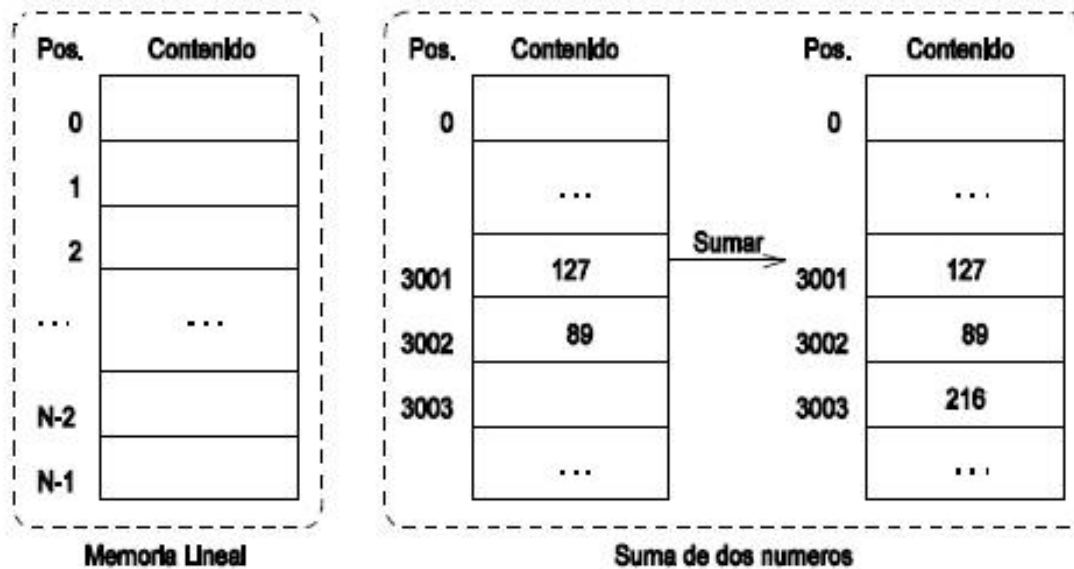


☐ Declaración de variables

☐ Definición de constantes

Tema 1. Fundamentos de Programación

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”

Tema 1. Fundamentos de Programación

➤ □ Declaración de variables

Para usar una variable en un programa debe de estar previamente declarada.

```
<tipo> identificador;  
<tipo> lista de identificadores;
```

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;
```

Tema 1. Fundamentos de Programación

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:

➤ Las mayúsculas y las minúsculas se consideran diferentes.

| | | | | |
|----------|----------|------------|-----------|--------------|
| 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 | |

Tema 1. Fundamentos de Programación

Identificadores en Java

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.

Tema 1. Fundamentos de Programación

Identificadores en Java

Convenciones

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

Inicialización de variables

```
int i = 0;  
float pi = 3.1415927f;  
double x = 1.0, y = 1.0;
```

Tema 1. Fundamentos de Programación

➤ Definición de constantes

```
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.
- Si intentamos cambiar el valor a una cte, se produce un error en tiempo de compilación.



```
final double CARGA_ELECTRON = 1.6E-19;  
CARGA_ELECTRON = 1.7E-19;
```



The final local variable CARGA_ELECTRON cannot be assigned.

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

Tema 1. Fundamentos de Programación



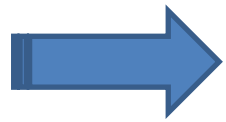
Expresiones y sentencias



☐ Construcción de expresiones

☐ Sentencia de asignación

Tema 1. Fundamentos de Programación



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 [;].

Tema 1. Fundamentos de Programación

➤ Construcción de expresiones

➤ Literales y variables son expresiones primarias:

1.7 // Literal real de tipo double. Un literal es la
 // especificación de un valor concreto de un tipo dado
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:

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

Tema 1. Fundamentos de Programación

➤ Construcción de expresiones

Tipo del resultados

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

Tema 1. Fundamentos de Programación

➤ ☐ Sentencia de asignación

Sintaxis:

```
<variable> = <expresión>;
```

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.

Tema 1. Fundamentos de Programación

➤ Sentencia de asignación

Ejemplos

```
x = x + 1;
```

```
int miVariable = 20;
```

```
otraVariable = miVariable;
```

Tipo del resultados

```
// Declaración con inicialización
```

```
// Sentencia de asignación
```

Tema 1. Fundamentos de Programación

Conversión de tipos

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

- La conversión de un tipo con menos bits a un tipo con más bits es automática (por ejemplo, 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

Tema 1. Fundamentos de Programación

Conversión de tipos

Ejemplos

```
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
```

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)*

Tema 1. Fundamentos de Programación

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 $+$

| Prioridad de operaciones o precedencia |
|--|
| () |
| ++, --, ! |
| *, /, % |
| +, - |
| <, <=, >, >= |
| =, != |
| && |
| |
| =, +=, -=, *=, /=, %= |

Operadores incremento y decremento. Operadores unarios

Java ofrece una notación abreviada para una operación de programación muy común, la de incrementar o decrementar el valor de una variable en 1. Los operadores de incremento, decremento son:

| Operador | Significado |
|----------|--|
| $x++$ | Primero evalúa y después incrementa x (post-incremento) |
| $++x$ | Incrementa x y luego evalúa (pre-incremento) |
| $x--$ | Primero evalúa y después decrementa x (post-decremento) |
| $--x$ | Decrementa x y después evalúa (pre-decremento) |

Ejemplos

Suponiendo $i = 3$ y $c = 10$, determine los resultados de las siguientes operaciones:

A. $x = i++ = 3++ \Rightarrow$ incrementa después de la asignación, así $x = 3$

B. $x = ++i = ++3 \Rightarrow$ incrementa antes de la asignación, así $x = 4$

C. $x = c++ = 10++ \Rightarrow$ incrementa después de la asignación, así $x = 10$

D. $x = --c + 2 = --10 + 2 = 9 + 2 = 11$

E. $x = i-- + ++c = 3-- + ++10 = 3 + 11 = 14$

Operadores combinados de asignación (op=)

| Operador | Nombre | Ejemplo | Equivale |
|----------|---------------------------|---------------------|------------------------|
| += | Asignación adición | <code>i += 8</code> | <code>i = i + 8</code> |
| -= | Asignación sustracción | <code>i -= 8</code> | <code>i = i - 8</code> |
| *= | Asignación multiplicación | <code>i *= 8</code> | <code>i = i * 8</code> |
| /= | Asignación división | <code>i /= 8</code> | <code>i = i / 8</code> |
| %= | Asignación resto | <code>i %= 8</code> | <code>i = i % 8</code> |

Tema 1. Fundamentos de Programación



Programas



- ☐ Estructura de un programa simple
- ☐ Estilo y documentación del código
- ☐ Errores de programación

Tema 1. Fundamentos de Programación



□ Estructura de un programa simple

- 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

```
}
```

Tema 1. Fundamentos de Programación

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 primer programa realizado

```
package org.ip.sesion01;  
public class HolaMundo {  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("HOLA MUNDO");  
    }  
}
```

Tema 1. Fundamentos de Programación

- ❑ La máquina virtual Java (JVM, Java Virtual Machine) ejecuta el programa invocando a la función main.
- ❑ Las llaves {} delimitan bloques en Java (conjuntos de elementos de un programa, delimitadores de ámbito).

Comentarios

Los comentarios sirven para incluir aclaraciones en el código.

Java permite tres tipos de comentarios:

// Comentarios de una línea

/* Comentarios de varias líneas */

/** Comentarios de documentación */ Son comentarios al estilo javadoc

Tema 1. Fundamentos de Programación

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 (por ejemplo, división por cero) o que el programa proporcione resultados incorrectos.

Errores lógicos

Los causados por un mal diseño del algoritmo.

Tema 1. Fundamentos de Programación

Hasta ahora nuestros programas se construirán:

```
package org.ip.sesionNN;
```

```
public class IdentificadorClase{
```

```
    /**
```

```
    * @param args
```

```
    */
```

```
    public static void main(String[] args) {
```

```
        declaración de variables y/o constantes
```

```
        sentencias (asignación, salida)
```

```
    }
```

```
}
```

Tema 1. Fundamentos de Programación

Ejemplo 1

```
package org.ip.sesion01;

/*****
 * Muestra lo que ocurre cuando divides por cero con enteros y reales
 *
 * 17.0 / 0.0 = Infinity
 * 17.0 % 0.0 = NaN
 * Exception in thread "main" java.lang.ArithmeticException: / by zero
 *
 *****/

public class DivisionPorCero {
    public static void main(String[] args) {
        System.out.println("17.0 / 0.0 = " + (17.0 / 0.0));    // Infinity
        System.out.println("17.0 % 0.0 = " + (17.0 % 0.0));    // NaN => not a number
        System.out.println("17 / 0 = " + (17 / 0));            // ERROR
        System.out.println("17 % 0 = " + (17 % 0));            // ERROR
    }
}
```

Tema 1. Fundamentos de Programación

Ejemplo 2

```
package org.ip.sesion01;
```

```
/*  
*****
```

- * Muestra un entero pseudo-aleatorio entre 0 y N - 1.
- * Ilustra una conversión explícita de tipos (cast) de double a int.
- *

```
*****  
*/
```

```
public class EnteroAleatorio {
```

```
    public static void main(String[] args) {
```

```
        int N = 10;
```

```
  
        // genera un real pseudo-aleatorio entre 0.0 1.0
```

```
        double r = Math.random();
```

```
  
        // lo convertiremos a un entero pseudo-aleatorio entre 0 y N-1
```

```
        int n = (int) (r * N);
```

```
  
        System.out.println("Su entero aleatorio es: " + n);
```

```
    }
```

```
}
```


Salida a consola

Java utiliza **System.out** para referirse al dispositivo *estándar de salida* y **System.in** para referirse al dispositivo *estándar de entrada*. Por defecto el dispositivo de salida estándar es el monitor y el de entrada es el teclado.

La clase **System** la podemos usar sin importarla porque esta en el paquete **java.lang** que es el único que se importa automáticamente y por tanto no es necesario indicarlo.

Para realizar una salida a consola, utilizaremos algunos de los siguientes métodos:

- **print**
- **println**
- **printf**

El método **print** es idéntico a **println** excepto que este último mueve el cursor a la siguiente línea después de mostrar el string y **print** no avanza el cursor a la línea siguiente.

