LENGUAJE DE PROGRAMACIÓN

CON **JAVA**

España - México - Colombia - Chile - Ecuador - Perú - Bolivia - Uruguay - Guatemala - Costa Rica

EDITORIAL MACRO®

Lenguaje de programación con JAVA
Autor: Eric Gustavo Coronel Castillo

© Derechos de autor registrados:
Empresa Editora Macro EIRL

© Derechos de edición, arte gráfico y diagramación reservados:
Empresa Editora Macro EIRL

Jefe de edición:
Cynthia Arestegui Baca

Coordinación de edición:
Magaly Ramon Quiroz

Diseño de portada:
Alessandra Bonilla Zapata

Corrección de estilo:
Hassel Ortiz Huamán

Diagramación:
Verónica Calderón Cáceres

Edición a cargo de:
© Empresa Editora Macro EIRL
Av. Paseo de la República N.° 5613, Miraflores, Lima, Perú

Teléfono: (511) 748 0560
E-mail: proyectoeditorial@editorialmacro.com
Página web: www.editorialmacro.com

Segunda edición: mayo de 2015
Tiraje: 1000 ejemplares

Impresión
Talleres gráficos de la Empresa Editora Macro EIRL
Jr. San Agustín N.° 612-624, Surquillo, Lima, Perú

ISBN N.° 978-612-304-288-2
Hecho el depósito legal en la Biblioteca Nacional del Perú N.° 2015-05711
Prohibida la reproducción parcial o total, por cualquier medio o método, de este libro sin previa autorización de la Empresa Editora Macro EIRL.

Eric Gustavo Coronel Castillo

Eric Gustavo Coronel Castillo es un profesional egresado de la Facultad de Electrónica de la Universidad Nacional de Ingeniería. Desde fines de los años ochenta se dedica a la Informática & Sistemas, especialmente en los temas de Desarrollo y Base de datos. En la actualidad es uno de los profesionales más reconocido de nuestro medio, debido a su amplia experiencia.

Como Consultor de Sistemas ha brindado sus servicios a instituciones públicas y privadas como la Universidad Nacional de Ingeniería, Policía Nacional del Perú, Prodeco Asociados S.A., Constructora Racarrumy S.A., Infecom EIRL, Casas & Cosas S.A., GrapPerú S.A.C., AlfaVía y a muchas pequeñas y microempresas.

Como docente cuenta con varios años de experiencia en el dictado de cursos de desarrollo orientados al procesamiento de datos, en la elaboración de manuales técnicos, y en la difusión de tecnologías emergentes. Actualmente presta sus servicios de docencia en el Centro de Extensión y Proyección Social de la Universidad Nacional de Ingeniería, y en el Instituto Superior San Ignacio de Loyola. También ha prestado sus servicios como docente a la Universidad Privada Alas Peruanas, Universidad Particular Antenor Orrego, Universidad Nacional de Trujillo, SENCICO, GrapPerú, programa PECI del Instituto San Ignacio de Loyola, Universidad Federico Villarreal, entre otros.

Como conferencista es invitado periódicamente a exponer seminarios de difusión tecnológica en las principales ciudades del país, como Trujillo (Universidad Nacional de Trujillo), Cusco (Universidad Nacional San Antonio Abad), Chimbote (Universidad Los Angeles), Lima (Universidad Nacional de Ingeniería, Universidad Inca Garcilaso de la Vega).

En octubre del 2000 emerge su primera publicación ***Diseño de Aplicaciones Cliente Servidor con Visual Basic 6.0 & SQL Server 7.0*** con el cual se hace conocido tanto a nivel local y nacional; en enero del 2001 publica su segundo libro Diseño de Aplicaciones Cliente Servidor con Power Builder 7.0 y SQL Server 7.0, con el que recibe el reconocimiento de varios países tales como Bolivia, Ecuador, Colombia, entre otros.

En el 2005 es invitado por la Empresa Editora MACRO, y en septiembre del mismo año publica su libro titulado ***Desarrollo de Aplicaciones con PHP y MySQL***. Forma parte del grupo de autores de dicha editorial.

Dedicatoria

Esta obra la dedico a mi familia

Agradecimientos

Realizar esta obra, realmente a sido interesante, puesto que me ha permitido conocer mucho más a Java y toda su filosofía de programación. El primer problema que encontré fue en el contenido, después de revisar algunos textos, *syllabus* de las asignaturas de institutos y universidades, y conversar con varios colegas y amigos, el común denominador es que no existía en el mercado peruano un libro donde se puedan encontrar los fundamentos del lenguaje de una manera sencilla, con ejemplo fáciles de entender. Al leer esta publicación recibe un libro que servirá de base para obtener la certificación *Sun Certified Programmer for the Java 2 Platform*, y espero que sea un gran inicio para obtener esta certificación.

Por eso esta obra es también de todas las personas que han participado desde el primer momento que empecé a diagramarla. Dialogué con muchas personas, pero debo agradecer de manera especial a mis amigos y colegas por sus aportes y tiempo dedicado:

- **Sergio Matsukawa Maeda:** Sergio siempre dispuesto a colaborar en mis publicaciones, sus aportes siempre son muy precisos.
- **Ricardo Marcelo Villalobos:** Ricardo es bastante crítico, sus aportes sirvieron para hacer varios cambios, tanto en el desarrollo de los temas, como en los ejemplos.
- **Julio Flores Manco:** Julio como siempre bastante colaborador, sus sugerencias fueron bastante valiosas.
- **Hugo Valencia Morales:** Los aportes de Hugo en esta obra también son muy interesantes.
- **Jorge Guerra Guerra:** Jorge es una persona que sabe mucho de Java, sus aportes son bastante interesantes, espero que pronto publique algún libro.
- **Lizeth Flores Cortez:** Lizeth trabaja con tecnología Java en **Synopsis**, sus comentarios y sugerencias fueron bastante alentadores.
- **Luís Alcántara Herrera:** Luís trabaja junto con Lizeth, sus comentarios y aportes fueron bastante alentadores, ya te estaré buscando para el segundo libro sobre Java.

Un agradecimiento muy especial a mi familia por apoyarme en todo momento, su respaldo y motivación fueron un pilar muy importante para poder concluir esta obra.

A usted amigo lector, espero me haga llegar sus comentarios, sugerencias, etc. a mi correo electrónico gcoronelc@gmail.com, ya que todos ellos servirán para mejorar mis siguientes publicaciones.

Finalmente, reiterando mi compromiso de seguir colaborando en el desarrollo de la computación e informática con mis publicaciones. El siguiente libro que estoy preparando es con respecto a Swing y JDBC.

Introducción

Java es el lenguaje de programación que más personas en el mundo utilizan, tanto en el ambiente académico, como para el desarrollo de aplicaciones comerciales.

Java es un Lenguaje Orientado a Objetos, desde el primer programa, por mas simple que este sea, esta usted implementado una clase. Entre sus principales características podemos mencionar:

- Multiplataforma, porque Java corre sobre cualquier sistema operativo.
- Multiproceso, porque un programa Java puede ejecutar varios procesos simultáneamente.
- Orientado a Objetos
- Seguro, porque elimina aspectos como los punteros, que pueden utilizarse para acceder a secciones de la memoria no permitida.

A esto tenemos que agregarle una gran cantidad de librerías y recursos para desarrollar todo tipo de aplicaciones, desde las más sencillas, como las de consola, aplicaciones de entorno gráfico, y las más avanzadas, como componentes de aplicaciones web, aplicaciones móviles, etc.

La madurez que ha tenido Java a lo largo del tiempo, hacen que sea el lenguaje preferido por empresas que realizan desarrollo serio, y en nuestro medio son cada vez más las empresas que lo utilizan.

Este libro es el primero de la serie de libros sobre Java que pienso escribir. En este primer volumen he desarrollado el aspecto solo del lenguaje, en el siguiente estaré desarrollando temas como Swing y JDBC.

Prólogo

Ser un desarrollador competente hoy en día, implica en primer lugar tener los conceptos claros sobre la **Programación Orientada a Objetos (POO)**, ya que es la base de los lenguajes de programación.

Java es el lenguaje de programación más utilizado en el desarrollo de aplicaciones robustas que tengan un requerimiento de funcionamiento de 24x7x365, y con la ventana de poder implementarlas en cualquier plataforma.

En este contexto, es necesario contar con un texto donde podamos encontrar los conceptos fundamentales de Java, y de la POO desarrollados de una manera sencilla y fácil de comprender.

Esta obra abarca estos temas de manera muy objetiva, donde el lector podrá comprender los conceptos de POO mediante diagramas UML y luego implementados en Java.

El libro esta organizado en 17 capítulos, en cada uno de ellos se van desarrollando los fundamentos del lenguaje de la POO y encontrará muchos ejemplos a través de los cual reforzará los conceptos teóricos.

Capítulo 01: Introducción General

En este capítulo se desarrolla una visión general sobre Java, para poder tener una idea clara del alcance que puede tener el desarrollo de aplicaciones con este maravilloso lenguaje.

Capítulo 02: Software e Instalación

En este capítulo usted conocerá de donde bajar el software y su respectiva instalación.

Capítulo 03: Herramientas de Desarrollo

Para desarrollar una aplicación en Java debemos utilizar un IDE, en este capítulo usted sabrá de donde obtener los IDE's mas utilizados en nuestro medio.

Capítulo 04: Fundamentos de la Programación Orientada a Objetos

En este capítulo encontrará los conceptos básicos de la POO, para poder entender la programación con Java.

Capítulo 05: Fundamentos del Lenguaje

En este capítulo empezará a desarrollar pequeños programas utilizando los diferentes tipos de operadores.

Capítulo 06: Instrucciones de Control

En este capítulo se desarrollan las diferentes instrucciones de control, las selectivas y las repetitivas. Es un tema fundamental para controlar el flujo de nuestros programas.

Capítulo 07: Arreglos

Los arreglos son la forma de mantener datos estructurados en memoria. En este capítulo aprenderá a crear y gestionar arreglos.

Capítulo 08: Clases y Objetos

Primer contacto con la POO. En este capítulo aprenderá a implementar clases, métodos, constructores y sobrecarga.

Capítulo 09: Herencia

La herencia es un aspecto fundamental en la POO. En este capítulo aprenderá a implementar la herencia.

Capítulo 10: Paquetes y Clases

Los paquetes son la manera como Java permite organizar nuestras clases. En este capítulo usted sabrá cómo crear y administrar sus paquetes.

Capítulo 11: Clases e Interfaces

En este capítulo aprenderá a crear e implementar interfaces, para garantizar la funcionalidad de sus objetos.

Capítulo 12: Excepciones

El control de errores es un tema fundamental en toda aplicación, en Java se implementa mediante excepciones. En este capítulo aprenderá a gestionar excepciones para controlar los errores en tiempo de ejecución.

Capítulo 13: Tipos Enumerados

Los tipos enumerados permiten al programador definir sus propios tipos para casos en los que necesita representar mediante un tipo de datos, valores finitos y puntuales, a estos nuevos tipos se les llama Tipos enumerados, como por ejemplo los meses del año. En este capítulo aprenderá a crear y gestionar tipos enumerados.

Capítulo 14: Autoboxing y Unboxing

Esta es una característica muy interesante del lenguaje Java, que alivia el trabajo de estar convirtiendo de manera explicita tipos de datos.

Capítulo 15: Hilos (Threads)

Los hilos permiten ejecutar varios procesos en paralelo. En este capítulo aprenderá a crear hilos y lanzarlos para su ejecución.

Capítulo 16: I/O (Entrada / Salida)

Importante es tener claro el concepto de flujo, ya que es la base para todo tipo de comunicación en Java. En este capítulo usted aprenderá el concepto de flujos, y algunas de sus aplicaciones, como lectura desde el teclado, y acceso a archivos.

Capítulo 17: Manejo de Cadenas

El manejo de cadenas es otro tema importante en un lenguaje de programación. En este capítulo usted aprenderá que las cadenas en Java son objetos de la clase String, y por lo tanto implementa métodos para su manipulación.

Finalmente, a usted amigo lector espero que esta obra le sea de mucha utilidad, y agradeceré mucho los comentarios que pueda hacer llegar a la dirección de mi correo electrónico gcoronel@ uni.edu.pe, que contribuirán a mejorar mis futuras publicaciones.

Índice

Capítulo 1: Fundamentos de la Programación

1.1 Historia de Java 25

1.1.1 Orígenes 25

1.1.2 Aspectos cronológicos 26

a. Java 1 26

b. Java 2 26

1.2 Características de Java 27

1.2.1 Plataforma de ejecución 27

1.2.2 Características del Lenguaje 28

a. Simple 28

b. Orientado a objetos 28

c. Distribuido 29

d. Robusto 29

e. Arquitectura neutral 30

f. Seguro 31

g. Portable 33

h. Interpretado 34

i. Multihilo 38

j. Dinámico 38

1.2.3 Ejecución de programas Java 39

1.3 Productos & Tecnología relacionada con Java 40

1.3.1 J2SE (Java2 Standard Edition) 40

a. JRE – Java2 *Runtime Enviroment* 41

b. JDK[1] – Java2 *Development Kip* 41

1.3.2 J2EE (Java2 Enterprise Edition) 41

1.3.3 J2ME (Java2 Micro Edition) 42

1.3.4 Java Card Technology 42
1.3.5 Java Web Services 43
1.3.6 Java Business Integration (JBI) 43
1.3.7 Java XML 43

Capítulo 2: Software e Instalación

2.1 Software 45
2.2 Instalación del JDK 46
2.3 Herramientas del JDK 51
2.3.1 Herramientas básicas del JDK 52
2.3.2 Herramientas de seguridad 53
2.3.3 Herramientas de internacionalización 53
2.3.4 Herramientas RMI (Remote Method Invocation) 54
2.4 Probando Java 54
2.4.1 Carpeta de trabajo 54
2.4.2 Verificando la variable de entorno PATH 54
2.5 Ejecutando el primer programa 56
2.6 Configurando la variable PATH 57

Capítulo 3: Herramientas de desarrollo

3.1 Historia de Java 61
3.2 EditPlus 62
3.3 JDeveloper 63
3.4 NetBeans 64
3.4.1 Inicio de un nuevo proyecto 66
3.5 JCreator Pro 68
3.5.1 Inicio de un nuevo proyecto 70
3.5.2 Agregar una nueva clase 71
3.6 Eclipse 72
3.6.1 Inicio de un nuevo proyecto 75
3.6.2 Agregar una nueva clase 75

3.7 JBuilder 77
3.7.1 Inicio de un nuevo proyecto 79
3.7.2 Agregar una nueva clase 79
Capítulo 4: Fundamentos de la Programación Orientada a Objetos
4.1 Introducción 83
4.2 Conceptos básicos sobre POO 84
4.2.1 Definición de clase 84
4.2.2 Definición de objeto 86
4.2.3 Mensaje 88
4.3 Encapsulación 89
4.4 Herencia 92
4.4.1 This 94
4.4.2 Super 94
4.5 Polimorfismo 95
4.6 Planteamiento de la implementación 97
4.7 Diseño de la estructura y el comportamiento de un objeto 98
4.7.1 Clase 98
4.7.2 Diferencia entre operación y método 99
Capítulo 5: Fundamentos de lenguaje
5.1 Esquema de un programa Java 101
5.1.1 Instanciación de una clase 102
5.1.2 Argumentos del método Main 104
5.2 Elementos básicos del lenguaje 107
5.2.1 Identificadores 107
5.2.2 Palabras clave e identificadores reservados 107
5.2.3 Comentarios 109
5.2.4 Punto y coma; bloques y espacios en blanco 109
5.2.5 Tipos de datos primitivos 111
a. Tipo boolean 112

b. Tipo char 113
c. Tipos byte, short, int y long 113
d. Tipos flot y double 115
5.2.6 Variables y operadores 116
a. Variables 116
b. Operadores aritméticos 116
c. Expresión lógica 119
5.2.7 Transformación de tipos: casting 126
5.3 Impresión en la consola 127
5.3.1 Método print() 127
5.3.2 Método println() 128
5.3.3 Método printf() 128
Capítulo 6: Instrucciones de control
6.1 Introducción 131
6.2 Instrucciones selectivas 131
6.2.1 Selectiva simple: if 131
6.2.2 Selectiva doble: if - else 133
6.2.3 Anidamiento de instrucciones if 134
6.2.4 Selectiva múltiple: switch 137
6.3 Instrucciones repetitivas 140
6.3.1 Bucle while 140
6.3.2 Bucle do – while 144
6.3.3 Bucle for 145
6.3.4 Bucle for con arreglos 149
6.3.5 Bucle for con colecciones 151
6.4 Instrucciones: break y continue 152
Capítulo 7: Arreglos
7.1 ¿Qué es un arreglo? 153
7.2 Arreglos unidimensionales – vectores 154

7.2.1 Declaración de arreglos 154
7.2.2 Creación de arreglos 154
7.2.3 Acceso a elementos de un arreglo 155
7.2.4 Inicialización de un arreglo 155
7.2.5 Averiguar el tamaño de un arreglo 156
7.3 Aplicación de arreglos unidimensionales 156
7.3.1 Generación de números aleatorios 156
7.4 Arreglos bidimensionales – matrices 162
7.4.1 Declaración de arreglos bidimensionales 162
7.4.2 Creación de arreglos 163
7.4.3 Acceso a los elementos de una matriz 163
7.4.4 Inicialización de una matriz 164
7.5 Aplicación de arreglos bidimensionales 164
Capítulo 8: Clases y objetos
8.1 Clases versus objetos 169
8.1.1 Encapsulación 170
8.1.2 Herencia 170
8.1.3 Polimorfismo 170
8.2 Definición de una clase 171
8.3 Declaración de objetos 171
8.3.1 Operador new 172
8.3.2 Asignación de objetos 172
8.4 Definición de variables o campos 173
8.5 Definición de métodos 174
8.6 Ocultando los datos 178
8.7 Sobrecarga 181
8.8 Constructores 182
8.8.1 Constructores con parámetros 184
8.8.2 Sobrecarga de constructores 186