Salida a consola formateada

En ocasiones estamos interesados en mostrar un número real con solo dos decimales o un número entero justificado a la derecha o a la izquierda etc. Para ello utilizamos **printf** cuya sintaxis es:

```
System.out.printf(especificadores de formato, item1, item2, ...);
```

Los *especificadores de formato* se dan entre comillas dobles e indican como se quiere mostrar la salida. Consisten en un signo de porcentaje (%) seguido de un carácter cuyo significado se muestra en la tabla.

Tabla de especificadores más frecuentes

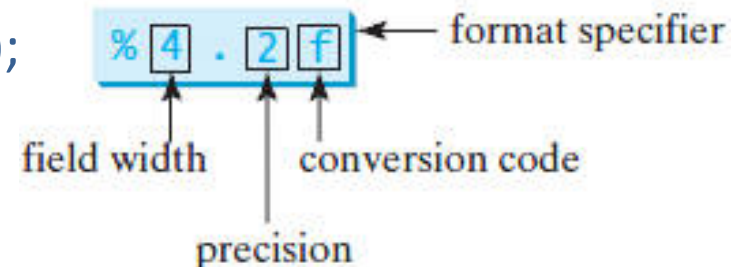
| Especificador | Salida | Ejemplo |
|---------------|-------------------|----------------|
| %b | Un valor booleano | true o false |
| %c | Un carácter | 'a' |
| %d | Un entero | 200 |
| %f | Un real | 45.460000 |
| %s | Una cadena | "Esto es Java" |

Los items pueden ser un valor numérico, un carácter, un booleano o una cadena.

Ejemplo: `double x = 2.0 / 3;`

`System.out.printf("x es %4.2f ",x);`

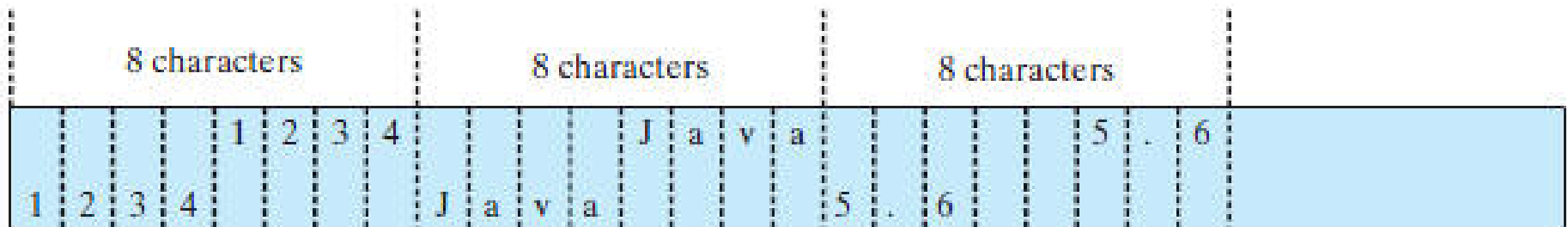
Mostraría `x es 0.67`



Por defecto, la salida está justificada a la derecha. Podemos poner el signo (-) para especificar que un *item* se quiere justificar a la izquierda. Por ejemplo las siguientes sentencias:

```
System.out.printf("%8d%8s%8.1f\n", 1234, "Java", 5.6);
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.6);
```

Mostrarían



Tema 1. Fundamentos de Programación



Estructuras de control



- ☐ Programación estructurada
- ☐ Estructuras selectivas/condicionales
- ☐ Estructuras repetitivas/iterativas

Tema 1. Fundamentos de Programación

➤ ☐ Programación estructurada

☐ Crisis del software (Dijkstra)

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

☐ IDEA PRINCIPAL:

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

Tema 1. Fundamentos de Programación

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).

En programación estructurada sólo se emplean tres construcciones:

☐ Secuencia

Conjunto de sentencias que se ejecutan en orden

Ejemplos: Sentencias de asignación.

☐ Selección

Elige qué sentencias se ejecutan en función de una condición.

Ejemplos: Estructuras de control condicional **if .. else** y **switch**

☐ Iteración

Las estructuras de control repetitivas repiten conjuntos de instrucciones.

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

Tema 1. Fundamentos de Programación

Estructuras de control selectivas

Por defecto,

las instrucciones de un programa se ejecutan **secuencialmente**

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.

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

Tema 1. Fundamentos de Programación

Sintaxis Selectiva Simple

```
if (condición)
    sentencia;
if (condición) {
    bloque
}
```

donde ***bloque*** representa un bloque de instrucciones.

Bloque de instrucciones:

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

Tema 1. Fundamentos de Programación

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  
}
```

Tema 1. Fundamentos de Programación

➤ 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!!!

Tema 1. Fundamentos de Programación

Ejemplos de uso selectiva simple

```
if (nota >= 5)
```

```
    System.out.println("APROBADO");
```

```
if ( nota >= 5) {
```

```
    System.out.println("APROBADO");
```

```
    System.out.println("Recupera todo");
```

```
}
```

Tema 1. Fundamentos de Programación

Sintaxis Selectiva Doble, Cláusula else

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

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

Tema 1. Fundamentos de Programación

Ejemplos de uso selectiva doble

```
if (nota >= 5)
    System.out.println("APROBADO");
else
    System.out.println("SUSPENSO");

if ( nota >= 5) {
    System.out.println("APROBADO");
    System.out.println("Recupera todo");

} else {
    System.out.println("SUSPENSO");
    System.out.println("No recupera");
}
```

Tema 1. Fundamentos de Programación

Ejemplos de uso selectiva con operadores relacionales

```
if (nota < 0) || (nota > 10)
    System.out.println("NOTA NO VALIDA");
else
    System.out.println("VALIDA");

if (nota >= 0) && (nota <= 10) {
    System.out.println("VALIDA");
} else {
    System.out.println("NOTA NO VALIDA");
}
```

Tema 1. Fundamentos de Programación

Sintaxis Selectiva Múltiple **if ... else if ...**

```
if (condición1)
    sentencia1;
else if (condición2)
    sentencia2;

.....

else
    sentenciaN;
```

Idem con *bloques*

Tema 1. Fundamentos de Programación

Ejemplo de uso selectiva múltiple

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

Se puede expresar con cuatro selectivas simple

Tema 1. Fundamentos de Programación

Sintaxis de Anidamiento de Selectivas Doble

```
if (condición1) {  
    sentencia1;  
} else {  
    if (condición2) {  
        sentencia2;  
    } else {  
        sentencia3;  
    }  
}
```

Tema 1. Fundamentos de Programación

Ejemplo de uso selectiva anidada

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

Se puede expresar con tres selectivas simple

Tema 1. Fundamentos de Programación

Sintaxis del operador condicional ? :

Java proporciona una forma de abreviar una sentencia **if**, el operador condicional ? :

variable = condición ? expresión1 : expresión2;

“equivale” a

if (condición)

variable = expresión1;

else

variable = expresión2;

Tema 1. Fundamentos de Programación

Ejemplo de uso del operador condicional '? :'

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

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

Tema 1. Fundamentos de Programación

Sintaxis Selectiva Múltiple con la sentencia 'switch'

```
switch (expresión) {  
    case expr_cte1:  
        bloque1;  
        break;  
    case expr_cte2:  
        bloque2;  
        break;  
    ...  
    case expr_cteN:  
        bloqueN;  
        break;  
    default:  
        bloque_por_defecto;  
}
```

Tema 1. Fundamentos de Programación

Consideraciones al respecto

- ☐ Permite seleccionar entre varias alternativas posibles
- ☐ Se selecciona a partir de la evaluación de una única expresión.
- ☐ La expresión del **switch** puede ser de tipo: byte, short, char, int, String, ...
- ☐ 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**.

Tema 1. Fundamentos de Programación

```
switch (nota) {  
    case 0:  
    case 1:  
    case 2:  
    case 3:  
    case 4:  
        resultado = "Suspenseo";  
        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";  
}
```


Tema 1. Fundamentos de Programación

```
int numeroDia=0;  
String dia = "jueves";
```

```
switch (dia) {  
    case "lunes":  
        numeroDia=1;  
        break;  
    case "martes":  
        numeroDia=2;  
        break;  
    case "miercoles":  
        numeroDia=3;  
        break;  
    case "jueves":  
        numeroDia=4;  
        break;  
    case "viernes":  
        numeroDia=5;  
        break;  
    case "sabado":  
        numeroDia=6;  
        break;  
    case "domingo":  
        numeroDia=7;  
        break;  
    default:  
        numeroDia=0;  
}
```

Tema 1. Fundamentos de Programación

➤ □ Estructuras de control repetitivas / iterativas / bucles

Permiten repetir una acción o conjunto de acciones un determinado número de veces. En ocasiones, ese número es conocido de antemano, pero en otras ocasiones, no.

En un bucle hay que tener en cuenta dos aspectos:

- **La condición de terminación**, me indica cuando dejo de ejecutar el bucle. Coloquialmente se dice que es la condición que me permite escapar del bucle.
- **Cuerpo del bucle**, es la acción o conjunto de acciones que se repiten.

Tema 1. Fundamentos de Programación

Vamos a presentar tres estructuras repetitivas

*El bucle **while***

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

Sintaxis **while** (condición de continuación del bucle)
 sentencia; //Cuerpo del bucle

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

Tema 1. Fundamentos de Programación

En un bucle, se evalúa la condición, si es verdadera, se ejecutan la sentencia o el bloque. Se volvería a evaluar la condición, de ser verdadera, nuevamente se ejecutan las sentencias del cuerpo del bucle. En el momento en el que la condición sea falsa, se dejan de ejecutar las sentencias y se escapa del bucle. Cada repetición de las instrucciones de un bucle es una **iteración**.

La ventaja que tiene esta estructura es que comprobada la condición por primera vez, si es falsa, nunca se llega a ejecutar el bucle.

También cabe la posibilidad de que si la condición no está bien construida, nunca se haga falsa con lo cual tendríamos los *temidos bucles infinitos*.

Tema 1. Fundamentos de Programación

Ejemplo. Ejemplo de un fragmento de código donde se utiliza una estructura repetitiva **while**

```
.....  
int i = 1;  
while (i <= 5) {  
    System.out.println("Iteración número "+i);  
    i++;  
}  
.....
```

Tema 1. Fundamentos de Programación

Seguimiento Manual

| Iteración | i | Salida a pantalla |
|-----------|---|----------------------|
| 0 | 1 | ----- |
| 1 | 2 | Iteración número...1 |
| 2 | 3 | Iteración número...2 |
| ⋮ | ⋮ | ⋮ |
| 5 | 6 | Iteración número...5 |

Tema 1. Fundamentos de Programación

Ejemplo. Bucle que no se ejecuta nunca

```
.....  
int i = 7;  
while (i <= 5) {  
    System.out.println("Iteración número "+i);  
    i++;  
}  
.....
```

Tema 1. Fundamentos de Programación

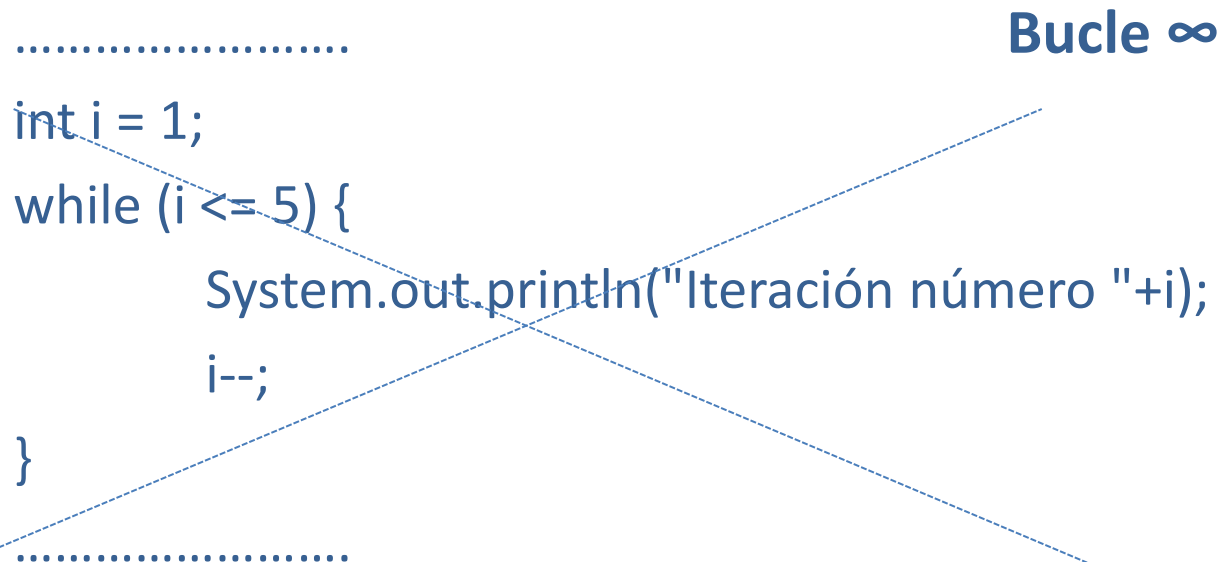
Ejemplo. Bucle infinito

.....

Bucle ∞

```
int i = 1;  
while (i <= 5) {  
    System.out.println("Iteración número "+i);  
    i--;  
}
```

.....



Tema 1. Fundamentos de Programación

En el bucle son característicos dos elementos: contadores y sumadores. A continuación se describen con un ejemplo.

.....

```
int i = 1;
```

```
suma = 0;
```

```
dato = 2;
```

```
while (i <= 5) {
```

```
    suma = suma + dato;
```

```
    dato = dato + 2;
```

```
    i = i + 1;
```

```
}
```

.....

```
// suma += dato;
```

```
// dato += 2;
```

```
// i++;
```

Contador: variable que se incrementa/decrementa en una cantidad fija en cada iteración o paso.

Sumador: variable que se incrementa/decrementa en una cantidad variable en cada iteración o paso.

Tema 1. Fundamentos de Programación

*El bucle **for***

Ayuda a simplificar la escritura de un bucle especialmente cuando conocemos de antemano el número de iteraciones del bucle.

Sintaxis

```
for (expr1; expr2; expr3) {  
    bloque; // Cuerpo del bucle  
}
```

Tema 1. Fundamentos de Programación

Ejemplo. Ejemplo de un fragmento de código donde se utiliza una estructura repetitiva **for**

```
.....  
for (int i = 1; i <= 5; i++) {  
    System.out.println("Iteración número "+i);  
}
```

.....

Salida

Iteración número 1

.....

Iteración número 5

Tema 1. Fundamentos de Programación

Esta estructura está especialmente indicada para recorridos de vectores y matrices que veremos más adelante.

Respecto al **funcionamiento** de la estructura **for**, está gobernada por una variable de control que se inicializa a un valor inicial (**expr1**) y dicha variable se va incrementando o decrementando en cada paso o iteración en una cantidad fija que se indica en **expr3** hasta que la variable alcanza el valor final que se expresa en **expr2** . Existe también la posibilidad de que las sentencias que componen el cuerpo del bucle no se ejecuten ninguna vez. Por ejemplo:

```
for (int i = 7; i <= 5; i++) {  
    System.out.println("Iteración número "+i);  
}
```

Tema 1. Fundamentos de Programación

Bucles con **decremento** de la variable de control en cada iteración

```
int n = 5;  
for (int i = n; i >= 1; i--) {  
    System.out.println("Iteración número "+i);  
}
```

Salida

Iteración número 5

.....

Iteración número 1

Tema 1. Fundamentos de Programación

Equivalencia entre for y while

Un fragmento de código como el que sigue con un bucle **while**:

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

puede **abreviarse** si utilizamos un bucle **for**:

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

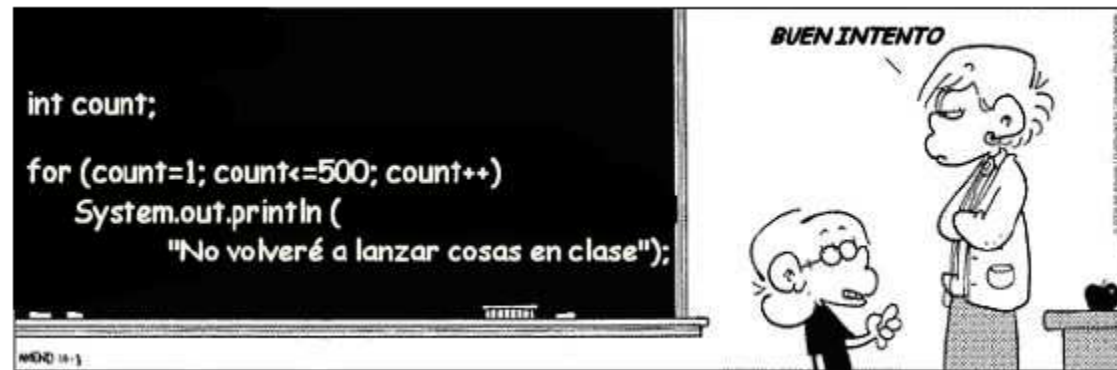
Tema 1. Fundamentos de Programación

En general,

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

equivale a

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



Tema 1. Fundamentos de Programación

Equivalencia entre for y while

Un fragmento de código como el que sigue con un bucle **while**:

```
while (dato != 0) {  
    //Cuerpo del bucle  
}
```

```
while (condición de continuidad) {  
    //Cuerpo del bucle  
}
```

puede expresarse si utilizamos un bucle **for**:

```
for (; dato != 0; ) {  
    //Cuerpo del bucle  
}
```

```
for (; condición de continuidad; ) {  
    //Cuerpo del bucle  
}
```


Tema 1. Fundamentos de Programación

*El bucle **do-while***

Es un 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);
```

Tema 1. Fundamentos de Programación

- 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).

Por ejemplo:

```
do {  
    System.out.println("Introduce un valor numérico para el mes");  
    mes = entrada.nextInt();  
} while (mes < 1 || mes > 12);
```

¿Qué bucle utilizar?

Los bucles **while** y **for** se llaman bucles *pre-test* porque la condición de continuación se comprueba antes de ejecutar el bucle. El bucle **do – while** se llama *post-test* porque la condición se comprueba después de que el bucle se ha ejecutado. Las tres sentencias de bucles: **for**, **while** y **do-while** son EQUIVALENTES es decir podemos escribir un bucle con cualquiera de ellas, con la que estemos más cómodos pero se suele:

1. Si se conocen de antemano el nº de iteraciones => **for**
2. Si no se conoce dicho número => **while**
3. Para validar datos => **do-while**

Bucles anidados

Se trata de uno o varios bucles contenidos en otros, es decir, tendremos un bucle cuyo cuerpo contenga al menos una sentencia que sea a su vez otro bucle y este a su vez puede tener otra sentencia que sea un bucle.

Tema 1. Fundamentos de Programación

Por ejemplo:

```
int N = 3;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        System.out.print("(" + i + "," + j + ")");
    }
    System.out.println();
}
```

| | | | |
|---------------|-------|-------|-------|
| Salida | (0,0) | (0,1) | (0,2) |
| | (1,0) | (1,1) | (1,2) |
| | (2,0) | (2,1) | (2,2) |

Tema 1. Fundamentos de Programación

Diseño de un bucle

A partir del enunciado de un problema, ¿cómo puedo construir correctamente un bucle?. En primer lugar optaremos por una estructura repetitiva siempre que haya proceso que se repite. A continuación, puede ser de ayuda responder a las preguntas:

- ¿Cuál es la condición de terminación del bucle?
- ¿Cómo se inicializa y actualiza la condición?
- ¿Cuál es el proceso que se repite?
- ¿Cómo se inicializa y actualiza el proceso?

Tema 1. Fundamentos de Programación

Uso de subprogramas

Los lenguajes de programación permiten descomponer un programa complejo en distintos subprogramas:

- **Funciones y procedimientos** en lenguajes de programación estructurada
- **Métodos** en lenguajes de programación orientada a objetos

Entre otras, las razones para crear un subprograma son:

- Reducir la complejidad del programa (“divide y vencerás”).
- Eliminar código duplicado.
- Mejorar la legibilidad del código.
- Promover la reutilización de código

Tema 1. Fundamentos de Programación

Supongamos que necesitamos obtener la suma de enteros:
de 1 a 10, de 20 a 30, de 35 a 45 respectivamente. Podríamos escribir el
fragmento de código que sigue.

```
int suma = 0;
for (int i = 1; i <= 10; i++)
    suma += i;
System.out.println("La suma de 1 a 10 es " + suma);
suma = 0;
for (int i = 20; i <= 30; i++)
    suma += i;
System.out.println("La suma de 20 a 30 es " + suma);
suma = 0;
for (int i = 35; i <= 45; i++)
    suma += i;
System.out.println("La suma de 35 a 45 es " + suma);
```

Tema 1. Fundamentos de Programación

Podemos observar que para obtener las sumas anteriores el código es muy parecido excepto que el comienzo y el final de los números enteros es diferente. Mediante la definición de un **método** y la invocación del mismo podemos evitar escribir tres veces casi el mismo fragmento de código.