Capítulo 9: Herencia

9.1 Fundamentos 191
9.2 Implementando la herencia 194
9.2.1 Palabra clave extends 194
9.2.2 Herencia de variables o campos 195
9.2.3 Herencia de métodos 196
9.3 Palabras claves: this y super 199
9.3.1 Palabra clave this 199
9.3.2 Palabra clave super 201
a. Invocar constructores de la superclase 201
b. Acceder a miembros de la superclase 201
9.4 Palabras claves: abstract, final y static 204
9.4.1 Palabra clave abstract 204
a. Sintaxis: Clases abstractas 205
b. Sintaxis: Métodos abstractos 205
9.4.2 Palabra clave final 207
a. Constantes 207
b. Métodos finales 208
c. Clases finales 208
9.4.3 Palabra clave static 213
a. Variables estáticas 213
b. Métodos estáticos 213

Capítulo 10: Paquetes y clases

10.1 Paquetes 217
10.1.1 Definición de paquete 217
10.2 Paquetes y CLASSPATH 220
10.3 Importación de paquetes 221
10.4 Protección de acceso 225

Capítulo 11: Clases e interfaces

11.1 ¿Qué es una interfaz? 231

11.2 Uso de interfaces 234

11.2.1 Definición de una interfaz 234

11.2.2 Implementación de una interfaz 235

11.2.3 Variables e interfaces 236

11.2.4 Interfaces pueden extenderse 238

11.3 Ejemplo ilustrativo 239

Capítulo 12: Excepciones

12.1 Fundamentos 243

12.1.1. Tipos de errores 243

a. Errores de sintaxis 243

b. Errores de lógica 244

c. Errores de ejecución 245

d. Errores típicos 246

12.1.2 ¿Qué es una excepción? 247

12.1.3 Gestión de excepciones 248

12.1.4 Esquema general 248

12.1.5 Tipos de excepciones 249

12.2 Gestionando excepciones 250

12.2.1 Excepciones no capturadas 250

12.2.2 Captura de errores 251

12.2.3 Descripción de la excepción 254

12.2.4 Cláusulas catch múltiples 254

12.2.5 Instrucción throw 257

12.2.6 Palabra clave throws 259

12.2.7 Palabra clave finally 261

12.2.8 Creación de excepciones personalizadas 263

12.2.9 Excepciones encadenadas 265

Capítulo 13: Tipos enumerados

13.1 Fundamentos 269

13.2 Usar datos enumerados 270

13.2.1 Palabra clave enum 270

13.2.2 Uso de tipos enumerados 270

13.2.3 Métodos values() y valueOf() 273

13.2.4 Tipos enumerados en línea 274

13.3 Los tipos enumerados son clases 276

13.3.1 Constructores y métodos 276

13.3.2 Implementación de interfaces 279

Capítulo 14: Autoboxing y Unboxing

14.1 Clases envolventes 283

14.1.1 Clase number 284

14.1.2 Double y Float 284

14.1.3 Métodos isInfinite() e isNaN() 286

14.1.4. Byte, Short, Integer y Long 287

14.1.5 Character 288

14.1.6 Boolean 289

14.2 Autoboxing 290

14.3 Unboxing 291

14.4 Operaciones con tipos envolventes 292

14.4.1 Incremento y decremento con tipos envolventes 292

14.4.2 Boolean versus boolean 292

14.4.3 Condicionales y Unboxing 293

14.4.4 Resolución de métodos sobrecargados 294

Capítulo 15: Hilos (Threads)

15.1 ¿Qué son hilos? 297

15.2 Implementación de hilos 301

15.2.1 La clase thread 301

15.2.2 El hilo principal 304
15.2.3 La interfaz runnable 306
15.3 Sincronización 311
15.3.1 Palabra clave synchronized 314
15.3.2 Instrucción synchronized 317
Capítulo 16: I/O (Entrada/Salida)
16.1 Flujos (streams) 321
16.1.1 Flujos de bytes 322
16.1.2 Flujos de caracteres 322
16.1.3 Flujos predefinidos 322
16.2 Entrada / Salida por consola 324
16.2.1 Métodos: print() y println() 324
16.2.2 Método read() 324
16.3 Trabajando con flujos de caracteres 326
16.3.1 Clase reader 326
16.3.2. Entrada por consola 327
16.3.3 Clase writer 329
16.3.4 Salida por consola 330
16.3.5 Lectura y escritura de archivos 330
Capítulo 17: Manejo de cadenas
17.1 Fundamentos 337
17.2 Constructores de la clase String 338
17.3 Operaciones con cadenas 340

Fundamentos de la programación

CAPÍTULO 1

Capacidad: En este capítulo veremos un panorama general de Java. Esto nos permitirá tener una idea acerca del alcance de este extraordinario lenguaje.

1.1 HISTORIA DE JAVA

1.1.1 ORÍGENES

En los primeros años de la década de los noventa, *Sun Microsystems* decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Posteriormente, decidió crear una filial, denominada *FirstPerson Inc.*, para dar margen de maniobra al equipo responsable del proyecto.

La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar códigos de escaso tamaño; por ello, James Gosling diseñó Java que originalmente se llamo *Oak* (roble).

Debido a la existencia de distintos tipos de CPUs y los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Se desarrolló un código «neutro» que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una «máquina hipotética o virtual» denominada **Java Virtual Machine** (JVM). Era la JVM que interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se convirtió en el principal lema del lenguaje: «Write once, Run *everywhere*». A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadoras, Java se introdujo a finales de 1995. La clave fue la incorporación de un intérprete Java en la versión 2.0 del programa *Netscape Navigator*, produciendo una verdadera revolución en Internet. Java 1.0 oficialmente es presentado en enero de 1996. Poco tiempo después a principios de 1997 apareció Java 1.1, mejorando sustancialmente la primera versión del lenguaje. Al año siguiente, Java 1.2 se rebautizó como Java 2.

- **Java 2**
  Java 2, antes llamado Java 1.2 o JDK 1.2 es la tercera versión importante del lenguaje de programación Java. Todas las versiones posteriores a Java 1.2 se conocen como Java 2. Actualmente estamos en Java 2 Versión 5.

1.1.2 Aspectos cronológicos

a. Java 1

- Java 1.0 (enero, 1996)
  Compuesto por 8 paquetes, 212 clases. Es la primera versión pública. La presión mediática hizo que se publicara demasiado pronto; lo cual significó que el diseño del lenguaje no sea bueno y existiera numerosos errores. Respecto a la seguridad, es restrictivo por defecto.

- Java 1.1 (marzo, 1997)
  Compuesto de 23 paquetes, 504 clases. Tiene mejoras de rendimiento en el JVM, nuevo modelo de eventos en AWT, clases anidadas, serialización de objetos, API de JavaBeans, archivos jar, internacionalización, API Reflection (*Reflexión*), JDBC (*Java Data base Connectivity*), RMI (*Remote Method Invocation*). Se añade la firma del código y la autentificación. Es la primera versión lo suficientemente estable.

b. Java 2

- Java 1.2 (diciembre, 1998)
  Compuesto de 59 paquetes, 1520 clases, JFC (*Swing*), Drag and Drop, Java2D, Corba, API Collections. Se producen mejoras notables en todos los niveles. Para enfatizar esto, Sun lo renombra como «Java 2». El JDK (Java Development Kit) se renombra como SDK (Software Development Kit). Se divide en J2SE, J2EE y J2ME.

- Java 1.3 (abril, 2000)
  Compuesto de 77 paquetes, 1595 clases. Está orientada sobre todo a la resolución de errores y a la mejora del rendimiento. Se producen algunos cambios menores como la inclusión de JNDI (*Java Naming and Directory Interface*) y la API Java Sound. También incluye un nuevo compilador de alto rendimiento JIT (*Just In Time*).

- Java 1.4 (2002)
  Compuesto de 103 paquetes, 2175 clases. También conocido como Merlin. Mejora notablemente el rendimiento y añade, entre otros, soporte de expresiones regulares, una nueva API de entrada/salida de bajo nivel (NIO, New I/O), clases para el trabajo con Collections, procesado de XML; y mejoras de seguridad como el soporte para la criptografía mediante las *Java Cryptography Extension* (JCE), la inclusión de la *Java Secure Socket Extension* (JSSE) y el *Java Authentication and Authorization Service* (JAAS).

- Java 1.5 (octubre 2004)
  Compuesto de 131 paquetes, 2656 clases. También conocido como Tiger, renombrado Java 2 5.0. Incluye como principales novedades:

  – Tipos genéricos (*generics*)
  – *Autoboxing/unboxing* conversiones entre tipos primitivos y los *wrappers* correspondientes.
  – Enumerados
  – Bucles simplificados
  – *Printf*
  – Funciones con número de parámetros variables
  – Metadatos en clases y métodos.

1.2 CARACTERÍSTICAS DE JAVA

1.2.1 PLATAFORMA DE EJECUCIÓN

Las aplicaciones Java (**.java**) para poder ejecutarlas deben ser compiladas, dando como resultado un código intermedio denominado *bytecode* (**.class**).

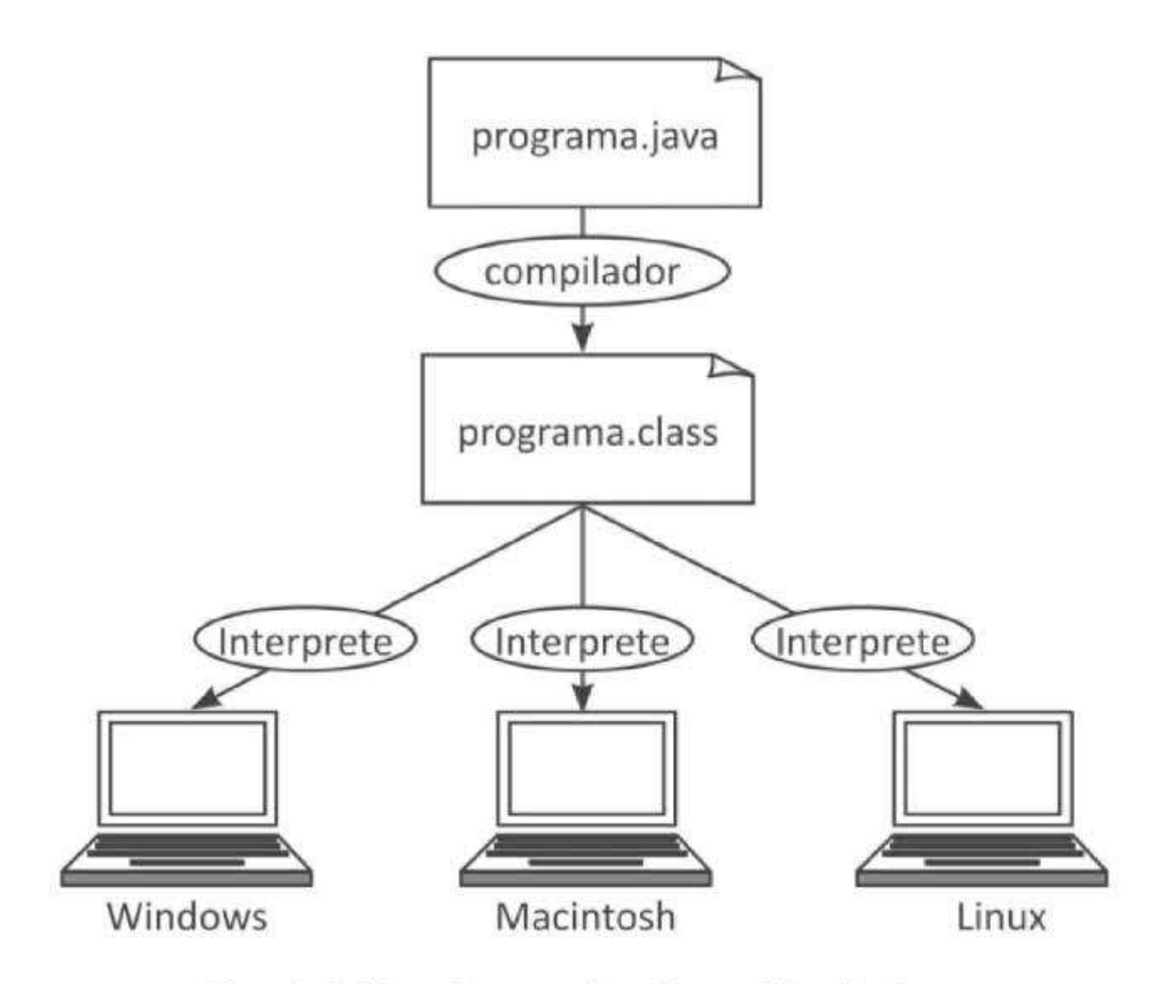

Fig. 1.1 Plataforma de ejecución de Java

Los *bytecode* son ejecutados por la Máquina Virtual de Java (JVM), que los interpreta y luego traduce al lenguaje máquina según la plataforma donde se ejecute.

La gran ventaja de la Máquina Virtual de Java es aportar portabilidad al lenguaje, de manera que desde *Sun* se han creado diferentes máquinas virtuales java para diferentes arquitecturas y así un programa **.class** escrito en Windows puede ser interpretado en un entorno Linux. Solo es necesario disponer de esta máquina virtual para dichos entornos.

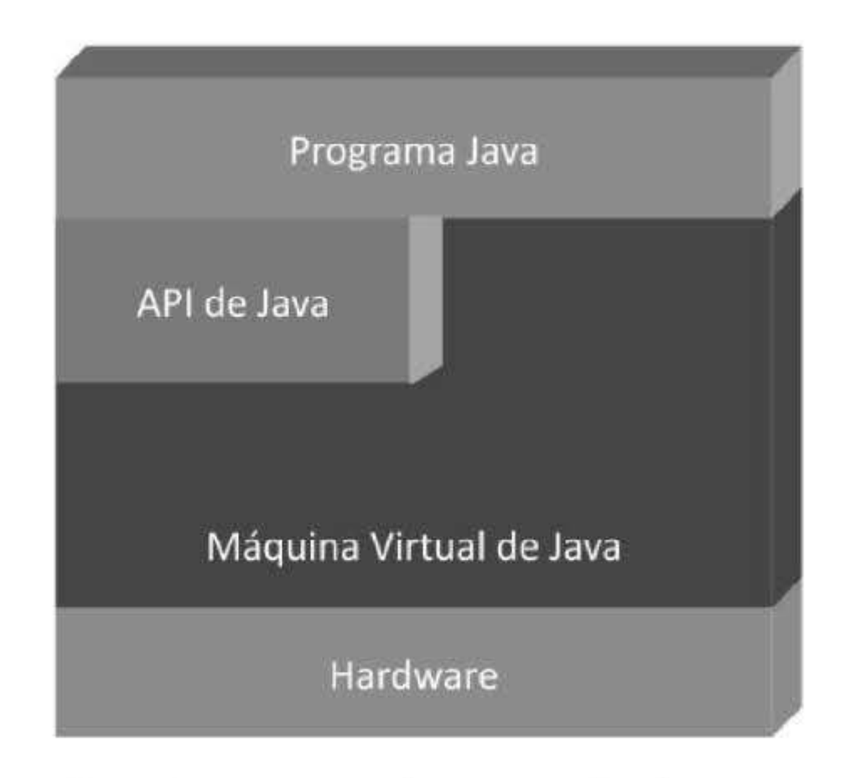

Fig. 1.2 Esquema de ejecución de Java

1.2.2 Características del Lenguaje

a. Simple

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de estos. Elimina muchas de las características de otros lenguajes, para mantener reducidas las especificaciones y añadir características muy útiles como el *Garbage Collector* (reciclador dinámico de memoria). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es un *thread* (hilo) de baja prioridad, cuando entra en acción permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.

Java reduce en un 50 % los errores más comunes de programación con otros lenguajes, ya que elimina muchas de las características de estos. Destacan:

- Aritmética de punteros
- No existen referencias
- Registros (**struct**)
- Definición de tipos (**typedef**)
- Macros (**#define**)
- Necesidad de liberar memoria (**free**)

Aunque en realidad lo que hace es eliminar las palabras reservadas (**struct, typedef**), pues las clases son algo parecidas.

b. Orientado a objetos

Java implementa la tecnología básica de lenguajes orientados a objetos con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo. Las plantillas de objetos se llaman **clases** y sus objetos, **instancias**. Estas instancias, como en otros lenguajes, necesitan ser construidas y destruidas en espacios de memoria.

Java incorpora funcionalidades como por ejemplo, la resolución dinámica de métodos. Esta característica deriva del lenguaje Objective C, propietario del Sistema Operativo Next. En otros lenguajes se suele trabajar con librerías dinámicas (DLLs) que obligan a recompilar la aplicación cuando se modifican las funciones que se encuentran en su interior. Este inconveniente es resuelto por Java mediante una interfaz específica llamada RTTI (*RunTime Type Identification*) que define la interacción entre objetos excluyendo variables de instancias o implementación de métodos. Las clases en Java tienen una representación en tiempo de ejecución que permite a los programadores interrogar por el tipo de clase y enlazar dinámicamente la clase con el resultado de la búsqueda.

c. Distribuido

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como **http** y **ftp**. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

La verdad es que Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas, interactuando entre ellas.

d. Robusto

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador (liberación o corrupción de memoria). También implementa los *arrays* auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobreescribir o corromper la memoria resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo de desarrollo de aplicaciones en Java.

Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los *byte-codes*, que son el resultado de la compilación de un programa Java. Es un código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el hardware, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.

Java proporciona, pues:

- Comprobación de punteros
- Comprobación de límites de *arrays*
- Excepciones
- Verificación de *byte-codes*

e. Arquitectura neutral

Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto, de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (*run time*) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado. Actualmente existen sistemas *run time* para Solaris 2.x, SunOs 4.1.x, Windows 95/98, Windows NT, Linux, Irix, Aix, Mac, Apple y probablemente haya grupos de desarrollo trabajando en versiones para otras plataformas.