Tema 1. Fundamentos de Programación



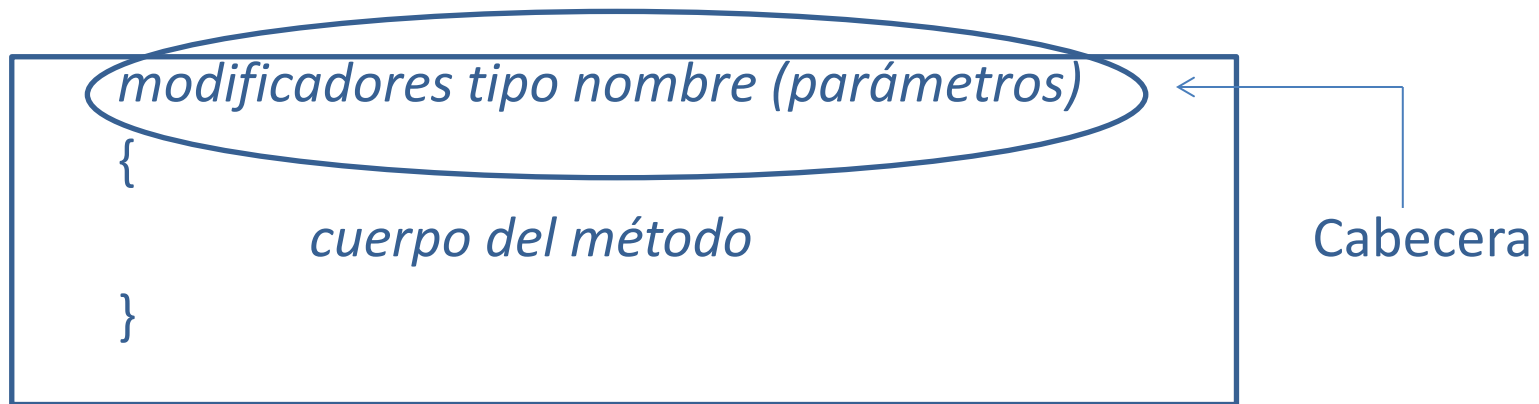
Métodos

- ☐ Definición e invocación
- ☐ Estado de la memoria
- ☐ Paso de parámetros
- ☐ Sobrecarga de métodos
- ☐ Recursión básica
- ☐ Abstracción de métodos y refinamiento por pasos

Tema 1. Fundamentos de Programación

➤ □ Definición e invocación

Sintaxis de la definición



La estructura de un método se divide en:

- **Cabecera** (determina su interfaz)

modificadores tipo nombre (parámetros)

Tema 1. Fundamentos de Programación

En la cabecera de un método se distingue:

➤ **Modificadores**

El modificador **public** indica que se puede acceder al método desde el exterior de la clase.

El modificador **static** indica que se trata de un método de clase (un método común para todos los objetos de la clase).

➤ **Tipo devuelto** (cualquier tipo primitivo, no primitivo o void)

Indica de que tipo es la salida del método, el resultado que se obtiene tras llamar al método desde el exterior.

NOTA:

void se emplea cuando el método no devuelve ningún valor.

➤ **Nombre del método**

Identificador válido en Java.

Tema 1. Fundamentos de Programación

CONVENCIÓN:

En Java, los nombres de métodos comienzan con minúscula.

- Cuando el método no devuelve ningún valor

(métodos void):

El nombre del método suele estar formado por un verbo.

Ejemplo: mostrarTriangulo

- Cuando el subprograma devuelve un valor

(métodos):

El nombre del método suele ser una descripción del valor devuelto por el método.

Ejemplo: esPrimo

Tema 1. Fundamentos de Programación

➤ Parámetros formales

Entradas que necesita el método para realizar la tarea de la que es responsable. Se indica el tipo y el identificador del parámetro. Los parámetros se separan por comas.

MÉTODOS SIN PARÁMETROS:

Cuando un método no tiene entradas, hay que poner ()

Signatura de un método

El nombre de un método, los tipos de sus parámetros y el orden de los mismos definen la *signatura* de un método.

Los modificadores y el tipo del valor devuelto por un método no forman parte de la signatura del método.

Tema 1. Fundamentos de Programación

- **Cuerpo** (define su implementación)

```
{
```

```
    // Declaraciones de variables
```

```
    ...
```

```
    // Sentencias ejecutables
```

```
    ...
```

```
    // Devolución de un valor (opcional)
```

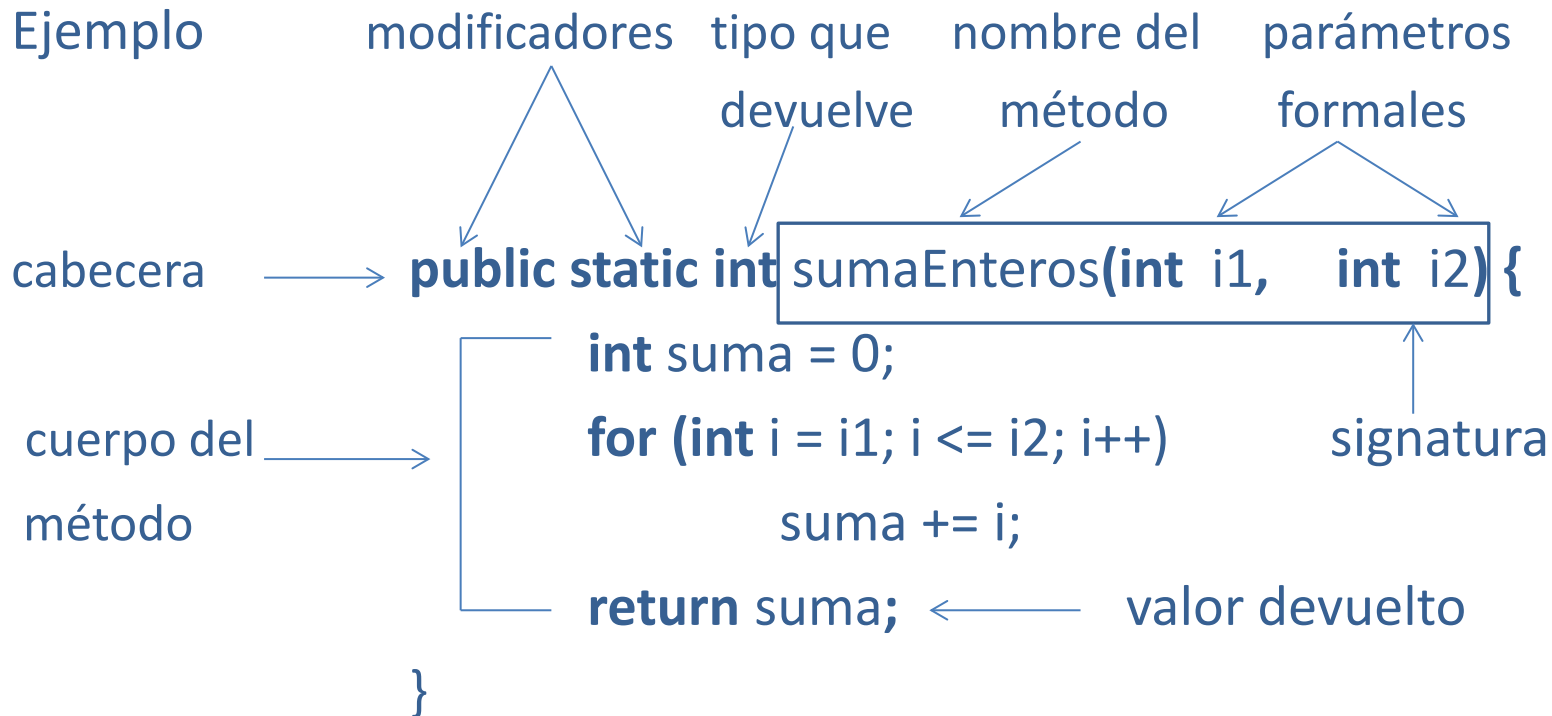
```
    ...
```

```
}
```

En el cuerpo del método se implementa el algoritmo necesario para realizar la tarea de la que el método es responsable.

Para devolver valores se utiliza la sentencia **return**. Los métodos que no devuelven nada no tendrán esta sentencia.

Tema 1. Fundamentos de Programación



Nota: sería incorrecto en la cabecera del método una declaración de parámetros del tipo: **public static int** sumaEnteros(**int i1, i2**) ¡¡ERROR!!

Tema 1. Fundamentos de Programación

Ejemplo: Definición de un método para obtener el valor máximo de dos enteros.

```
public static int max(int num1, int num2) {  
    int result;  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    return result;  
}
```


Tema 1. Fundamentos de Programación

Invocación de un método

Para usar un método, es necesario llamarlo o invocarlo. Hay dos formas de llamar a un método, dependiendo de si devuelve o no un valor.

- Si el método devuelve un valor, la llamada al método normalmente se trata como un valor. Por ejemplo:

```
int maximo = max (3, 4);
```

↓
nombre del método

parámetros actuales o reales

```
System.out.println(max(3,4));
```

- Si el método no devuelve ningún valor (void), la llamada se hará dando únicamente el identificador del método. Por ejemplo:

```
mostrarMenu();
```

Tema 1. Fundamentos de Programación

Plantilla de programa con uso de métodos

```
package org.ip.sesionNN;  
public class IdentificadorClase {  
    // Definición de métodos  
    public static tipo identificadorMetodo(parámetros){  
        // Declaraciones de variables  
        ...  
        // Sentencias ejecutables  
        ...  
        // Devolución de un valor (opcional)  
    }  
    public static void main(String[] args){  
        .....  
        //Llamada o invocación al método  
    }  
}
```

Tema 1. Fundamentos de Programación

El programa sin uso de métodos para obtener el máximo de dos números enteros

```
package org.ip.sesion04;
```

```
public class MayorValor {
```

```
    public static void main(String[] args){
```

```
        int num1 = 9, num2 = 2, result;
```

```
        if(num1 > num2)
```

```
            result = num1;
```

```
        else
```

```
            result = num2;
```

```
        System.out.println("El máximo entre " + num1 + " y "+  
            num2 + " es" + result );
```

```
    }
```

```
}
```

Tema 1. Fundamentos de Programación

El programa utilizando un método quedaría:

```
package org.ip.sesion04;
public class MayorValor {
    public static int max(int num1, int num2) {
        int result;
        if (num1 > num2)
            result = num1;
        else
            result = num2;

        return result;
    }
    public static void main(String[] args){
        int i = 9, j = 2;
        int k = max(j, i);
        System.out.println("El máximo entre " + i +
            " y " + j + " es" + k );
    }
}
```

Diagram illustrating the correspondence between formal parameters and actual parameters:

- parámetros formales (points to the parameters of the `max` method)
- parámetros actuales o reales (points to the arguments passed to the `max` method in the `main` method)

Correspondencia:
nº, posición y tipo

Tema 1. Fundamentos de Programación

Seguimiento del programa

paso el valor i

paso el valor j

```
public static void main(String[] args) {  
    int i = 9;  
    int j = 2;  
    int k = max(i, j);  
    System.out.println("El máximo entre " + i  
        + " y " + j + " es " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

Tema 1. Fundamentos de Programación

Ejemplo. Suma de enteros de 1 a 10, de 20 a 30 de 35 a 45

```
package org.ip.sesion04;
```

```
public class Suma {
```

```
    public static int sumaEnteros(int i1, int i2) {
```

```
        int suma = 0;
```

```
        for (int i = i1; i <= i2; i++)
```

```
            suma += i;
```

```
        return suma;
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        System.out.println(" La suma de 1 a 10 es " + sumaEnteros(1, 10));
```

```
        System.out.println (" La suma de 20 a 30 es " + sumaEnteros(20, 30));
```

```
        System.out.println (" La suma de 35 a 45 es " + sumaEnteros(35, 45));
```

```
    }
```

```
}
```

Tema 1. Fundamentos de Programación

Ejemplo. Algoritmo de Euclides sin uso de métodos

```
package org.ip.sesion03;

public class Euclides {
    /**
     * Calcula el máximo común divisor de dos enteros positivos utilizando el
     * algoritmo de Euclides
     *
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        int dato1 = 25;
        int dato2 = 10;
        int aux;
        while (dato1 % dato2 != 0) {
            aux = dato1;
            dato1 = dato2;
            dato2 = aux % dato2;
        }
        System.out.println("El MCD de los valores introducidos es " + dato2);
    }
}
```

Tema 1. Fundamentos de Programación

Ejemplo. Algoritmo de Euclides con uso de métodos

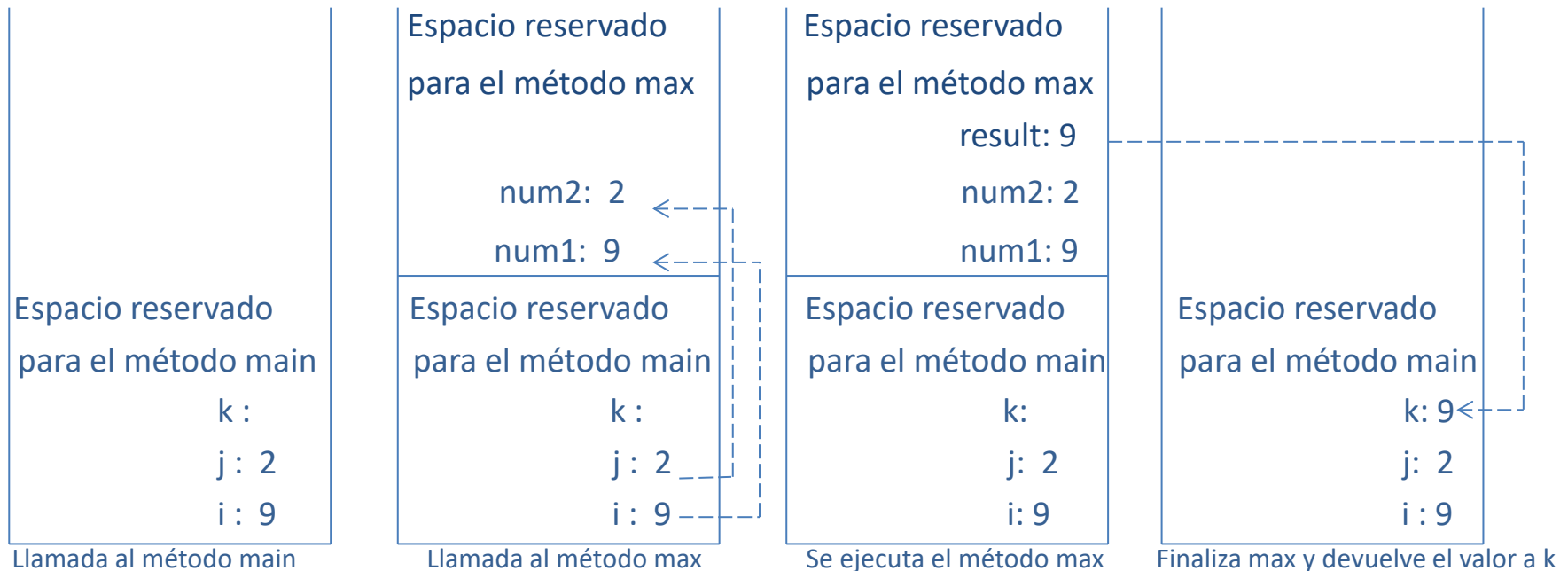
```
package org.ip.sesion04;

public class Euclides {
    /**
     * Calcula el máximo común divisor de dos enteros positivos utilizando el
     * algoritmo de Euclides
     *
     * @param args
     */
    public static int mcdEuclides(int dato1, int dato2) {
        int aux;
        while (dato1 % dato2 != 0) {
            aux = dato1;
            dato1 = dato2;
            dato2 = aux % dato2;
        }
        return dato2;
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int dato1 = 25;
        int dato2 = 10;
        System.out.println("El MCD de " + dato1 + " y " + dato2 + " es " + mcdEuclides(dato1, dato2));
    }
}
```


Tema 1. Fundamentos de Programación

Estado de la memoria

Cuando se hace una llamada a un método, el sistema almacena los parámetros y las variables en una zona de memoria conocida como *pila* (último elemento en entrar, primero en salir). En la figura se muestra el funcionamiento para el programa que calcula el máximo de dos enteros



Tema 1. Fundamentos de Programación

➤ Paso de parámetros

Cuando hacemos una llamada a un método necesitamos proporcionarle parámetros. Se relaciona el parámetro real o actual (llamada) con el formal (método). El parámetro real pasa el valor al parámetro formal y si éste se modifica en el método, el parámetro real no se ve afectado.

Estamos haciendo un paso de parámetros ***por valor***.

En el ejemplo siguiente el valor de x (1) se pasa al parámetro formal n al hacer la llamada al método incremento. En ese método, n se incrementa en 1 pero esa modificación no afecta para nada a x, se ha hecho una copia y son posiciones de memoria distintas.

Tema 1. Fundamentos de Programación

```
package org.ip.sesion04;

public class Incremento {

    public static void incremento(int n) {
        n++;
        System.out.println("n dentro del metodo es " + n);
    }

    public static void main(String[] args) {
        int x = 1;
        System.out.println("Antes de la llamada al metodo, x es " + x);
        incremento(x);
        System.out.println("Despues de la llamada al metodo, x es " + x);
    }
}
```

Salida

```
Antes de la llamada al metodo, x es 1
n dentro del metodo es 2
Despues de la llamada al metodo, x es 1
```

```
package org.ip.sesion04;

public class TestPasoPorValor {

    /** Intercambia dos variables */
    public static void swap(int n1, int n2) {
        System.out.println("\t*** Dentro del metodo swap ***");
        System.out.println("\t\tAntes del intercambio n1 es " + n1 + " n2 es " + n2);
        // Intercambio n1 con n2
        int temp = n1;
        n1 = n2;
        n2 = temp;
        System.out.println("\t\tDespues del intercambio n1 es " + n1 + " n2 es " + n2);
    }

    public static void main(String[] args) {
        // Declarar e inicializar variables
        int num1 = 1;
        int num2 = 2;

        System.out.println("Antes de la llamada al metodo swap, num1 es " + num1
            + " y num2 es " + num2);

        // Llamada al metodo swap
        swap(num1, num2);

        System.out.println("Despues de la llamada al metodo swap, num1 es " + num1
            + " y num2 es " + num2);
    }
}
```

Salida

```
Antes de la llamada al metodo swap, num1 es 1 y num2 es 2
    *** Dentro del metodo swap ***
        Antes del intercambio n1 es 1 n2 es 2
        Despues del intercambio n1 es 2 n2 es 1
Despues de la llamada al metodo swap, num1 es 1 y num2 es 2
```

Tema 1. Fundamentos de Programación

Sobrecarga de métodos

Lenguajes como Java permiten que existan distintos métodos con el mismo nombre siempre y cuando su signatura no sea idéntica. Esto se conoce como ***sobrecarga de métodos***.

Si dos métodos tienen el mismo nombre pero distinta lista de parámetros (tanto número, como tipo), el compilador de Java determina cual debe utilizar basándose en la signatura.

En el ejemplo siguiente tres métodos tienen el mismo nombre, *max*, pero difieren en la lista de parámetros.

Salida

El maximo entre 3 y 4 es 4
El maximo entre 3.0 y 5.4 es 5.4
El maximo entre 10.7, 3.8 y 5.1 es 10.7