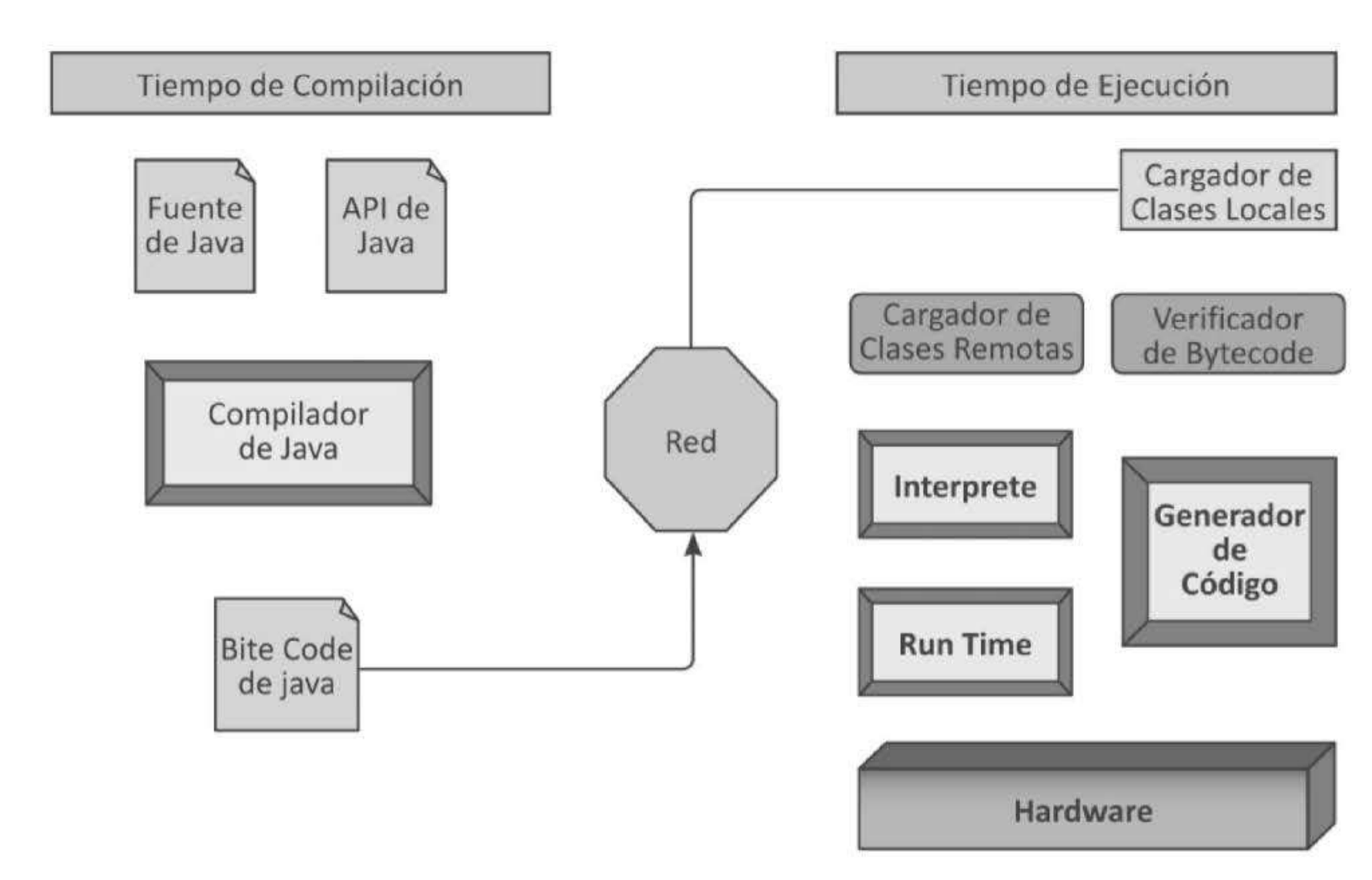

Fig. 1.3 Esquema de ejecución de aplicaciones Java

El código fuente Java se «compila» a un código de *bytes* de alto nivel independientemente de la máquina. Este código (*byte-codes*) está diseñado para ejecutarse en una máquina hipotética que es implementada por un sistema *run time*, que sí es dependiente de la máquina.

En una representación en que tuviésemos que indicar todos los elementos que forman parte de la arquitectura de Java sobre una plataforma genérica, obtendríamos un diagrama como el de la Figura 1.4.

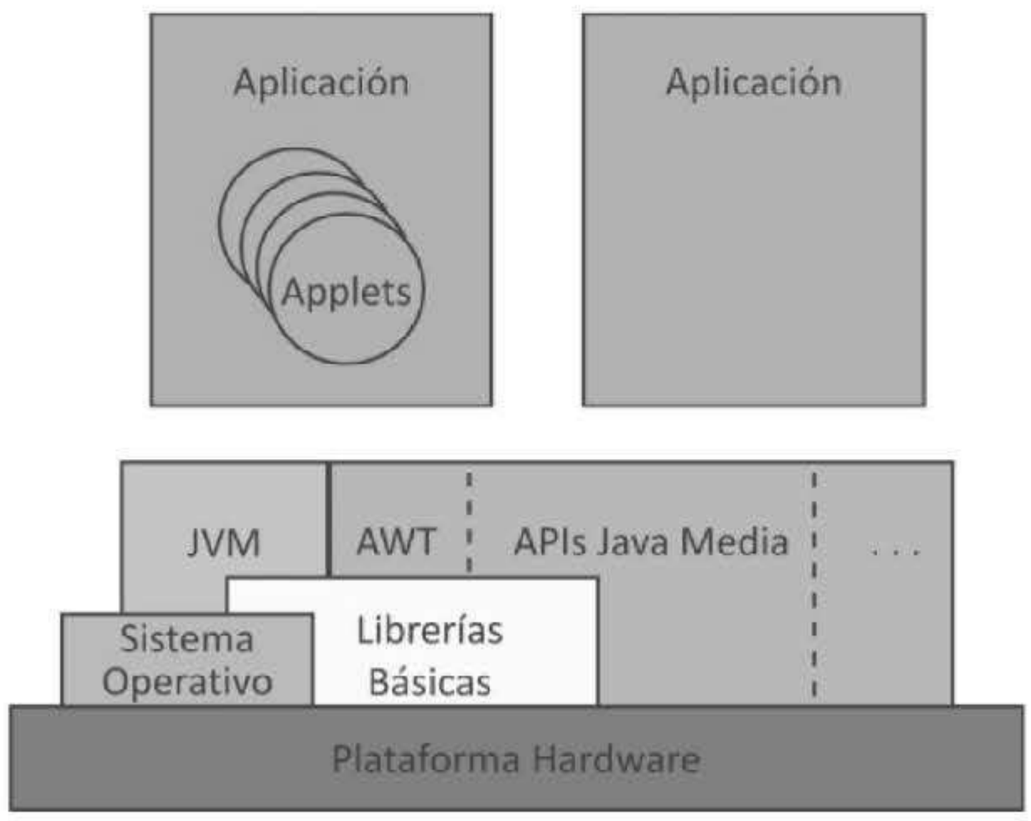

Fig. 1.4 Java sobre una plataforma genérica

En ella podemos ver que lo verdaderamente dependiente del sistema es la Máquina Virtual Java (JVM) y las librerías fundamentales, que también nos permiten acceder directamente al hardware de la máquina. Además, los APIs de Java entrarán en contacto directo con el hardware y serán dependientes de la máquina, como ejemplo de este tipo de APIs podemos citar:

- Java 2D: gráficos 2D y manipulación de imágenes
- Java Media Framework : Elementos críticos en el tiempo: audio, video, etc.
- Java Animation: Animación de objetos en 2D
- Java Telephony: Integración con telefonía
- Java Share: Interacción entre aplicaciones multiusuario
- Java 3D: Gráficos 3D y su manipulación

f. Seguro

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el *casting* implícito que hacen los compiladores de los lenguajes tradicionales, se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un navegador, combinamos las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El lenguaje C por ejemplo tiene lagunas de seguridad importantes, como son los errores de alineación. Los programadores de C utilizan punteros en conjunción con operaciones aritméticas permitiendo al programador que un puntero haga referencia a un lugar conocido de la memoria y pueda sumar (o restar) algún valor, para referirse a otro lugar de la memoria. Si otros programadores conocen nuestras estructuras de datos pueden extraer información confidencial de nuestro sistema. Con un lenguaje como C, se pueden tomar números enteros aleatorios y convertirlos en punteros para luego acceder a la memoria:

```
printf( "Escribe un valor entero: " );
scanf( "% u",&p untero );
printf( "Cadena de memoria: % s\n", puntero );
```

Otra laguna de seguridad u otro tipo de ataque es el **caballo de Troya**. Se presenta un programa como una utilidad, resultando tener una funcionalidad destructiva. Por ejemplo, en UNIX se visualiza el contenido de un directorio con el comando ls. Si un programador deja un comando destructivo bajo esta referencia, se puede correr el riesgo de ejecutar un código malicioso, aunque el comando siga cumpliendo la funcionalidad que le supone, después de lanzar su carga destructiva. Por ejemplo, después de que el caballo de Troya haya enviado por correo el **/etc/shadow** a su creador, ejecuta la funcionalidad de ls presentando el contenido del directorio. Se notará un retardo, pero nada inusual.

El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de *byte-codes* que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto.

Si los *byte-codes* pasan la verificación sin generar ningún mensaje de error, entonces sabemos que:

- El código no produce desbordamiento de operandos en la pila.
- El tipo de los parámetros de todos los códigos de operación son conocidos y correctos.
- No ha ocurrido ninguna conversión ilegal de datos, tal como convertir enteros en punteros.
- El acceso a los campos de un objeto se sabe que es legal: *public*, *private*, *protected*.
- No hay ningún intento de violar las reglas de acceso y seguridad establecidas.

El Cargador de clases también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de ficheros local del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo caballo de Troya; ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres a la que hace referencia. Esto imposibilita que una clase suplante a una predefinida.

En resumen, las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o de sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del *kernel* del sistema operativo. Además, para evitar modificaciones por parte de los *crackers* de la red, implementa un método ultraseguro de autentificación por clave pública. El Cargador de clases puede verificar una firma digital antes de realizar la instancia de un objeto. Por tanto, ningún objeto se crea y almacena en memoria, sin que se validen los privilegios de acceso. Es decir, la seguridad se integra en el momento de compilación, con el nivel de detalle y de privilegio que sea necesario.

Dada, pues la concepción del lenguaje y si todos los elementos se mantienen dentro del estándar marcado por Sun, no hay peligro. Java imposibilita, también, abrir algún fichero de la máquina local (siempre que se realizan operaciones con archivos, estas trabajan sobre el disco duro de la máquina de donde partió el **applet**), no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra. Además, los intérpretes que incorporan los navegadores de la web son aún más restrictivos. Bajo estas condiciones (y dentro de la filosofía de que el único ordenador seguro es el que está apagado, desenchufado, dentro de una cámara acorazada en un búnker y rodeado por mil soldados de los cuerpos especiales del ejército), se puede considerar que Java es un lenguaje seguro y que los **applets** están libres de virus.

Respecto a la seguridad del código fuente, JDK proporciona un desemsamblador de *byte-code*, que permite que cualquier programa pueda ser convertido a código fuente, lo que para el programador significa una vulnerabilidad total a su código. Utilizando Java no se obtiene el código fuente original, pero sí desmonta el programa mostrando el algoritmo que se utiliza, que es lo realmente interesante. La protección de los programadores ante esto es utilizar llamadas a programas nativos, externos (incluso en **C o C++**) de forma que no sea descompilable todo el código; aunque así se pierda portabilidad. Esta es otra de las cuestiones que Java tiene pendientes.

g. Portable

Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario, a través de un sistema abstracto de ventanas de forma que estas puedan ser implantadas en entornos Unix, Pc o Mac.

h. Interpretado

El intérprete Java (sistema *run-time*) puede ejecutar directamente el código-objeto. Enlazar (*linkar*) un programa normalmente consume menos recursos que compilarlo, por lo que los desarrolladores con Java pasarán más tiempo desarrollando y esperando menos por el ordenador. No obstante, el compilador actual del JDK es bastante lento. Por ahora, que todavía no hay compiladores específicos de Java para las diversas plataformas, este es más lento que otros lenguajes de programación, como **C++**; puesto que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional.

Se dice que Java es de 10 a 30 veces más lento que C, y que tampoco existen en Java proyectos de gran envergadura como en otros lenguajes. La verdad es que ya hay comparaciones ventajosas entre Java y el resto de los lenguajes de programación, y una ingente cantidad de folletos electrónicos que supuran fanatismo en favor y en contra de los distintos lenguajes contendientes con Java. Lo que se suele dejar de lado en todo esto, es que primero habría que decidir hasta que punto Java, un lenguaje en pleno desarrollo, está maduro como lenguaje de programación para ser comparado con otros; como por ejemplo con Smalltalk, que lleva más de 20 de desarrollo.

La verdad es que Java para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado como compilado. Y esto no es ningún contrasentido, debido a que el código fuente escrito con cualquier editor se compila generando el ByteCode. Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones de las máquinas propias de cada plataforma. El ByteCode corresponde al 80 % de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma. Para ello hace falta el *run-time*, que sí es completamente dependiente de la máquina y del sistema operativo que interpreta dinámicamente el ByteCode y, además, añade el 20 % de instrucciones que faltan para su ejecución. Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el *run-time* correspondiente al sistema operativo utilizado.

No obstante, este panorama está cambiando a pasos agigantados, y aunque Java sigue siendo básicamente un lenguaje interpretado, la situación se acerca mucho a la de los programas compilados, sobre todo en lo que se refiere a la rapidez en la ejecución del código. Para comprender el funcionamiento de HotSpot, que es el último lanzamiento que hace Sun y que promete interpretar los ByteCodes más rápido que un programa compilado, las siguientes imágenes muestran cómo actúa el sistema *run-time* en los diversos casos que hasta ahora pueden darse.

La imagen de la Figura 1.5 muestra las acciones correspondientes a un compilador tradicional. El compilador traslada las sentencias escritas en lenguaje de alto nivel a múltiples instrucciones, que luego son enlazadas junto con el resultado de múltiples compilaciones previas que han dado origen a librerías, y uniendo todo ello, se genera un programa ejecutable.

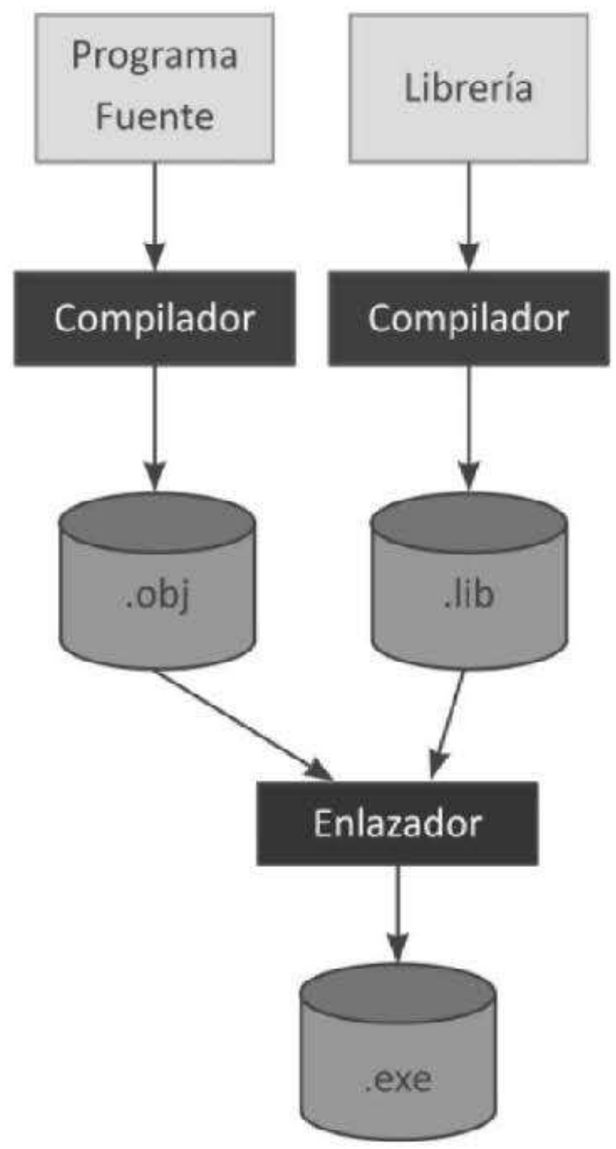

Fig. 1.5 Compilador tradicional

La imagen de la Figura 1.6 muestra la forma de actuar de un intérprete. Básicamente es un enorme bucle, en el cual se va leyendo o recogiendo cada una de las instrucciones del programa fuente que se desea ejecutar; se analiza, divide y ejecuta. Luego se va a recoger la siguiente instrucción que se debe interpretar y se continúa con este proceso hasta que se terminan las instrucciones o hasta que alguna contenga la orden de detener la ejecución de las instrucciones que componen el programa fuente.

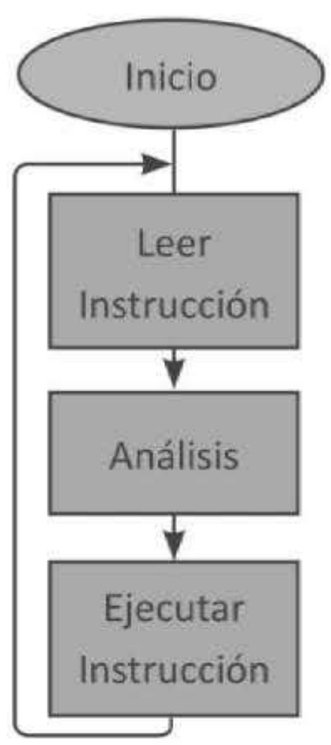

Fig. 1.6 Forma de actuar de un intérprete

La imagen de la Figura 1.7, muestra un tipo más eficiente que el anterior, el intérprete de ByteCodes, que fue popularizado hace más de veinte años por la Universidad de California al crear el UCSD Pascal. En este caso, el intérprete trabaja sobre instrucciones que ya han sido trasladadas a un código intermedio en un paso anterior. Así aunque se ejecute en un bucle, se elimina la necesidad de analizar cada una de las instrucciones que componen el programa fuente, porque ya lo han sido en el paso previo. Este es el sistema que Java utiliza, y la Máquina Virtual Java es un ejemplo de este tipo de intérprete. No obstante, sigue siendo lento aunque se obtenga un código independiente de plataforma muy compacto, que puede ser ejecutado en cualquier ordenador que disponga de una máquina virtual capaz de interpretar los ByteCodes trasladados.

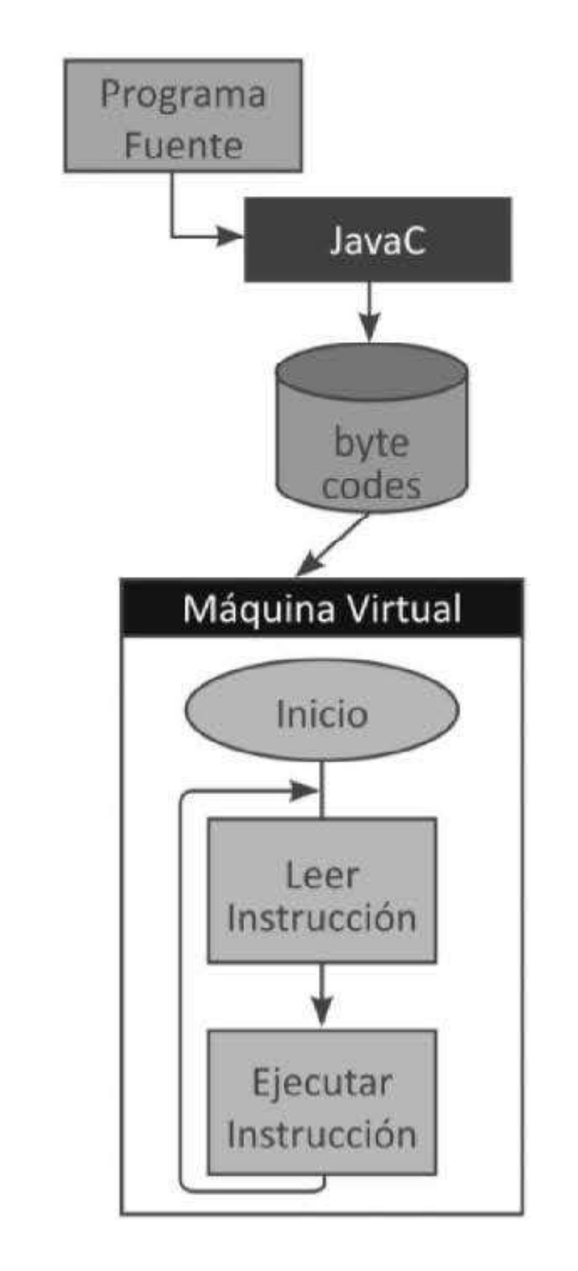

Fig. 1.7 Tipo de intérprete más eficiente

Un paso adelante en el rendimiento del código Java lo han representado los compiladores **Just-In-Time**, que compilan el código convirtiéndolo a código máquina antes de ejecutarlo. Es decir, un compilador JIT va trasladando los ByteCodes al código máquina de la plataforma según los va leyendo y realizando un cierto grado de optimización. El resultado es que cuando el programa se ejecute, habrá partes que no; de forma que no serán compiladas, y el compilador JIT no perderá el tiempo en optimizar códigos que nunca se van a ejecutar. No obstante, los compiladores JIT no pueden realizar demasiadas optimizaciones, ya que hay códigos que ellos no ven, así que aunque siempre son capaces de optimizar la parte del código de inicialización de un programa, hay otras partes que deben ser optimizadas según se van cargando, con lo cual existe una cierta cantidad de tiempo que inevitablemente ha de perderse.

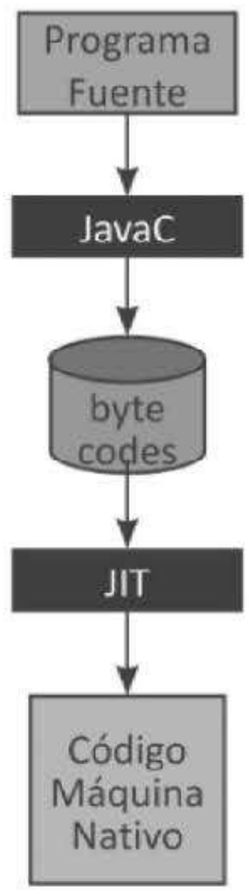

Fig. 1.8 Compilador Just-In-Time

Y, finalmente, se presenta la última tendencia en lo que a compilación e intérpretes se refiere. Lo último en que trabaja Sun es HotSpot, una herramienta que incluye un compilador dinámico y una máquina virtual para interpretar los ByteCodes, tal como se muestra en la Figura 1.9. Cuando se cargan los ByteCodes producidos por el compilador por primera vez, estos son interpretados en la máquina virtual. Cuando ya están en ejecución, el *profiler* mantiene información sobre el rendimiento y selecciona el método sobre el que se va a realizar la compilación. Los métodos ya compilados se almacenan en un caché en código máquina nativo. Cuando un método es invocado, esta versión en código máquina es la que se utiliza, en caso de que exista; de ocurrir lo contrario, los ByteCodes son reinterpretados. La función control que muestra el diagrama es como un salto indirecto a través de la memoria que apunta tanto al código máquina como al interpretado, aunque Sun no ha proporcionado muchos detalles sobre este extremo.

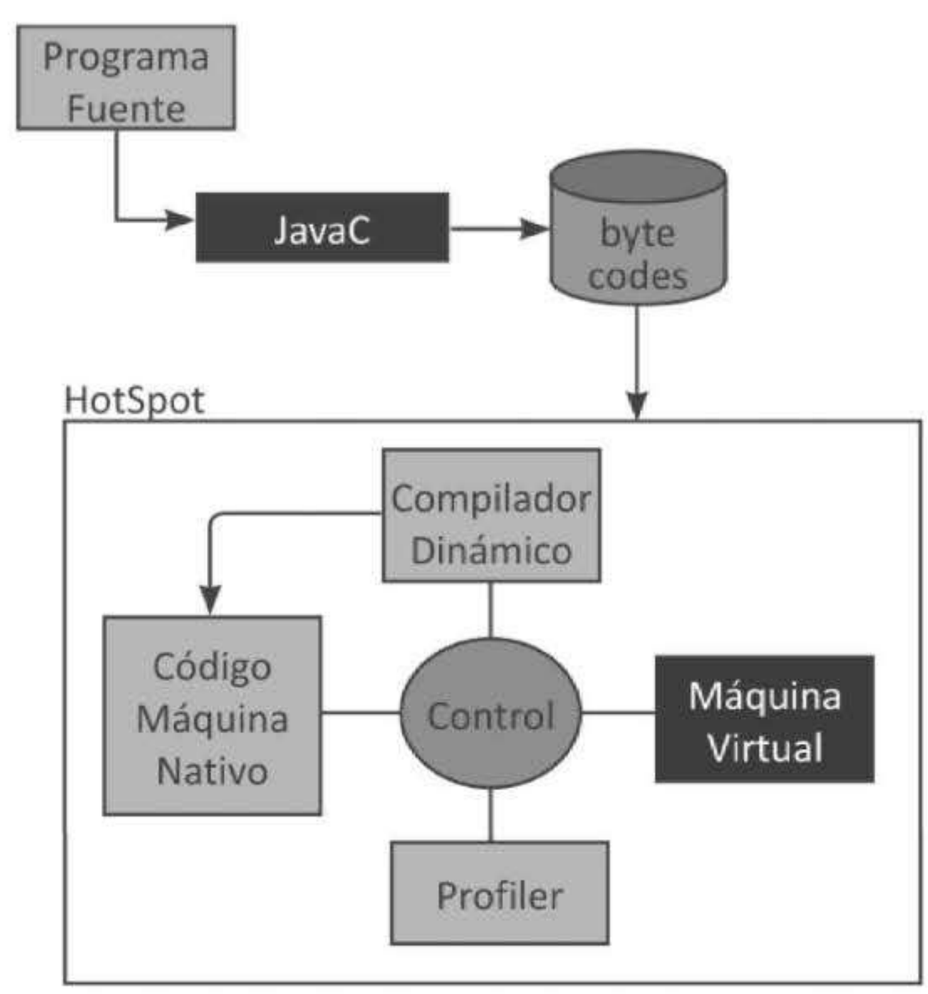

Fig. 1.9 Compilador – Intérprete HotSpot

i. Multihilo

Al ser multihilo (*multithreaded*), Java permite muchas actividades simultáneas en un programa. Los hilos son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar estos hilos construidos en el mismo lenguaje, son más fáciles de usar y más robustos que sus homólogos en los lenguajes tradicionales.

El beneficio de ser multihilo consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.) de la plataforma, aún supera a los entornos de flujo único de programa (*single-threaded*) tanto en facilidad de desarrollo como en rendimiento.

Cualquiera que haya utilizado la tecnología de navegación concurrente, sabe lo frustrante que puede ser esperar por una imagen que se está trayendo de un sitio interesante desde la red. En Java, las imágenes se pueden ir trayendo en un hilo de ejecución independiente, permitiendo que el usuario pueda acceder a la información de la página sin tener que esperar por el navegador.

j. Dinámico

Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el mismo tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán la ejecución de las aplicaciones actuales (siempre que mantengan el API anterior).

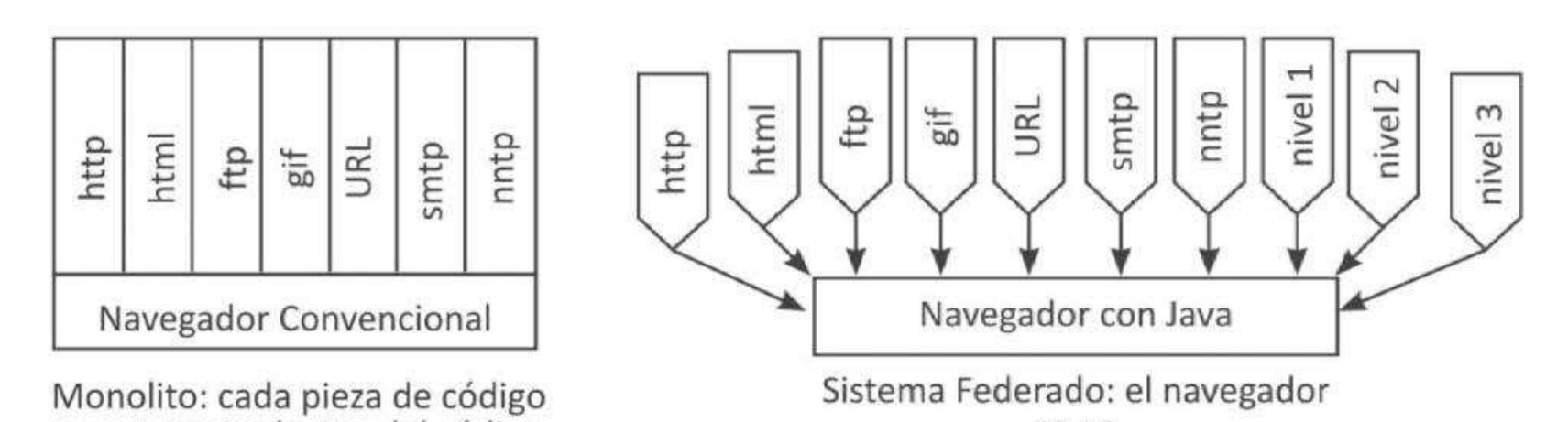

Fig. 1.10 Beneficios de Java

Java también simplifica el uso de protocolos nuevos o actualizados. Si su sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, Java es capaz de traer automáticamente cualquier pieza que el sistema necesite para funcionar.

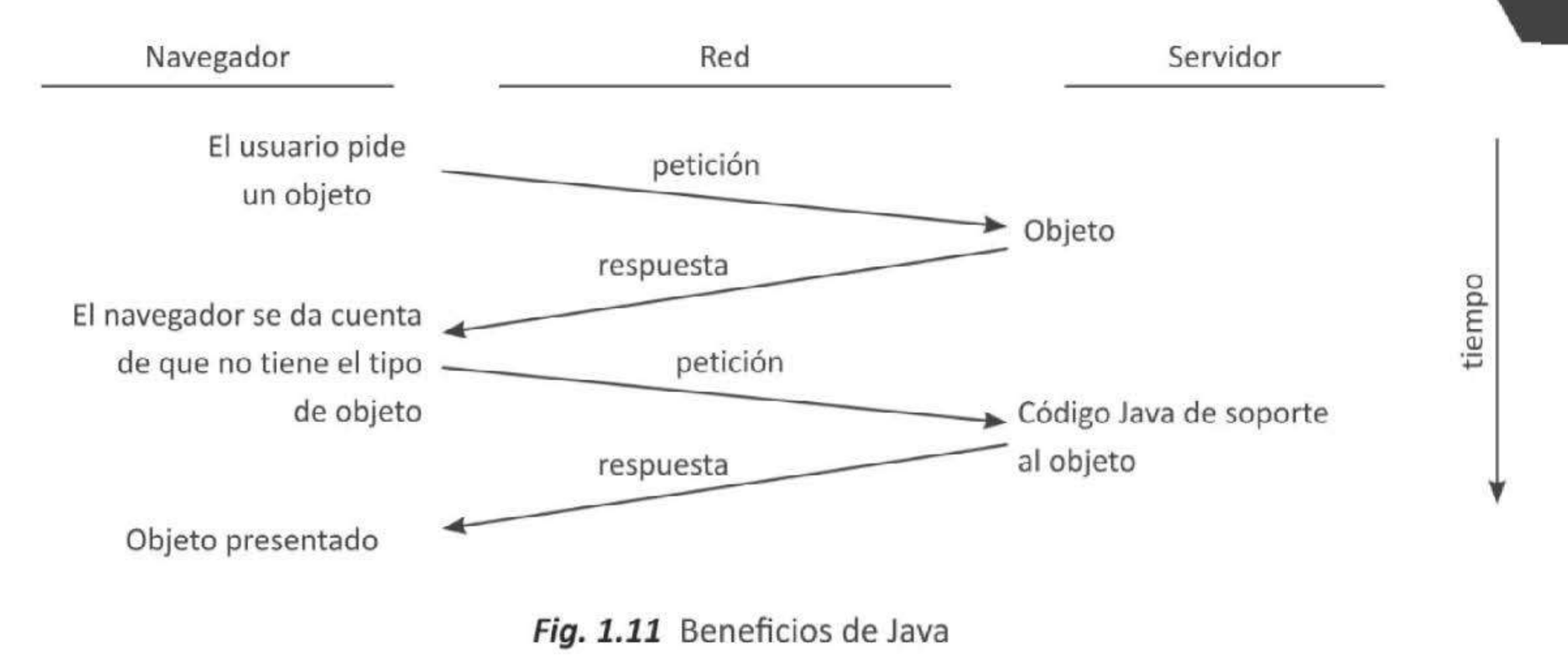

Fig. 1.11 Beneficios de Java

Para evitar que los módulos de ByteCode o los objetos o nuevas clases no se traigan de la red cada vez que se necesiten, Java implementa las opciones de persistencia, para que no se eliminen cuando se limpie la caché de la máquina.