```
package org.ip.sesion04;

public class TestSobrecargaMetodos {
    /** Devuelve el maximo entre dos valores enteros */
    public static int max(int num1, int num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
    /** Devuelve el maximo entre dos valores reales */
    public static double max(double num1, double num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
    /** Devuelve el maximo entre tres valores reales */
    public static double max(double num1, double num2, double num3) {
        return max(max(num1, num2), num3);
    }

    public static void main(String[] args) {
        System.out.println("El maximo entre 3 y 4 es " + max(3, 4));
        System.out.println("El maximo entre 3.0 y 5.4 es " + max(3.0, 5.4));
        System.out.println("El maximo entre 10.7, 3.8 y 5.1 es " + max(10.7, 3.8, 5.1));
    }
}
```

➤ Recursión básica

Si consultamos en la Real Academia Española (www.rae.es) lo que se entiende por recurrencia y recurrente encontramos:



REAL ACADEMIA ESPAÑOLA

DICCIONARIO DE LA LENGUA ESPAÑOLA - Vigésima segunda edición

recurrencia.

1. f. Cualidad de recurrente.

2. f. *Mat.* Propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes.



REAL ACADEMIA ESPAÑOLA

DICCIONARIO DE LA LENGUA ESPAÑOLA - Vigésima segunda edición

chos reservados

recurrente.

(Del ant. part. act. de *recurrir*, *recurrens*, *-entis*).

1. adj. Que recurre.

2. adj. Que vuelve a ocurrir o a aparecer, especialmente después de un intervalo.

3. adj. *Anat.* Dicho de un vaso o de un nervio: Que en algún lugar de su trayecto vuelve hacia el origen.

4. adj. *Mat.* Dicho de un proceso: Que se repite.

5. com. Persona que entabla o tiene entablado un recurso.

Hablamos de **recurrencia**, cuando definimos algo en función de si mismo (una propiedad, un tipo de objeto, una operación, etc.). La recurrencia aparece de forma natural en algunas definiciones o problemas matemáticos. Por ejemplo:

- ✓ Los números naturales se definen:
 - 0 es un n° natural
 - sucesor(x) es un n° natural si x lo es.

- ✓ La potencia con exponentes enteros se puede definir:
 - $a^0 = 1$
 - $a^n = a \cdot a^{(n-1)}$

- ✓ El factorial de un entero positivo se define:
 - $0! = 1$
 - $n! = n \cdot (n-1)!$

Diremos que una **definición recurrente** es aquella en la que se define algo en términos de *versiones más pequeñas de sí mismo*. En programación supone la posibilidad de que un *método se llame a sí mismo* y también supone el poder definir tipos de datos recursivos. Prácticamente todos los lenguajes de programación tienen esa posibilidad y para otros (los funcionales) es la herramienta por excelencia.

Definiremos un **algoritmo recurrente** o basado en relaciones de recurrencias como aquel que *se expresa en términos de instancias más pequeñas de sí mismo y un caso base*. Por tanto, un algoritmo recurrente tendrá:

- ✓ **Un caso base.** Caso para el que la solución puede establecerse de forma no recurrente. Dará la *condición de terminación*.
- ✓ **Un caso general.** También llamado caso recurrente, para el que la solución se expresa en términos de una versión más pequeña de sí mismo.

Importante:

Siempre que planteemos una solución recurrente a un problema deberemos asegurarnos de tener el **caso base** para el cual la evaluación no es recurrente y además cada llamada se realiza con un valor más pequeño para conseguir llegar al caso base o a la condición de terminación.

Ejemplo: Define un método recursivo que permita calcular el factorial de un entero no negativo.

```
public static long factorialRecursivo(int n) {  
    if (n == 0) ← Caso base  
        return 1;  
    else  
        return n * factorialRecursivo(n - 1); ← Llamada recursiva  
}
```

```
package org.ip.tema01;

public class Factorial {

    // Metodo iterativo para calcular el factorial
    public static long factorialIterativo(int n){
        if (n == 0) return 1;
        long fact = 1;
        int i = 1;
        while (i <= n) {
            fact = fact * i;
            i++;
        }
        return fact;
    }

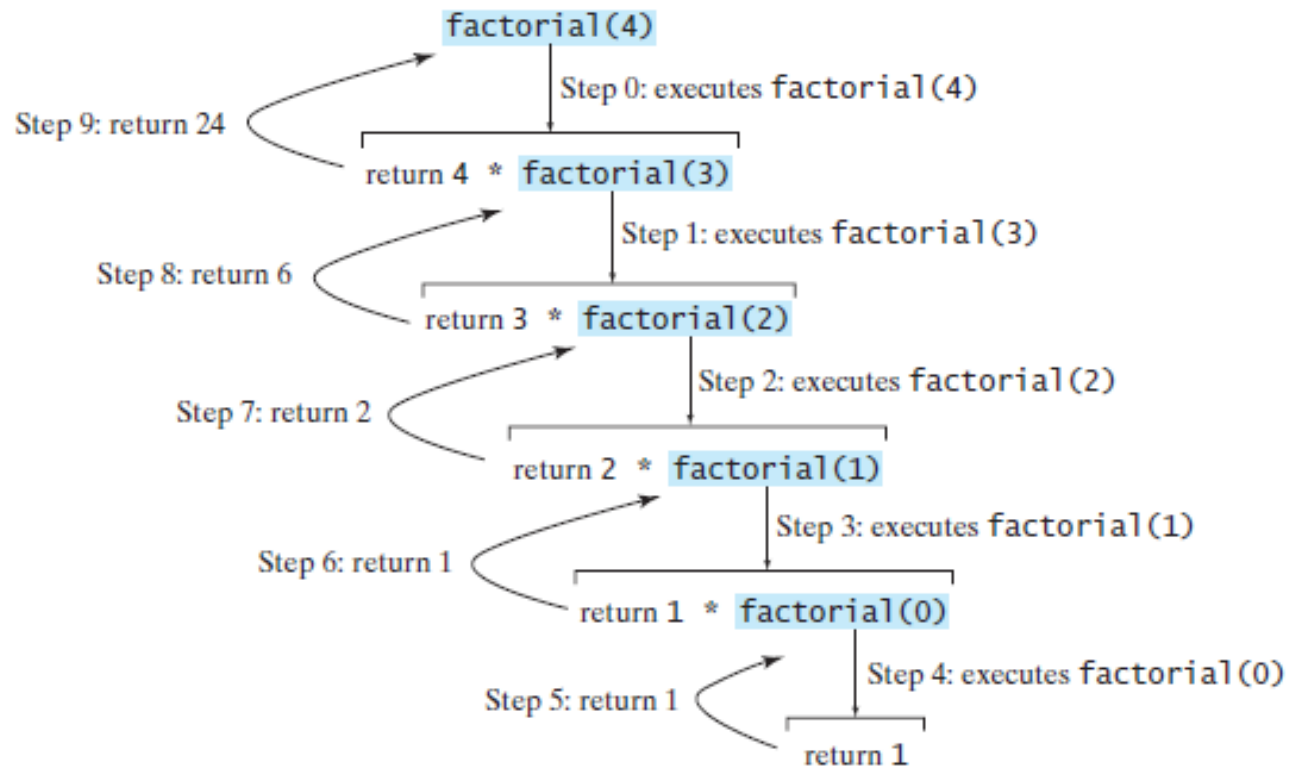
    // Metodo recursivo para calcular el factorial
    public static int factorialRecursivo(int n){
        if (n == 0)
            return 1;
        else
            return n * factorialRecursivo(n - 1);
    }

    public static void main(String[] args) {
        System.out.println("El factorial recursivo de 7 es " + factorialRecursivo(7));
        System.out.println("El factorial iterativo de 7 es " + factorialIterativo(7));
    }
}
```

Método erróneo

```
// Metodo recursivo erroneo para calcular el factorial
public static int factorialRecursivoErroneo(int n){
    return n * factorialRecursivoErroneo(n - 1);
}
```

Seguimiento de la llamada



Estado de la pila

| | | | | | | | |
|---|--|--|--|--|--|--|--|
| | | | | | | 5 | Space Required for factorial(0) n: 0 |
| | | | | | 4 | Space Required for factorial(1) n: 1 | Space Required for factorial(1) n: 1 |
| | | | 3 | Space Required for factorial(2) n: 2 | Space Required for factorial(2) n: 2 | Space Required for factorial(2) n: 2 | Space Required for factorial(2) n: 2 |
| | 2 | Space Required for factorial(3) n: 3 | Space Required for factorial(3) n: 3 | Space Required for factorial(3) n: 3 | Space Required for factorial(3) n: 3 | Space Required for factorial(3) n: 3 | Space Required for factorial(3) n: 3 |
| 1 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 |
| | | | | | | | |
| 6 | Space Required for factorial(1) n: 1 | | | | | | |
| | Space Required for factorial(2) n: 2 | 7 | Space Required for factorial(2) n: 2 | | | | |
| | Space Required for factorial(3) n: 3 | Space Required for factorial(3) n: 3 | 8 | Space Required for factorial(3) n: 3 | | | |
| | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | Space Required for factorial(4) n: 4 | 1 | Space Required for factorial(4) n: 4 | |

Ejemplo: Define un método recursivo que permita obtener el número de Fibonacci para un índice dado.

La serie de Fibonacci comienza con 0 y 1, y cada número siguiente es la suma de los dos que le preceden. Es decir:

| | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|----|----|----|----|----|-----|
| Serie | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | ... |
| índice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |

Se puede definir recursivamente:

fibonacci(0) = 0;

fibonacci(1) = 1;



Caso base

fibonacci(indice) = fibonacci(indice-2) + fibonacci(indice-1); Caso general

```
public static long fibonacci(long n) {  
    if (n == 0) //caso base  
        return 0;  
    else if (n == 1) // caso base  
        return 1;  
    else  
        return fibonacci(n - 1) + fibonacci(n - 2); }  
}
```

```
package org.ip.tema01;

import java.util.Scanner;
```

```
public class CalcularFibonacci {
```

```
    // Método recursivo para calcular el número de fibonacci
```

```
    public static long fibonacci(int n) {
```

```
        if (n == 0)           // Caso base
```

```
            return 0;
```

```
        else if (n == 1)      // Caso base
```

```
            return 1;
```

```
        else
```

```
            // Reduccion y llamada recursiva
```

```
            return fibonacci(n - 1) + fibonacci(n - 2);
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        @SuppressWarnings("resource")
```

```
        Scanner entrada = new Scanner(System.in);
```

```
        System.out.print("Introduce el índice para obtener el número de Fibonacci: ");
```

```
        int indice = entrada.nextInt();
```

```
        System.out.println("El número de Fibonacci para el índice " + indice
```

```
            + " es " + fibonacci(indice));
```

```
        int terminos; // nº de términos a mostrar
```

```
        System.out.print("Indica el número de términos que desea mostrar de la serie de Fibonacci: ");
```