1.2.3 Ejecución de programas Java

- **Esquema de ejecución de una aplicación**

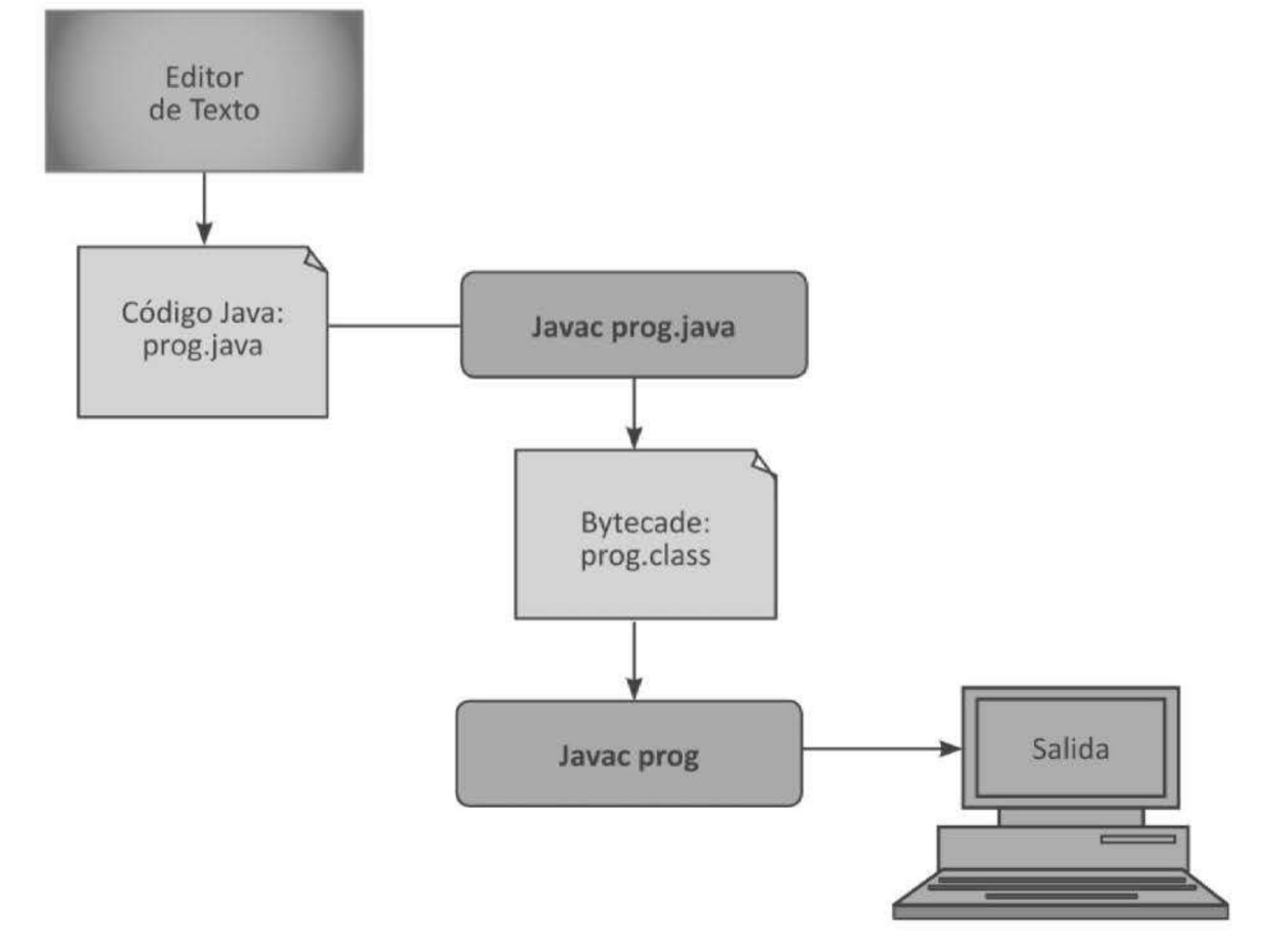

Fig. 1.12 Ejecución de una aplicación

- **Esquema de una ejecución de un Applet**

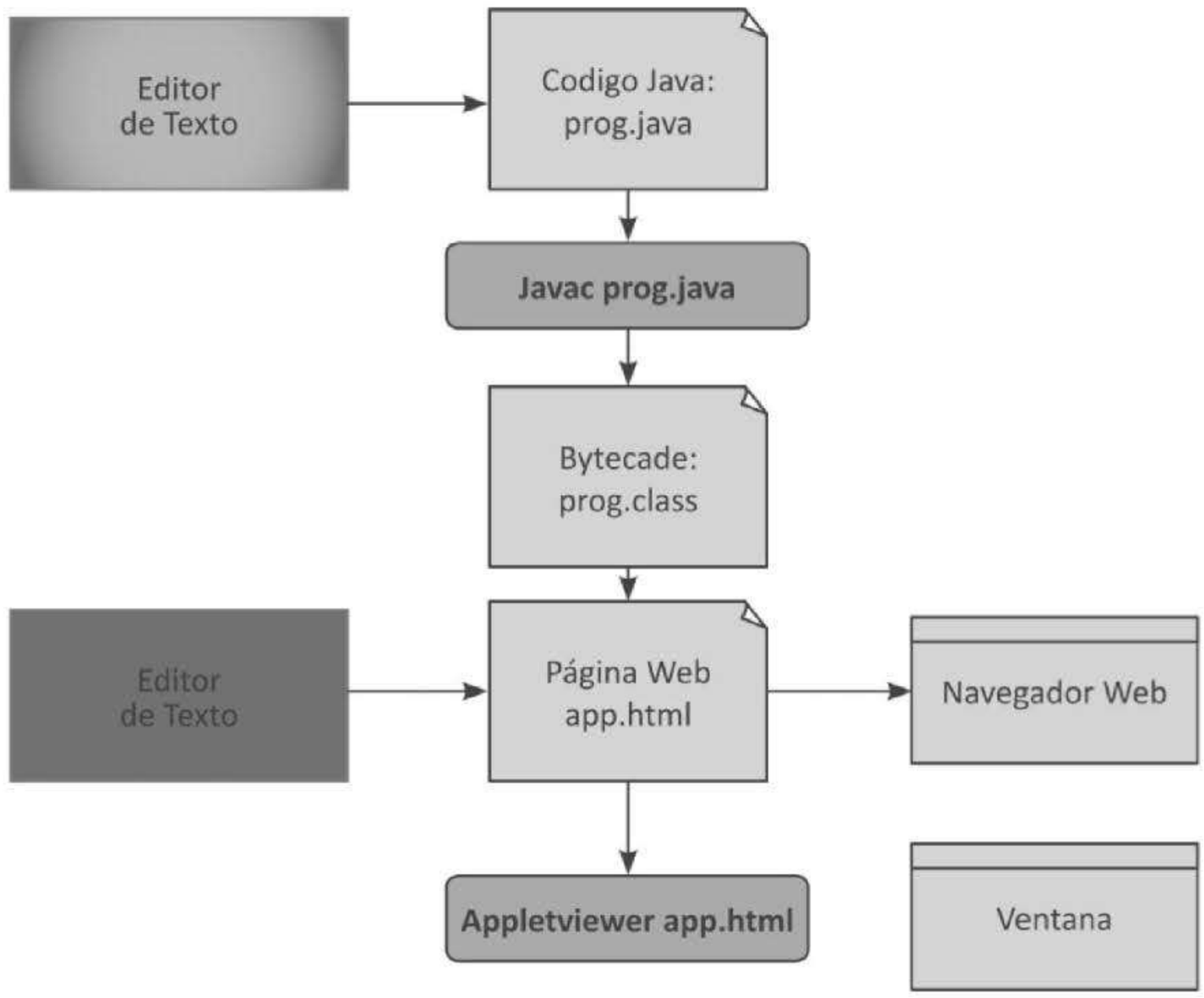

Fig. 1.13 Ejecución de Applets

1.3 PRODUCTOS & TECNOLOGÍA RELACIONADA CON JAVA

1.3.1 J2SE (JAVA2 STANDARD EDITION)

La plataforma de Java 2, Edición Estándar (J2SE) proporciona un entorno completo para el desarrollo de aplicaciones de escritorio y servidor, además para su despliegue.

Hay dos productos principales en la plataforma de J2SE:

- JRE – Java2 *Runtime Enviroment* (Entorno de Ejecución)
- JDK – Java2 *Development Kip* (kit de desarrollo)

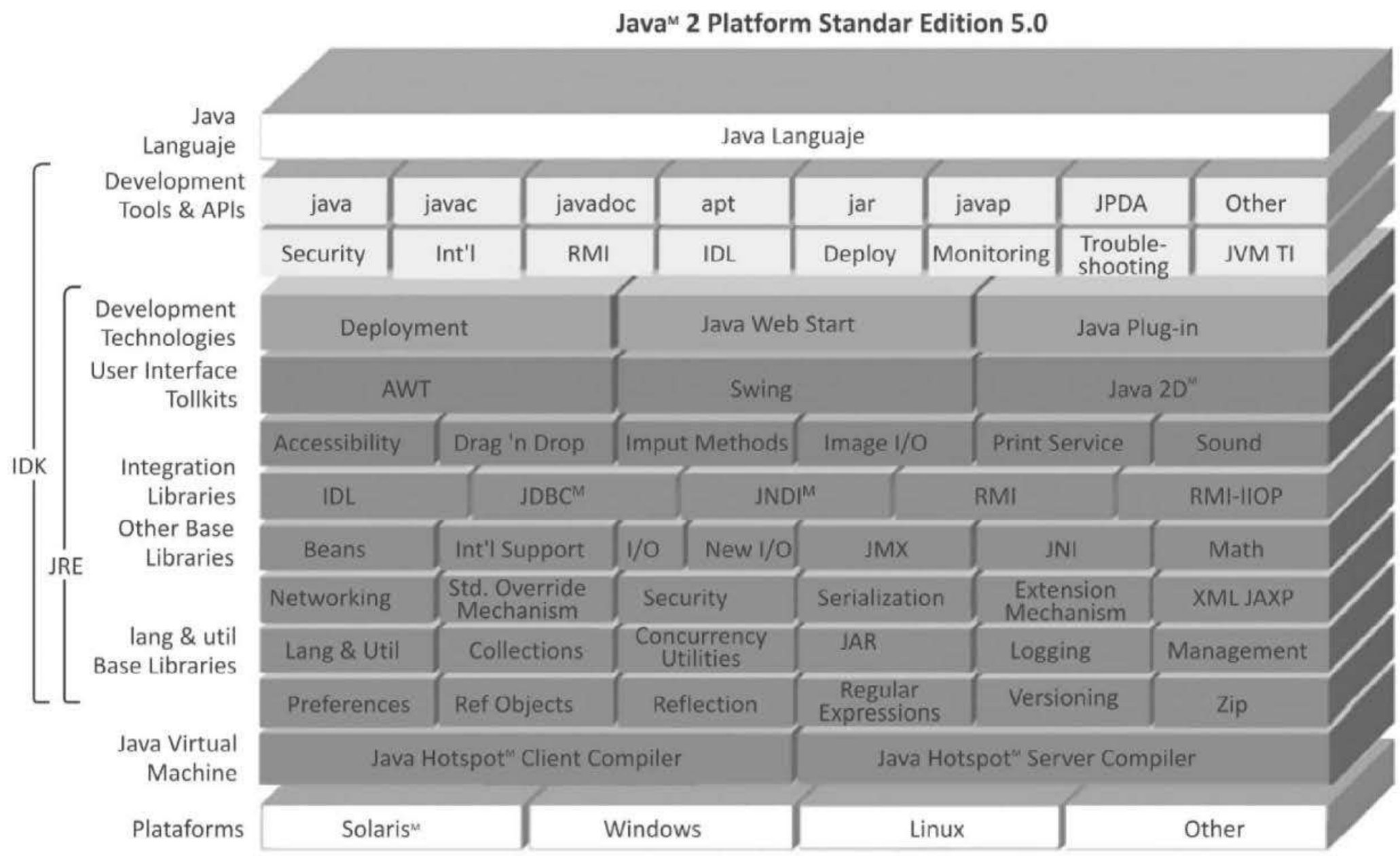

Fig. 1.14 Plataforma de Java2 Edición Estándar

a. JRE – Java2 *Runtime Enviroment*

El JRE provee las APIs de Java, la máquina virtual de Java y otros componentes necesarios para ejecutar los applets y aplicaciones desarrollados en Java. Es también la base para las tecnologías en la plataforma de Java 2, Edición Empresarial (JEE) para el desarrollo y despliegue de software empresarial. El JRE no contiene las herramientas y las utilidades tales como compiladores o depuradores para el desarrollo de *applets* y aplicaciones.

b. JDK [1]– Java2 *Development Kip*

El JDK incluye el JRE más las herramientas de desarrollo de líneas de comandos, tales como compiladores y depuradores que son necesarios para desarrollar *applets* y aplicaciones.

1.3.2 J2EE (Java2 Enterprise Edition)

La plataforma de Java 2, Edición Empresarial (JEE) define el estándar para el desarrollo de aplicaciones distribuidas basadas en componentes. JEE simplifica la construcción de aplicaciones empresariales que son portables, escalables y que se integran fácilmente con aplicaciones y **data legacy**. JEE es también una plataforma para construir y usar servicios web. Incorpora servicios web estándares, ello significa que dichos servicios en un entorno de J2EE pueden interactuar con otros que no están basados en J2EE, como **.net**.

1.3.3 J2ME (Java2 Micro Edition)

La plataforma de Java 2, Micro Edition proporciona un entorno robusto y flexible para aplicaciones que funcionan en dispositivos, tales como teléfonos móviles, PDAs y cajas Set-Top de TV; además de una amplia gama de dispositivos. J2ME incluye la máquina virtual de Java y un conjunto de APIs estándar definidas por *Java Community Process*, por grupos de expertos que incluyen a fabricantes de dispositivos, vendedores de software y abastecedores de servicios principales.

1.3.4 Java Card Technology

La tecnología Java Card permite a tarjetas inteligentes y otros dispositivos con memoria muy limitada ejecutar pequeñas aplicaciones, llamar a *applets* que emplean tecnología Java. Los proveedores de tarjetas inteligentes fabrican sobre una plataforma de ejecución segura e interoperable que pueda almacenar y actualizar múltiples aplicaciones en un solo dispositivo. La tecnología Java Card es compatible con estándares existentes de tarjetas inteligentes

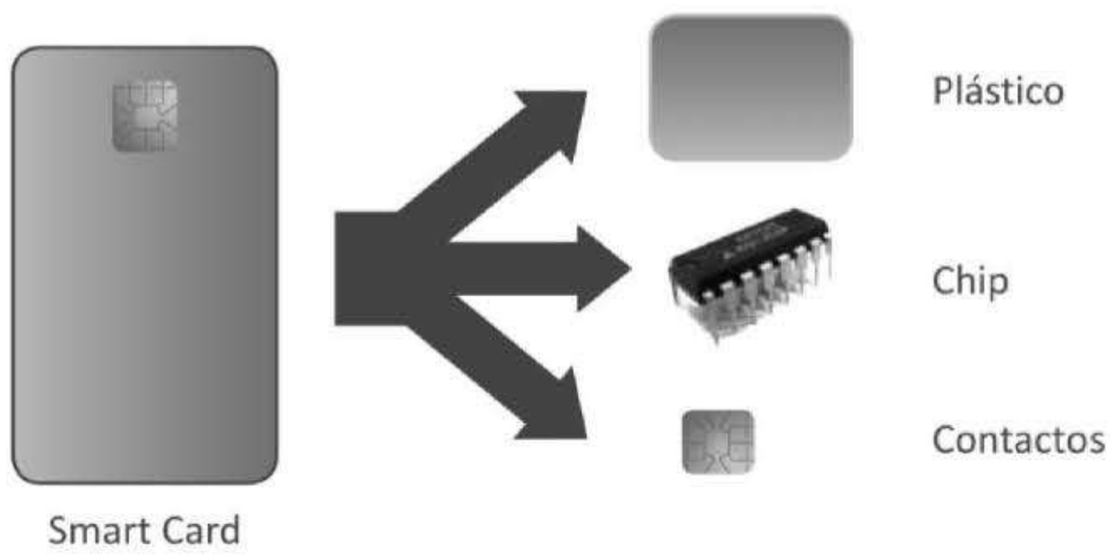

Fig. 1.15 Tarjeta inteligente

Podemos mencionar las siguientes ventajas:

- Código independiente de la plataforma y tarjeta
- Multiaplicación
- Instalación de applets dinámica
- Flexibilidad por el enfoque OO
- Compatibilidad ISO7816

1.3.5 Java Web Services

La plataforma Java2, plataforma *Erterprise Edition* proporciona los APIs y herramientas que se necesitan para un rápido diseño, desarrollo, prueba, despliegue a servicios y clientes web que interoperen entre ellos y funcionen en cualquier plataforma. Esta interoperabilidad es posible porque el intercambio de los datos entre las aplicaciones se realiza en documentos XML estandarizados.

1.3.6 Java Business Integration (JBI)

La especificación JBI es un estándar para lo que puede ser llamado un contenedor de contenedores. Describe la arquitectura y comportamiento de un *framework* que hospeda componentes a modo de *plug-ins*, y permite que interactúen usando mensajes WSDL (*Web Services Description Language*), y el modelo de servicios. El objetivo es permitir a los usuarios obtener los componentes que ofrecen las funciones necesarias para proveer una solución de integración, sin estar vinculado a la particular implementación de un vendedor concreto. Los componentes serán portables entre implementaciones JBI.

1.3.7 Java XML

XML (*Extensible Markup Language*) es un modo flexible de crear formatos e información común y compartir tanto el formato como la data en la *World Wide Web*, intranets, y cualquier otro sitio.

XML puede ser usado por una persona, grupos de personas, o compañías que deseen compartir información en forma consistente. Se basa en un texto plano para representar los datos, siendo esto la clave tecnológica para el desarrollo de servicios web.

Software e instalación

CAPÍTULO 2

Capacidad: En este capítulo veremos de donde obtener el software y el proceso de instalación.

2.1 SOFTWARE

Utilizaremos la edición estándar. Los datos para descargar el software son los siguientes:

Página web	http://java.sun.com/j2se/1.5.0/download.jsp
Nombre del archivo	jdk-1_5_0_05-windows-i586-p.exe
Versión	5.0
Sistemas operativo	Windows
Tamaño	58,866 Kb

En esta página se encontrará el enlace para descargar el JDK 5.0, también muestra un enlace para descargar la herramienta **netBeans** que incorpora el JDK 5.0.

En la Figura 2.1 tenemos una imagen de la página web, se observa marcado el enlace para descargar el JDK.

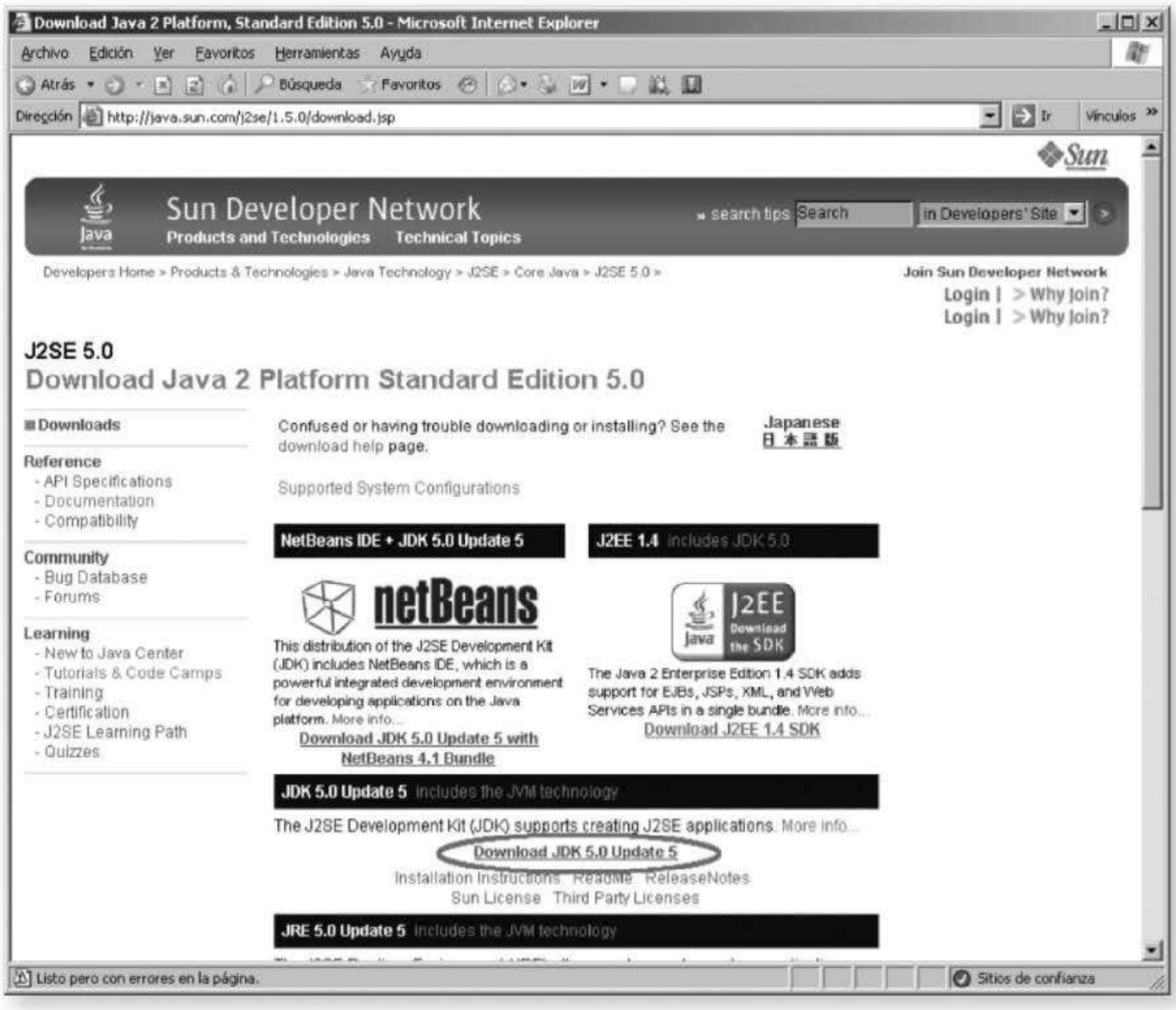

Fig. 2.1 Página web para bajar JDK 5.0

2.2 INSTALACIÓN DEL JDK

Después de descargar el JDK debe seguir los siguientes pasos para realizar la instalación:

1. Ejecutar el archivo y proceder a su instalación, inicialmente tenemos la ventana de Bienvenida mientras se prepara la instalación, tal como se ilustra en la Figura 2.2.

Fig. 2.2 Ventana de Bienvenida

2. Luego tenemos las condiciones de licenciamiento, solo acepte y haga clic en el botón **Next**.

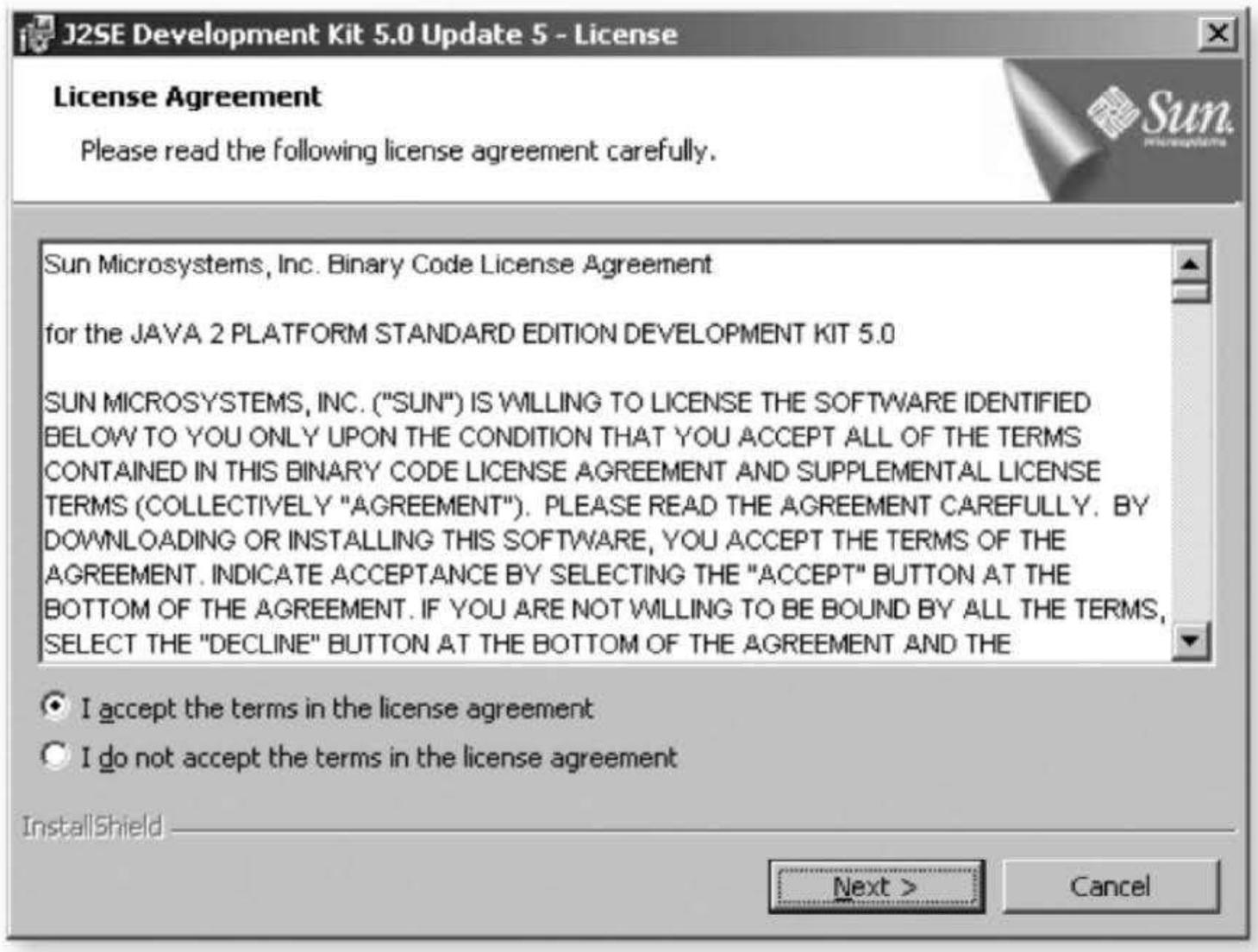

Fig. 2.3 Ventana de Licenciamiento

3. En el siguiente paso debemos seleccionar los productos a instalar y el directorio destino, por defecto están marcados todos los productos, déjelo así y haga clic en el botón **Next**.

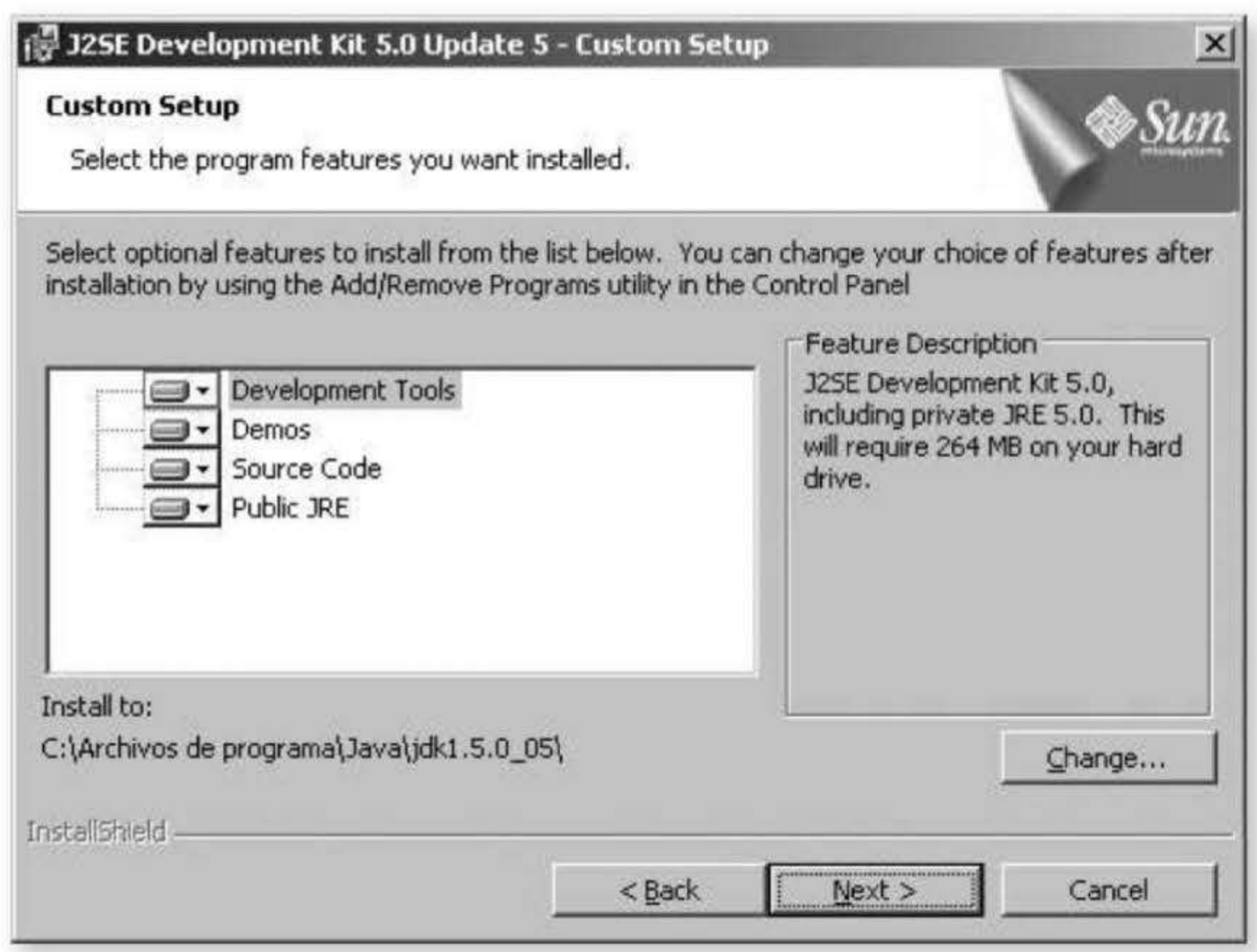

Fig. 2.4 Productos a instalar y directorio destino

4. A continuación se inicia el proceso de instalación del conjunto de herramientas de desarrollo, esto puede tardar varios minutos.

Fig. 2.5 Instalación de las herramientas de desarrollo

5. En la siguiente ventana tenemos la posibilidad de cambiar la ubicación de algunos archivos de instalación o dejar las opciones sugeridas; luego haga clic en el botón **Next**.

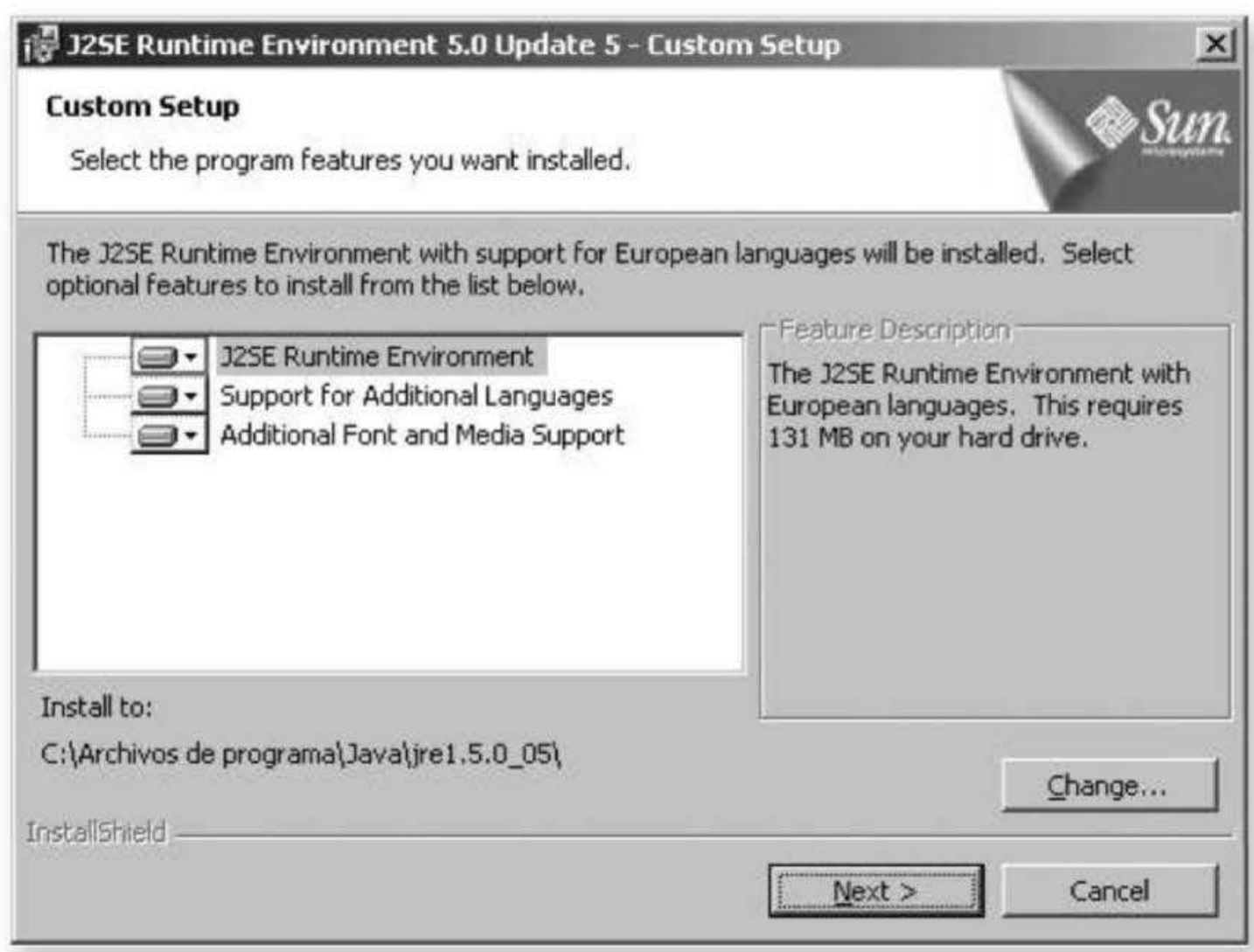

Fig. 2.6 Ubicación de algunos archivos

6. En la siguiente ventana seleccione el navegador que utilizará, luego haga clic en el botón **Next**.

Fig. 2.7 Selección del navegador

7. En la siguiente ventana se puede apreciar la instalación del **run-time**, esto tomará algunos minutos.

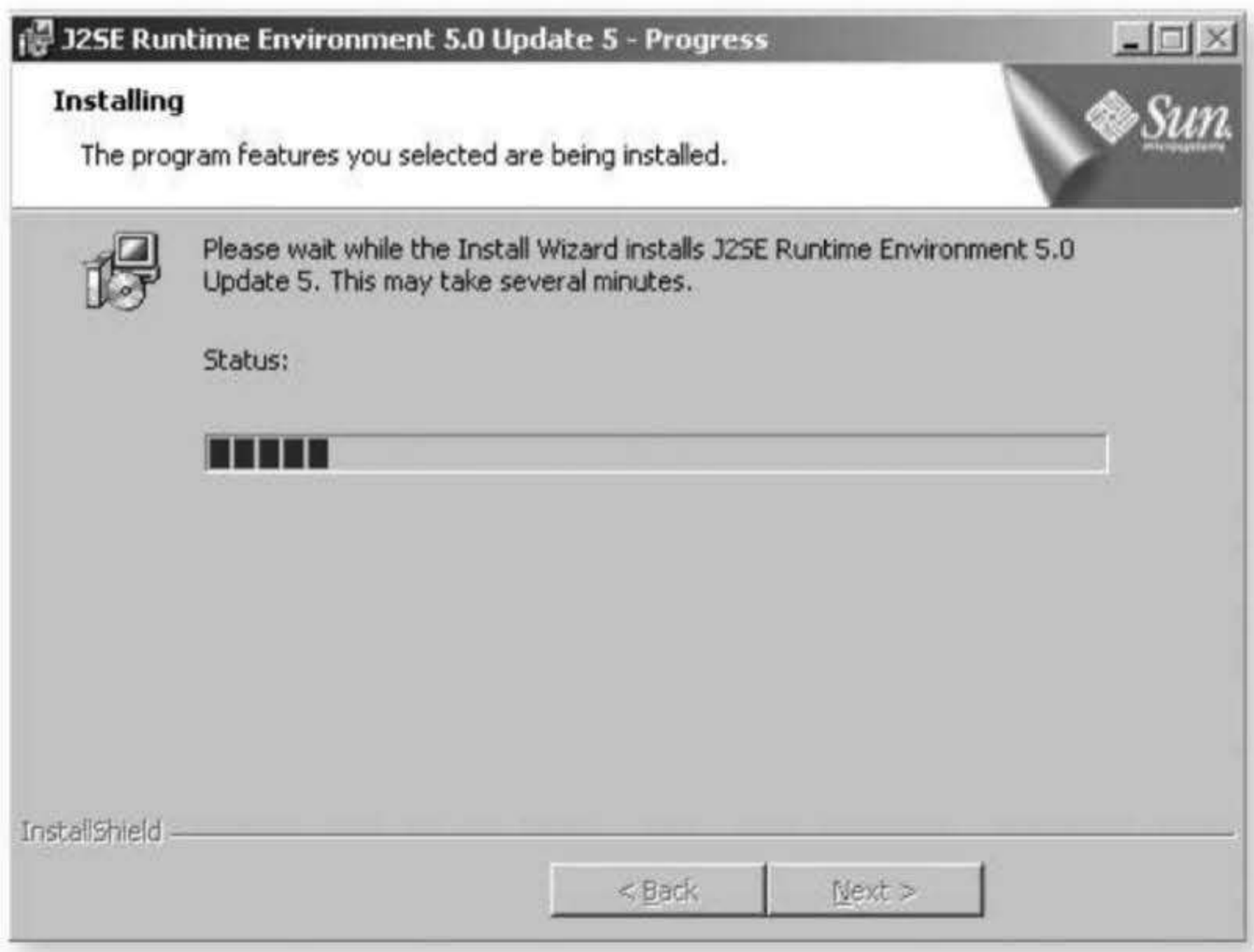

Fig. 2.8 Instalación del *run-time*

8. Finalmente llegará a la finalización de la instalación, debe hacer clic en el botón **Finish**.

Fig. 2.9 Finalización de la instalación

Al finalizar el proceso de instalación se puede verificar que se han creado dos carpetas, tal como se observa en la Figura 2.10.