```
        terminos = entrada.nextInt();
```

```
        for (int i = 0; i < terminos; i++) {
```

```
            System.out.print(fibonacci(i) + "\t");
```

```
        }
```

```
    }
```

```
}
```

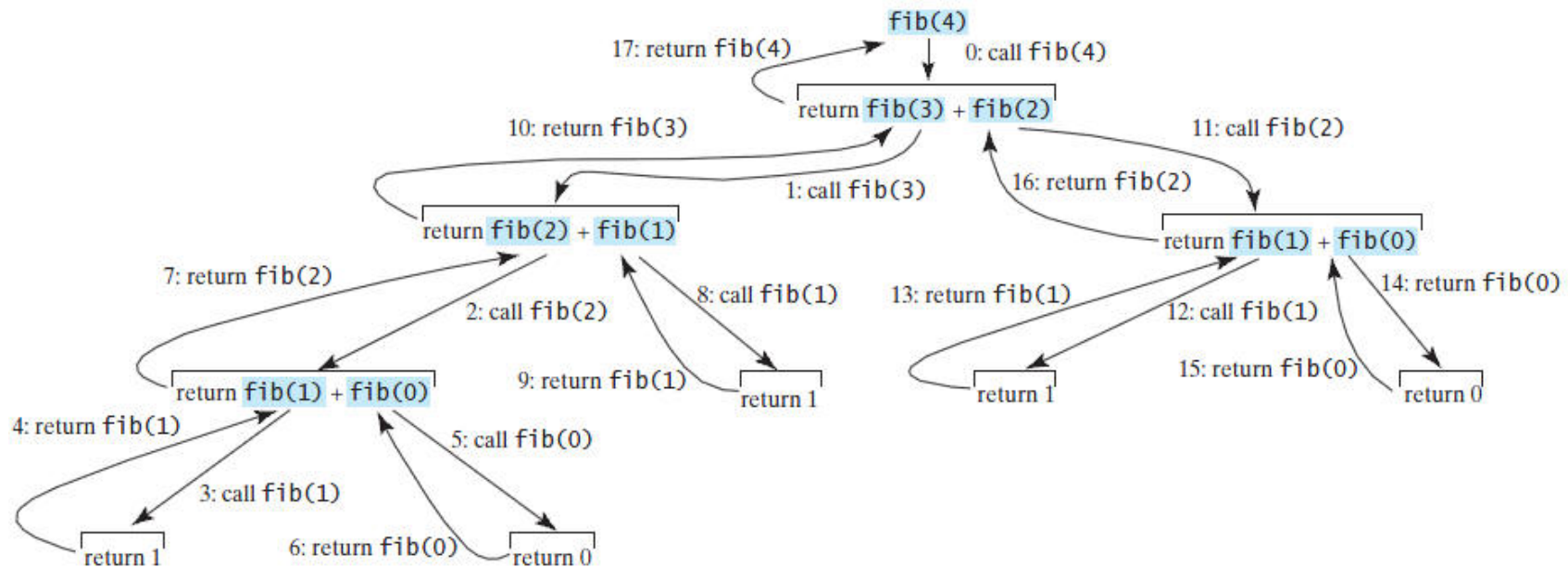
Introduce el índice para obtener el número de Fibonacci: 7

El número de Fibonacci para el índice 7 es 13

Indica el número de términos que desea mostrar de la serie de Fibonacci: 10

| | | | | | | | | | |
|---|---|---|---|---|---|---|----|----|----|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |
|---|---|---|---|---|---|---|----|----|----|

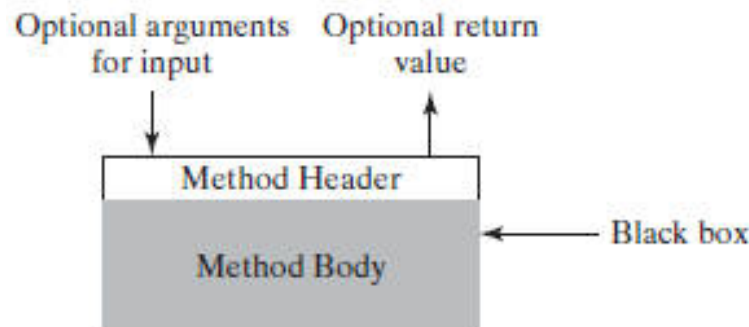
Seguimiento de una llamada al método recursivo fibonacci mostrando el orden de las llamadas.



Observamos que hay muchas llamadas recursivas repetidas. Por ejemplo `fib(2)` se llama 2 veces, `fib(1)` 3 veces esto va a suponer más consumo de tiempo y memoria. Este es un claro ejemplo de una implementación del problema sencilla de entender por la propia definición pero poco eficiente. Resultaría más eficiente la solución iterativa.

➤ □ Abstracción de métodos y refinamiento por pasos

La clave para el desarrollo de software es aplicar el concepto de *abstracción*. Aprenderemos a lo largo del curso distintos niveles de abstracción. La *abstracción de métodos* se alcanza separando el uso del método de su implementación. El usuario o cliente utilizará el método sin necesidad de tener ningún conocimiento de cómo ha sido implementado. Los detalles de implementación se encapsulan en el método y son ocultos para el usuario que invoca el método. Esto se conoce como *ocultación de la información* o *encapsulamiento*. Si decidiese cambiar la implementación, el usuario del programa no se debe ver afectado porque no se cambiara la signatura del métodos.



El concepto de *abstracción de métodos* se puede aplicar al proceso de desarrollo de programas. Cuando escribimos un programa largo, se usa la estrategia *divide y vencerás*, también conocida como *refinamiento por pasos*, para descomponer el problema en sub-problemas. Los sub-problemas pueden volverse a descomponer en otros más pequeños para hacerlos más manejables.

Supongamos que queremos hacer un programa que muestre el calendario para un mes y un año dado. El programa pedirá al usuario que introduzca el año y el mes y se mostrará el calendario como sigue:

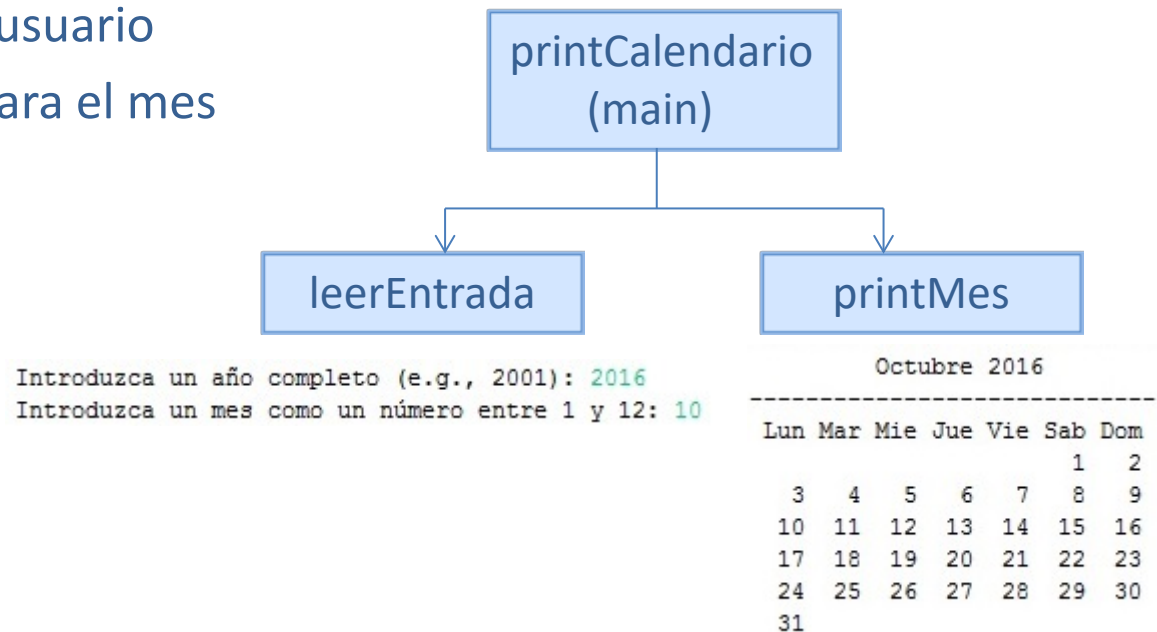
```
Introduzca un año completo (e.g., 2001): 2016
Introduzca un mes como un número entre 1 y 12: 10
      Octubre 2016
-----
Lun Mar Mie Jue Vie Sab Dom
          1   2
  3   4   5   6   7   8   9
10  11  12  13  14  15  16
17  18  19  20  21  22  23
24  25  26  27  28  29  30
31
```

Diseño descendente

¿Cómo podríamos empezar el programa? ¿Empezaríamos inmediatamente a escribir código? Seguramente que los programadores principiantes comenzarían teniendo en cuenta todos los detalles, sin embargo esos detalles son importantes pero solo al final del programa. Para hacer más sencilla la solución del problema utilizamos la abstracción de métodos y los detalles se implementan más tarde.

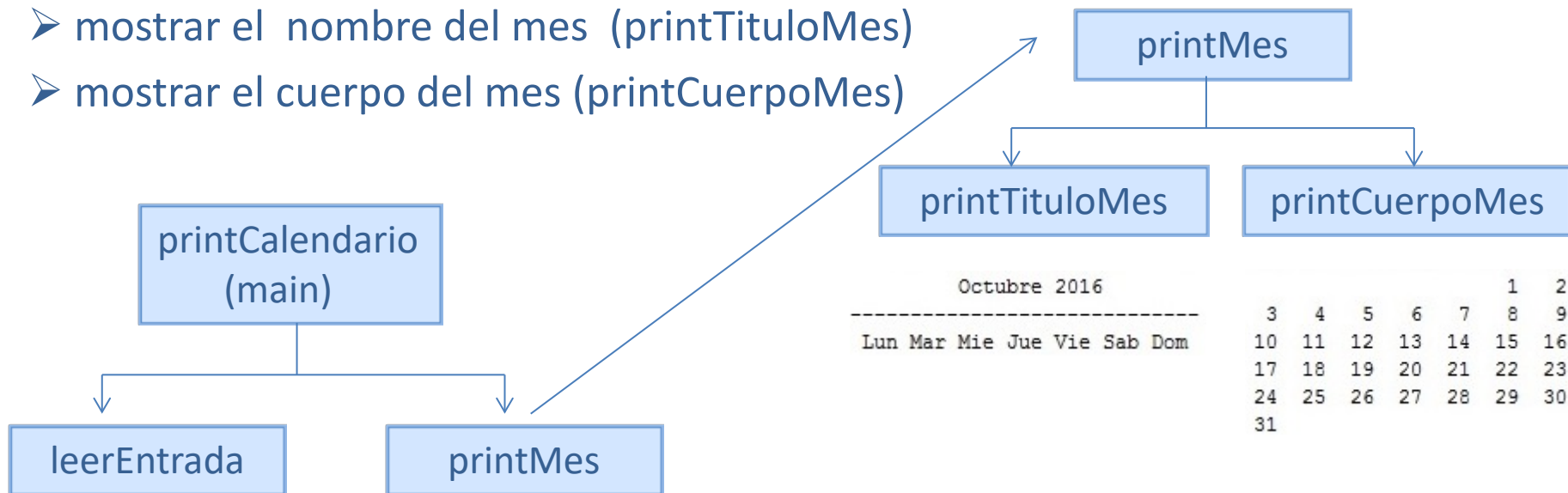
En este ejemplo, el problema se divide en dos sub-problemas:

- Obtener la entrada del usuario
- Mostrar el calendario para el mes



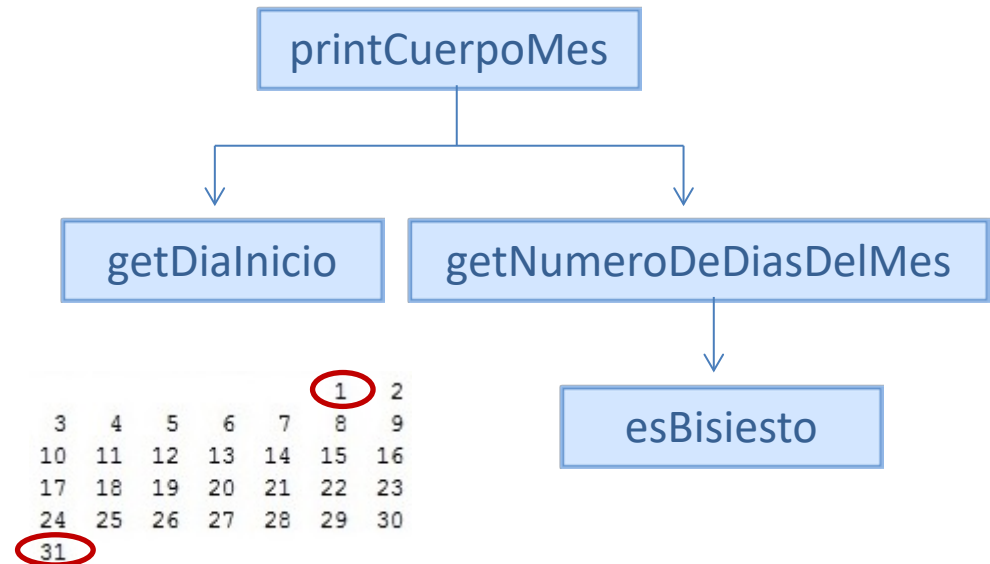
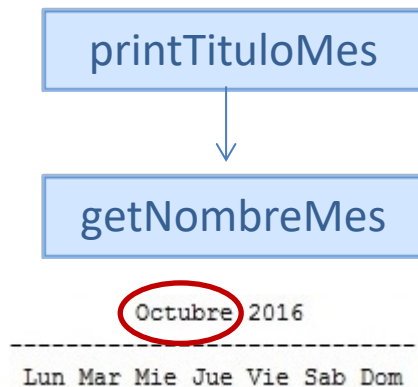
El sub-problema de mostrar el calendario (`printMes`) podríamos dividirlo en otros dos sub-problemas más sencillos :

- mostrar el nombre del mes (`printTituloMes`)
- mostrar el cuerpo del mes (`printCuerpoMes`)



El titulo del mes consistiría en tres líneas: mes y año, una línea -----, y el nombre de los siete días de la semana. Necesitaríamos obtener el nombre del mes (por ejemplo Enero) a partir de un valor numérico del mes (por ejemplo 1). Esto significa que podríamos hacer `getNombreMes` para mostrar dicho nombre

Es decir,

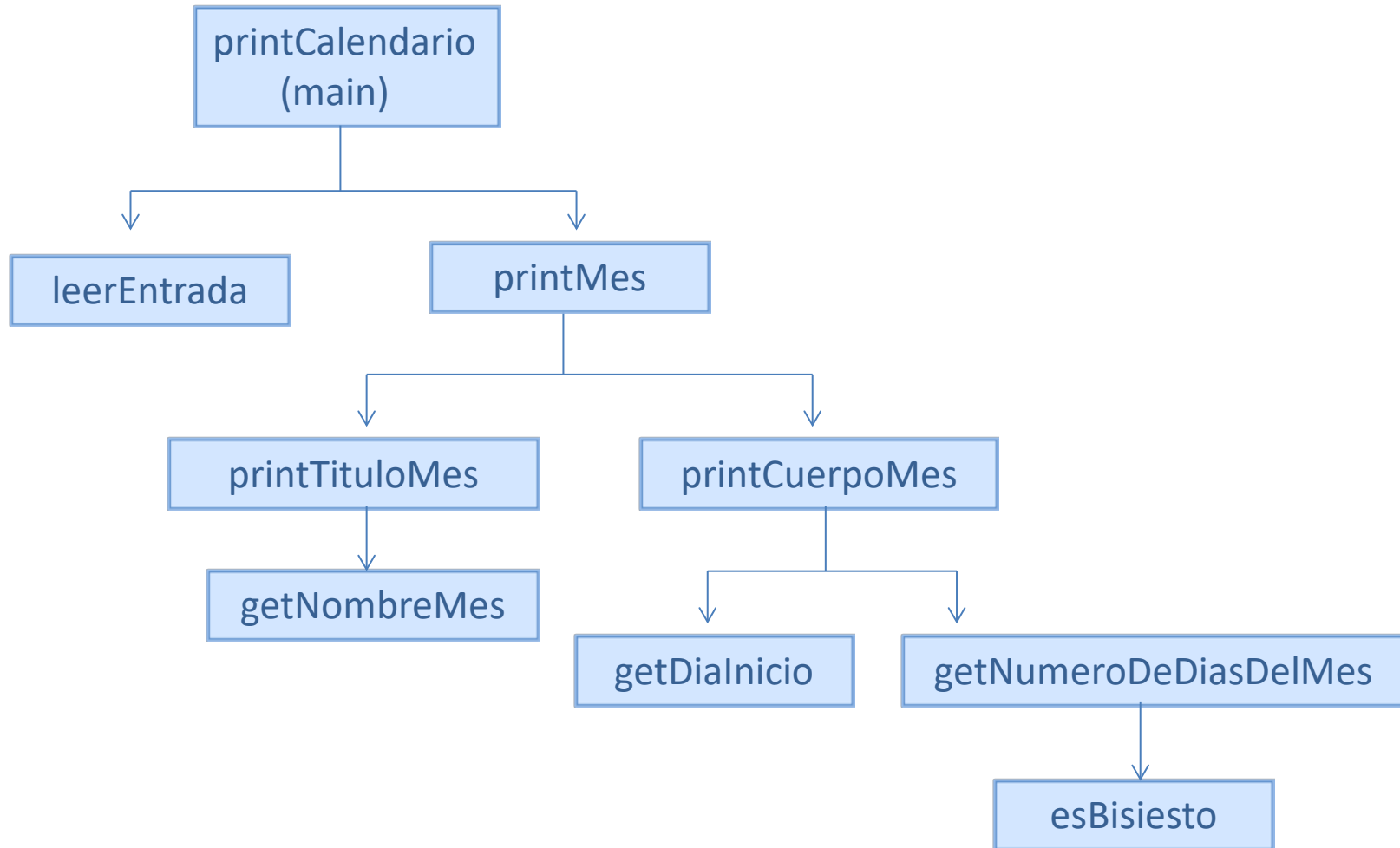


Para mostrar el cuerpo del mes, necesitaríamos conocer:

- El día de la semana en el que empieza el mes (lunes, martes, etc.), `getDiaInicio`.
- El número de días que tiene el mes, `getNumeroDeDiasDelMes`. A su vez, para obtener dicho número, necesitamos saber si el año es bisiesto.

Por ejemplo: Diciembre de 2005 tiene 31 días y el 1 de Diciembre fue jueves.

La estructura completa quedaría



Implementación de abajo a arriba

Ahora tendríamos que centrar la atención en la implementación. En general, un sub-problema corresponde a un método, incluso algunos son tan simples que no son necesarios. Tendremos que decidir qué sub-problemas se implementan como métodos y cuales se incluyen en otros métodos. Por ejemplo, el sub-problema **leerEntrada** puede implementarse en el método **main**.

De cada método haríamos una versión simple e incompleta del mismo (resguardo). Esto permite construir rápidamente un esqueleto del programa. Implementamos el método **main** y usamos los resguardos para los métodos y el programa tendría un aspecto:

```
package org.ip.tema01;
import java.util.Scanner;
public class PrintEsqueletoCalendario {
    /** Método main */
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        // El usuario introduce el año
        System.out.print("Introduzca un año completo (e.g., 2001): ");
        int año = entrada.nextInt();
    }
}
```



```
// El usuario introduce el mes
```

```
System.out.print("Introduzca un mes como un número entre 1 y 12: ");
```

```
int mes = entrada.nextInt();
```

```
// Muestra el calendario para el mes y el año introducidos
```

```
printMes(año, mes);
```

```
}
```

```
/** printMes puede parecerse a esto */
```

```
public static void printMes(int año, int mes) {
```

```
    //Debe mostrar el titulo del mes y el cuerpo del mes
```

```
}
```

```
/** printTituloMes puede parecerse a esto */
```

```
public static void printTituloMes(int año, int mes) {
```

```
}
```

```
/** printCuerpoMes puede parecerse a esto */
```

```
public static void printCuerpoMes(int año, int mes) {
```

```
}
```

/** getNombreMes puede parecerse a esto */

```
public static String getNombreMes(int mes) {  
    return "Enero"; // Un valor de ejemplo  
}
```

/** getDialInicio puede parecerse a esto */

```
public static int getDialInicio(int año, int mes) {  
    return 1; // Un valor de ejemplo  
}
```

/** getNumeroTotalDeDiasDelMes puede parecerse a esto */

```
public static int getNumeroDeDiasDelMes(int año, int mes) {  
    return 31; // Un valor de ejemplo  
}
```

/** esBisiesto puede parecerse a esto */

```
public static boolean esBisiesto(int año) {  
    return true; // Un valor de ejemplo  
}}
```

¡MUCHAS GRACIAS!



UNIVERSIDAD DE ALMERÍA