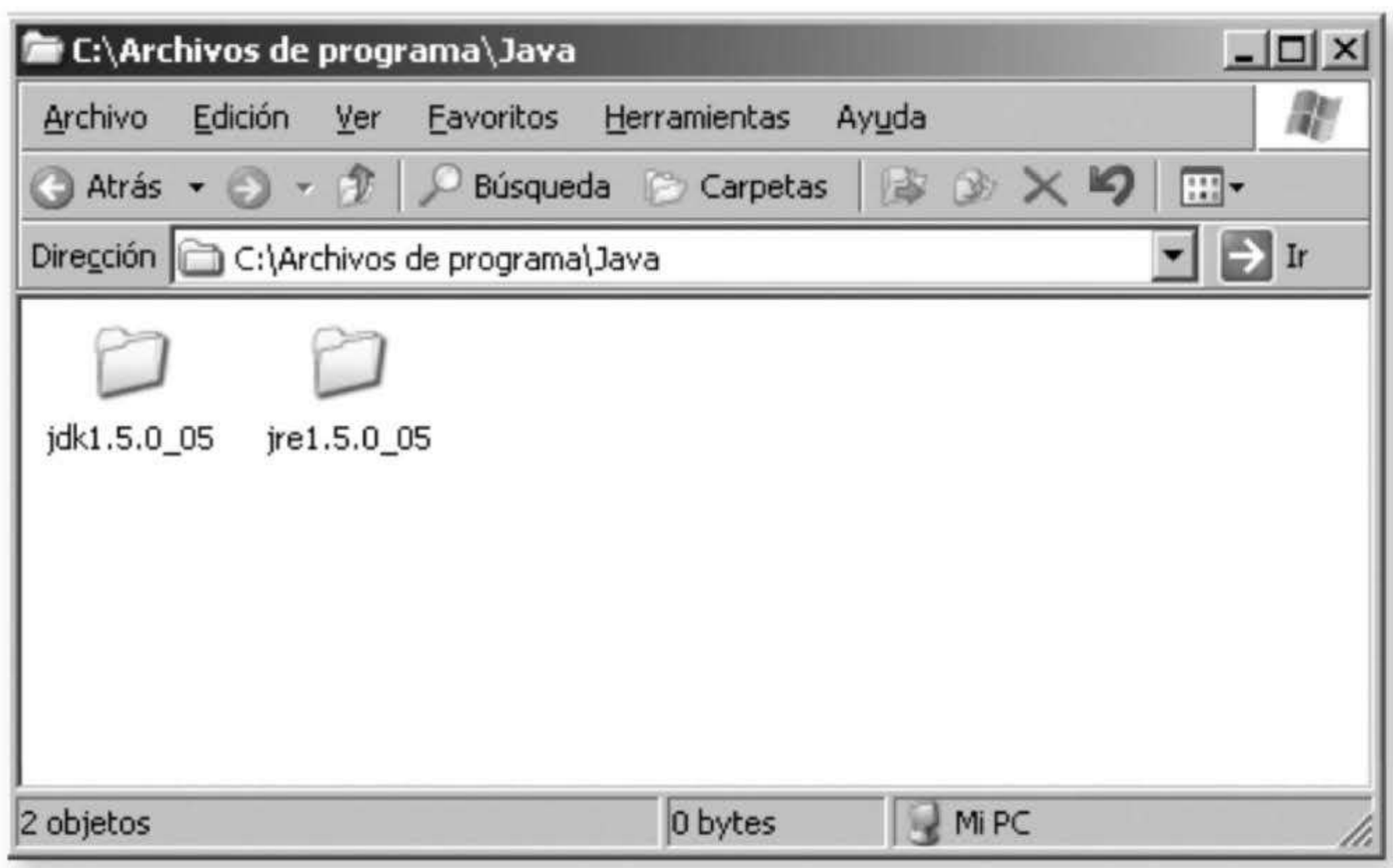

Fig. 2.10 Carpetas del producto

jdk1.5.0_05	Esta carpeta contiene las herramientas de desarrollo, por ejemplo el compilador de Java, el depurador, el *run-time* y ejemplos sobre varios temas del lenguaje Java.
jre1.5.0_05	Esta carpeta contiene las librerías, la maquina virtual y otros componentes para ejecutar *applets* y programas desarrollados en Java.

2.3 HERRAMIENTAS DEL JDK

Las herramientas del JDK se encuentran en la carpeta:

```
C:\Archivos de programa\Java\jdk1.5.0_05\bin; tal como se aprecia en la Figura
2.11.
```

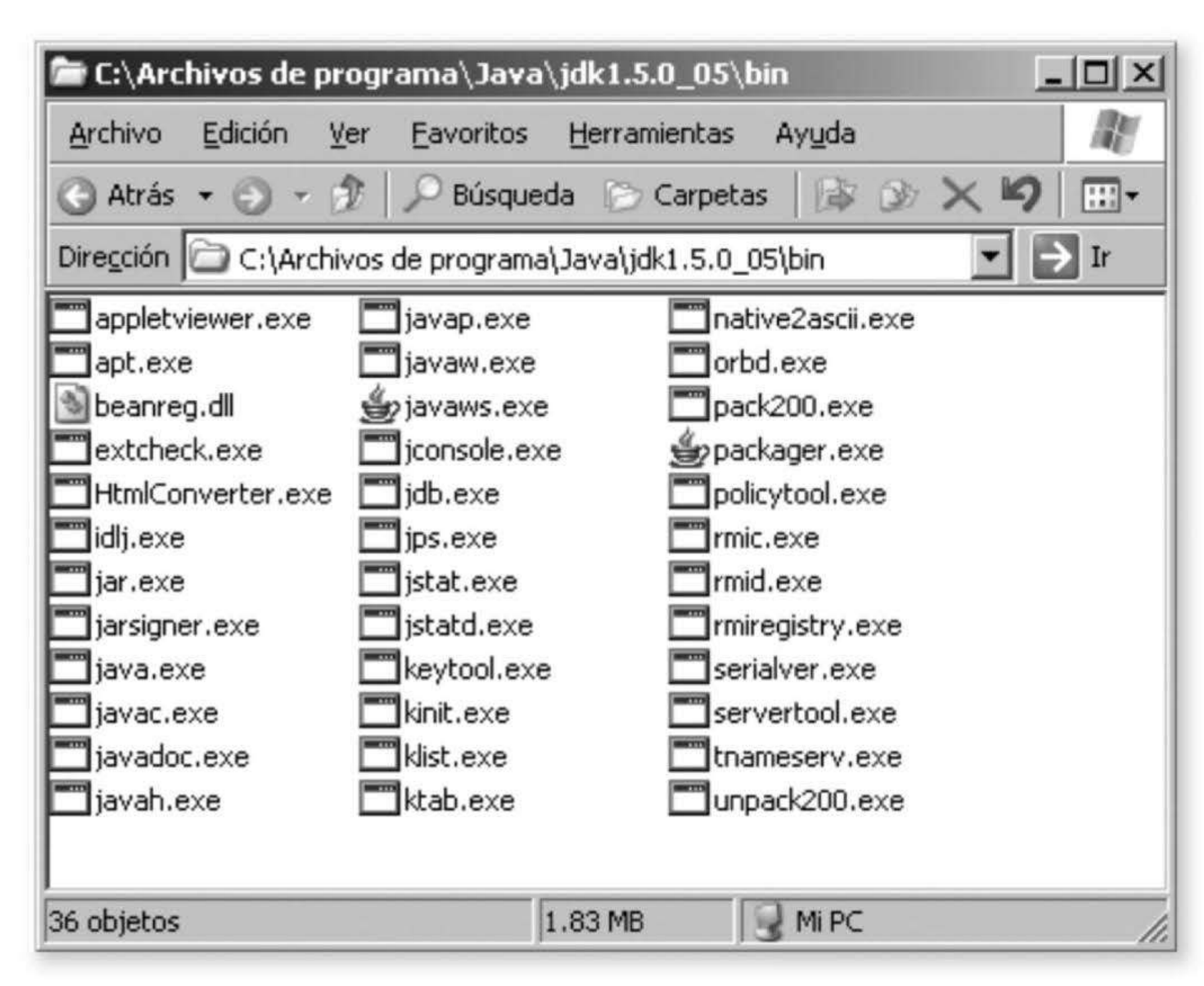

Fig. 2.11 Herramientas del JDK

2.3.1 Herramientas básicas del JDK

A continuación tenemos una breve descripción de las herramientas del JDK.

javac.exe	Compilador de los archivos fuentes de Java (**.java**) y genera los *bytecode* (**.class**).
java.exe	El programa Java es una interfaz simple a base de línea de comandos para acceder a la Máquina Virtual. Es el intérprete que ejecuta *bytecodes* creados por Java, su compilador.
javadoc.exe	Este programa permite documentar programas y paquetes. Genera documentos HTML.
apt.exe	Se trata de una utilidad de línea de comando para el proceso de anotación. Incluye un grupo de APIs reflexivas y brinda una infraestructura para procesar anotación de programas. Estas API's reflexivas proveen un tiempo de construcción, basadas en código fuente, vistas de solo lectura de la estructura del programa.
appletviewer.exe	Permite ejecutar *applets*.
jar.exe	Permite manejar archivos .jar. Varios *applets* y sus componentes pueden ser incluidos en un solo archivo **JAR**.
jdb.exe	Es un depurador para clases de Java. Está basado en una línea de comandos de texto y tiene una sintaxis de comandos como la del **dbx** en UNIX o depuradores **gdb**.
javah.exe	Genera archivos fuentes y de cabecera en C (archivos **.h** y **.c**) que describen las clases especificadas. Estos archivos de C proporcionan la información necesaria para implementar métodos nativos para las clases especificadas en C.
javap.exe	Desensambla el archivo de clases especificado por los nombres de clases en la línea de comandos e imprime una versión medianamente comprensible de esas clases.
extcheck.exe	Comprueba en un fichero **JAR** los conflictos en el título y la versión con cualquier extensión instalada en el software JDK.

2.3.2 Herramientas de seguridad

keytool.exe	La herramienta **keytool** se emplea para gestionar un almacén de claves (*keystore*), que es una base de datos protegida que contiene claves y certificados para un usuario o grupo de usuarios. El acceso al mismo está protegido por un sistema de contraseñas y además, cada clave privada puede protegerse con una contraseña individual.
jarsigner.exe	Esta herramienta se utiliza para firmar y validar la firma de archivos **JAR**. La herramienta jarsigner utiliza el almacén de claves que crea y gestiona el programa **keytool** para buscar las claves privadas y certificados que necesita para firmar archivos **JAR**. Como los accesos al almacén de claves y a las claves privadas están protegidos con contraseña, solo los usuarios que los conozcan podrán acceder a las claves para firmar archivos **JAR**.
policytool.exe	Esta herramienta se utiliza para crear y modificar los archivos de configuración de políticas. La herramienta tiene una interfaz gráfica cómoda para trabajar con estos ficheros, haciéndola preferible a la edición manual.

Estas herramientas de seguridad le ayudan a obtener, listar y manejar *tickets* Kerberos.

kinit.exe	Herramienta para obtener *tickets* Kerberos v5.
klist.exe	Herramienta de línea de comando para listar las entradas en el caché de credenciales y key **tab**.
ktab.exe	Herramienta de línea de comando que ayuda al usuario a manejar las entradas en la tabla de claves.

2.3.3 Herramientas de internacionalización

native2ascii.exe	Convierte ficheros que contienen codificaciones de caracteres no soportados a ficheros con codificaciones Latin-1 o Unicote.

2.3.4 Herramientas RMI (Remote Method Invocation)

Las siguientes herramientas se utilizan para desarrollar aplicaciones distribuidas con Java.

rmic.exe	Esta herramienta se utiliza para la generación de las clases que actuarán como **stub**, en el cliente, y esqueleto, en el servidor.
rmiregistry.exe	Crea e inicia un registro de objeto remoto en el puerto indicado de la máquina actual.
rmid.exe	Activación RMI del *system daemon*.
serialver.exe	Retorna el serialVersionUID para una o más clases de una manera adecuada para ser copiada en una clase envolvente.

2.4 Probando Java

2.4.1 Carpeta de trabajo

La carpeta de trabajo para realizar esta prueba es:

```
C:\java2n100\cap2
```

Y la carpeta donde están las herramientas de Java es:

```
C:\Archivos de programa\Java\jdk1.5.0_05\bin
```

2.4.2 Verificando la variable de entorno PATH

La variable de entorno PATH tiene las rutas donde el sistema operativo busca las aplicaciones cuando no se encuentra en la carpeta actual; por lo tanto debe contener la ruta donde se encuentran las herramientas de Java.

1. Cargar la consola de comandos, ejecute el comando **cmd** desde el dialogo **Ejecutar**.

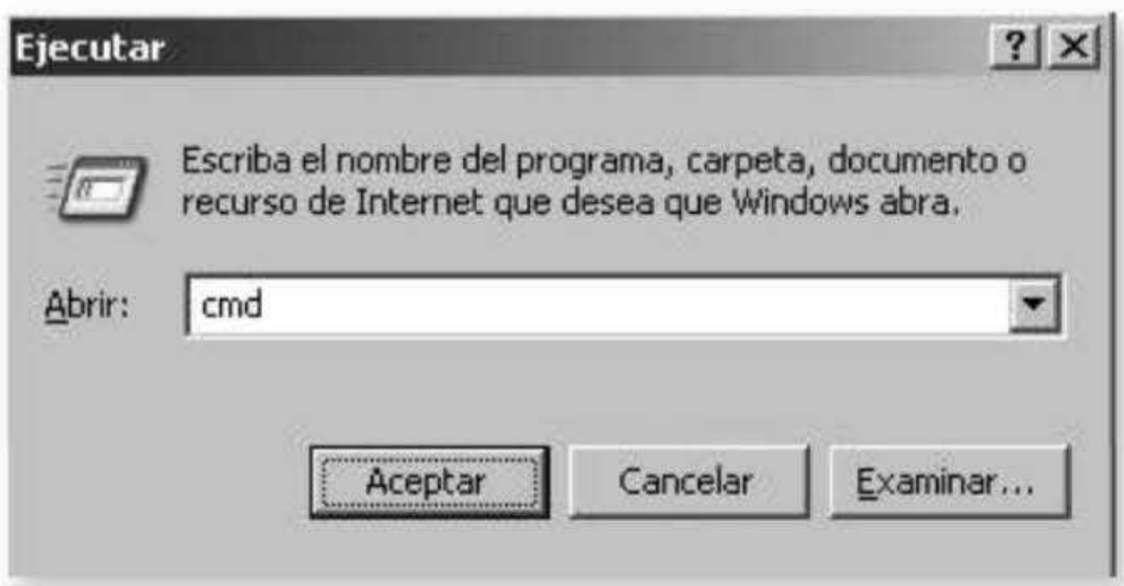

Fig. 2.12 Dialogo Ejecutar

2. En la consola de comando debe ubicarse la carpeta de trabajo.

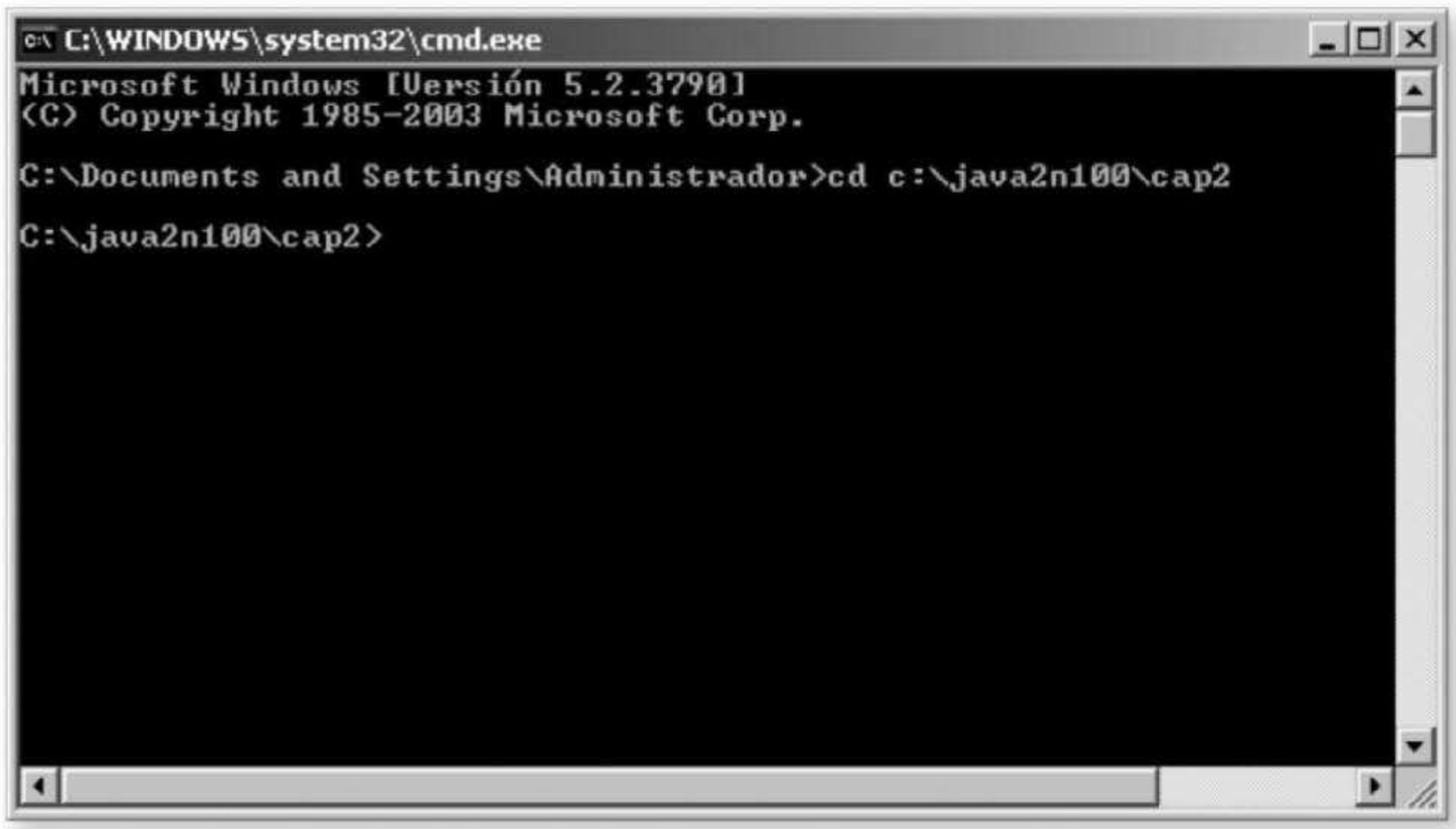

Fig. 2.13 Consola de comandos

3. Verificar si la variable de entorno PATH contiene la ruta de la carpeta **bin** de Java.

```
C:\java2n100\cap2>set path [Enter]
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\system32\WBEM;C:\Archivos-
de programa\Rational\common;C:\Archivos de programa\Microsoft SQL Ser-
ver\80\Tools\BINN;

PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
```

Observamos que no tiene la ruta.

4. Ahora debe agregar la ruta de la carpeta **bin** a la variable de entorno PATH.

```
C:\java2n100\cap2>set path=%path%;C:\Archivos de programa\Java\jdk1.5.0_05\
bin    [Enter]

C:\java2n100\cap2>set path [Enter]
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\system32\WBEM;C:\Archivos de
programa\Rational\common;C:\Archivos de programa\Microsoft SQL Server\80\
Tools\BINN;
;"c:\Archivos de programa\java\jdk1.5.0_05\bin"
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
```

Ahora estamos en condiciones de ejecutar nuestros primeros ejemplos.

2.5 EJECUTANDO EL PRIMER PROGRAMA

1. Usando el bloc de notas proceda a crear el siguiente programa y grábelo con el nombre **prog0201 java** en la carpeta **C:\java2n100\cap2**.

prog0201.java

```
public class  prog0201
{public static void main(String[] args)
{System.out.println("Java esta listo.");System.out.println("Ahora debo
ponerme las pilas.");} //main} // class
```

2 . Ahora debe compilar el programa y crear el *bytecode*.

```
C:\java2n100\cap2>javac prog0201.java  [Enter]
```

Puede verificar que se ha creado el programa **prog0201.class**.

3. Luego ejecute el programa.

```
C:\java2n100\cap2>java prog0201                [Enter]
Java esta listo.
Ahora debo ponerme las pilas.
```

Si no tuvo problemas logró ejecutar su primer programa en Java, pero veamos el primer problema que tiene, la variable PATH modificada solo es válida en la sesión de consola donde se modificó, es decir que si abre otra ventana de consola no podrá ejecutar programas Java, a menos que vuelva a configurar la variable PATH. Si cierra la consola, se perderá la configuración de la variable PATH.

2.6 CONFIGURANDO LA VARIABLE PATH

Esta variable de entorno puede ser configurada en forma personalizada para el usuario actual o a nivel de sistema para cualquier usuario.

1. Cargar la ventana de **Propiedades del sistema** desde el **Panel de control.**

Fig. 2.14 Ventana de Propiedades del sistema

2. Ubicarse en la ficha **Opciones avanzadas.**

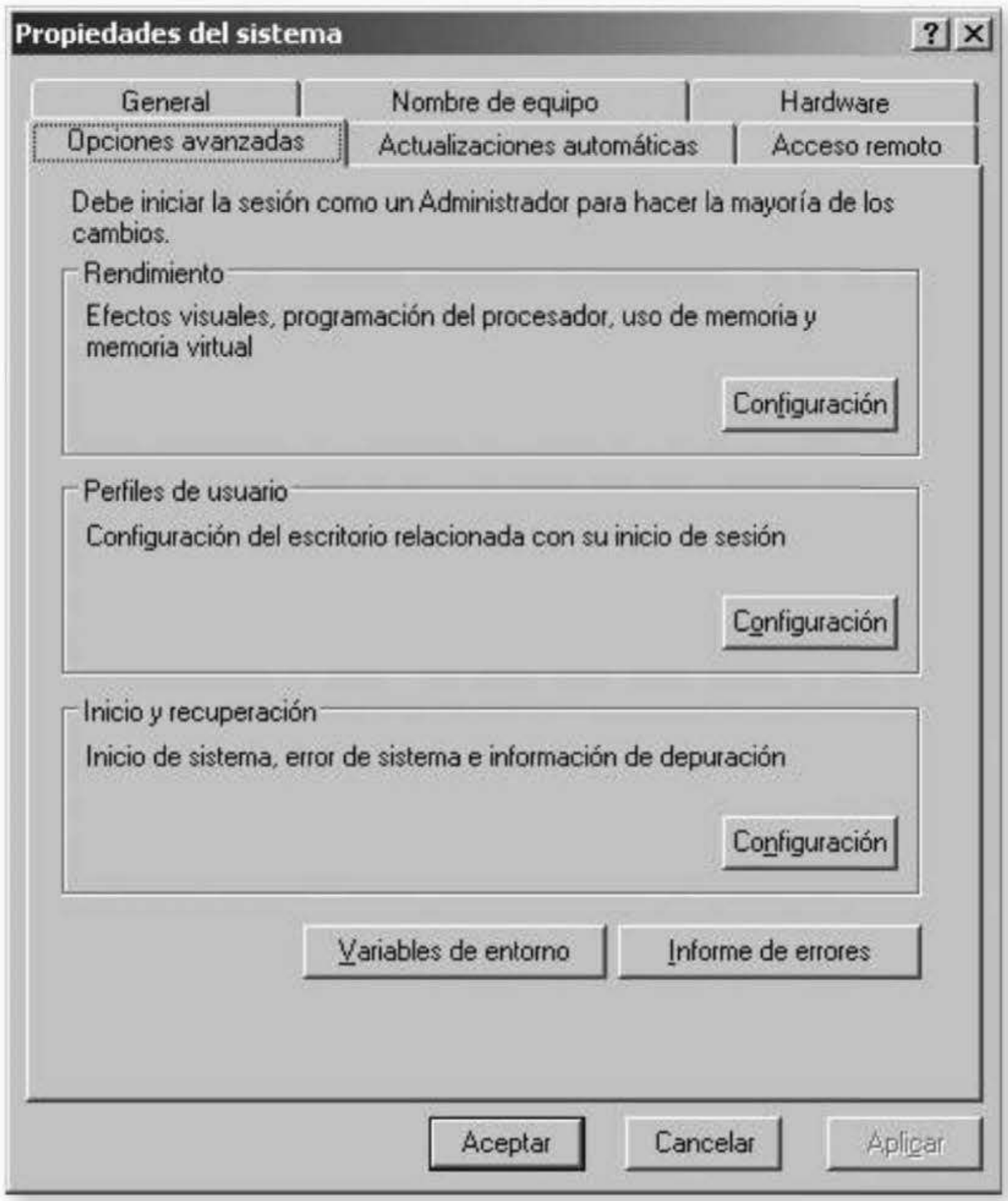

Fig. 2.15 Opciones avanzadas del sistema

Desde esta ficha podemos configurar las variables de entorno.

3. Cargar la ventana **Variables de entorno** haciendo clic en el botón **Variables de entorno**.

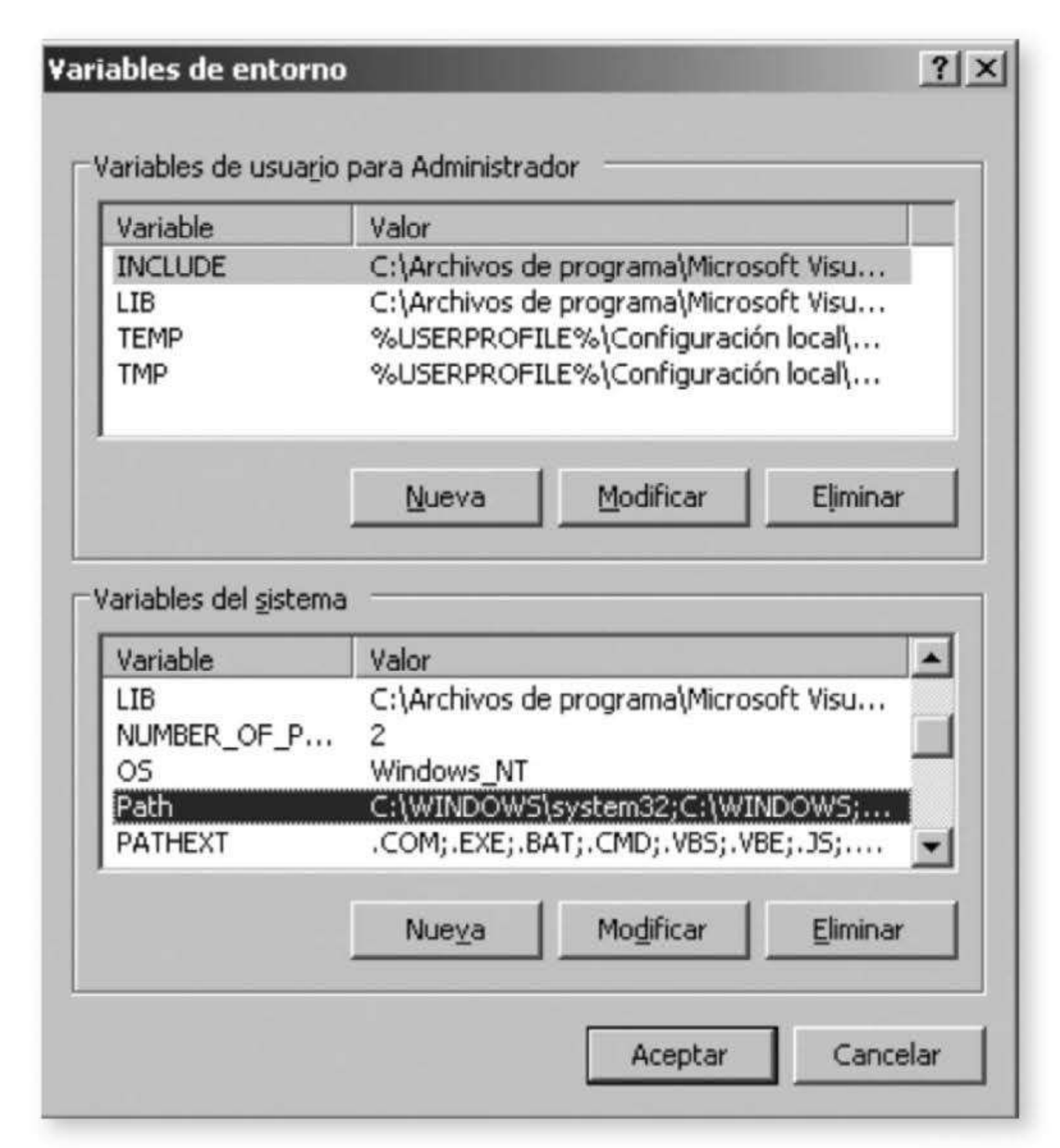

Fig. 2.16 Variables de entorno

Desde este diálogo podemos configurar la variable PATH en forma personalizada para el usuario actual (Administrador) o a nivel del sistema.

4. Ubicarse en la variable PATH en la lista **Variables del sistema**, y hacer clic en el botón **Modificar** correspondiente a la misma lista.

Fig. 2.17 Edición de la variable PATH

5. En el campo **Valor de la variable** agregar la ruta de la carpeta **bin** de Java y hacer clic en el botón **Aceptar**.

Fig. 2.18 Modificación de la variable PATH

6. Finalmente hacer clic en el botón **Aceptar** del diálogo **Variables de entorno** y en el diálogo **Propiedades del sistema**.

   ¡Ahora está listo para ejecutar las herramientas de Java desde cualquier ventana de consola!

Herramientas de desarrollo

CAPÍTULO 3

Capacidad: En este capítulo presentaré diversas herramientas de desarrollo que usted puede utilizar para la construcción de sus aplicaciones.

3.1 HISTORIA DE JAVA

Para trabajar con Java se debe elegir una herramienta de desarrollo, debido a que no cuenta con un entorno de desarrollo propio. Podemos utilizar desde un editor de textos simple como el bloc de notas, hasta entornos de desarrollo denominados IDEs que permiten manejar proyectos, depuración, entre otras cosas.

A continuación una breve relación de algunos IDEs que tenemos en el mercado.

Empresa	Producto	Versión
EditPlus	EditPlus	2.20
Oracle	Oracle JDeveloper 10g (Free)	10.1.3
Borland	JBuilder	2005
IBM	Websphere Server Application Developer	6.0
Sun Microsystems	Sun Java Creador	2004Q2
Macromedia	JRun	4.0
Xinox Software	JCreator Pro	3.5
NetBeans	NetBeans	4.1
Eclipse	Eclipse	3.1.1

3.2 EDITPLUS

EditPlus es un editor bastante ligero, reconoce colores de Java, HTML, y otros lenguajes.

Para obtener una copia de este editor los datos son los siguientes:

Sitio Web	http://www.editplus.com/
Nombre de Archivo	epp220_en.exe
Tamaño	939 Kb

En la Fig. 3.1 tenemos la interfaz de EditPlus.

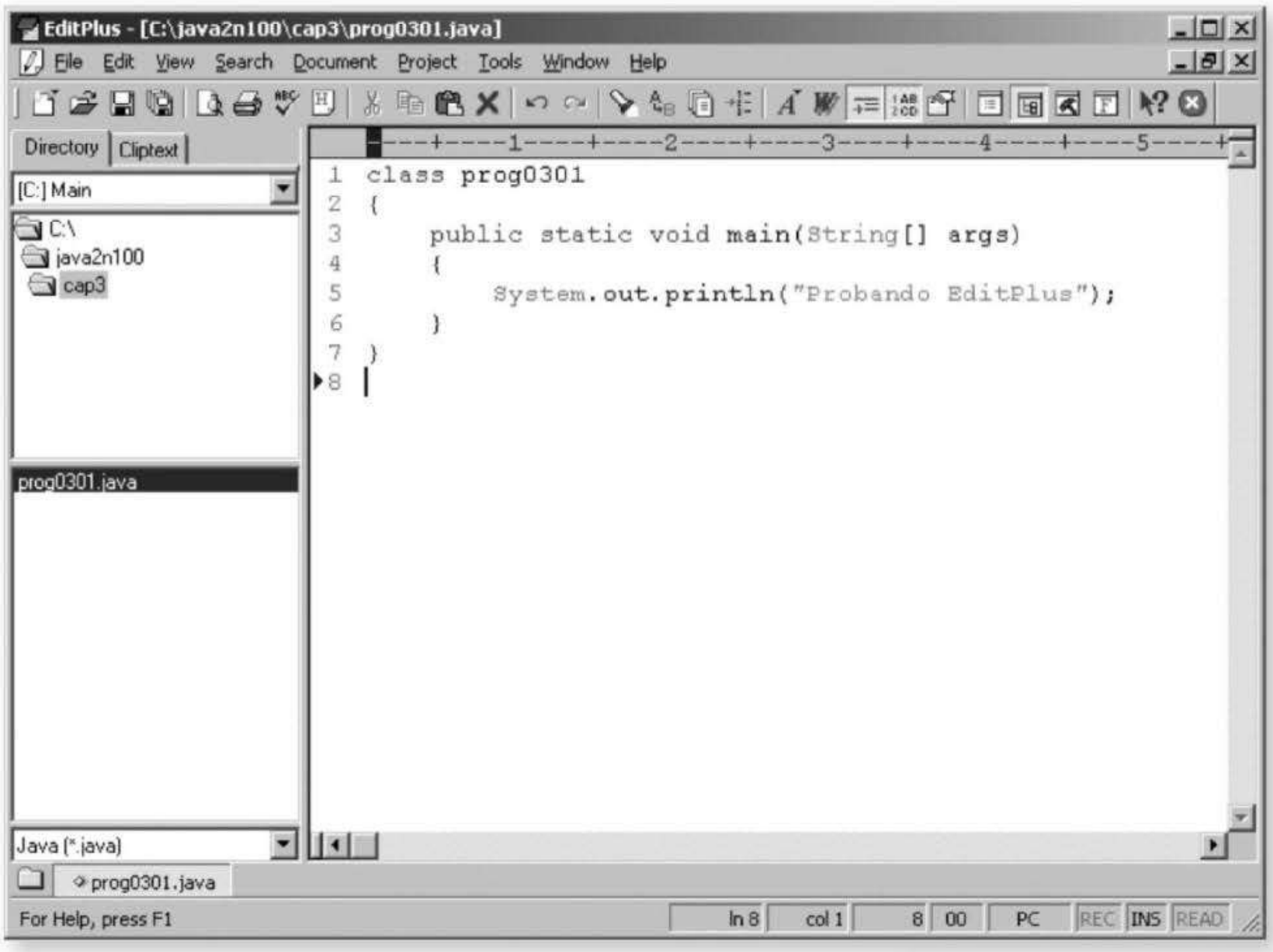

Fig. 3.1 Interfaz de EditPlus

EditPlus es solo un editor, por lo tanto para compilar y ejecutar el programa debemos hacerlo desde la consola de comandos, tal como se ilustra a continuación:

prog0301.java

```
public class prog0301
{
    public static void main(String[] args)
    {
        System.out.println("Probando EditPlus");
    }
}
```

La compilación y ejecución:

```
C:\java2n100\cap3>javac prog0301.java        [Enter]

C:\java2n100\cap3>java prog0301              [Enter]
Probando EditPlus
```

3.3 JDEVELOPER

JDeveloper es un IDE bastante completo e interesante, además es *free*.

Para obtener una copia de este editor los datos son los siguientes:

Sitio Web	http://www.oracle.com/technology/products/jdev/
Nombre de Archivo	– jdevstudio1013.zip (contiene el JDK 5.0) – jdevstudiobase1013.zip (no contiene el JDK)
Tamaño	– jdevstudio1013.zip (306 Mb) – jdevstudiobase1013.zip (250 Mb)

Después de desempaquetar el archivo **jdevstudio1013.zip** tenemos la carpeta **jdevstidio1013**, dentro de esta tenemos el programa **jdeveloper.exe**, este es el programa que debe ejecutarse para cargar el IDE (ver Fig. 3.2).

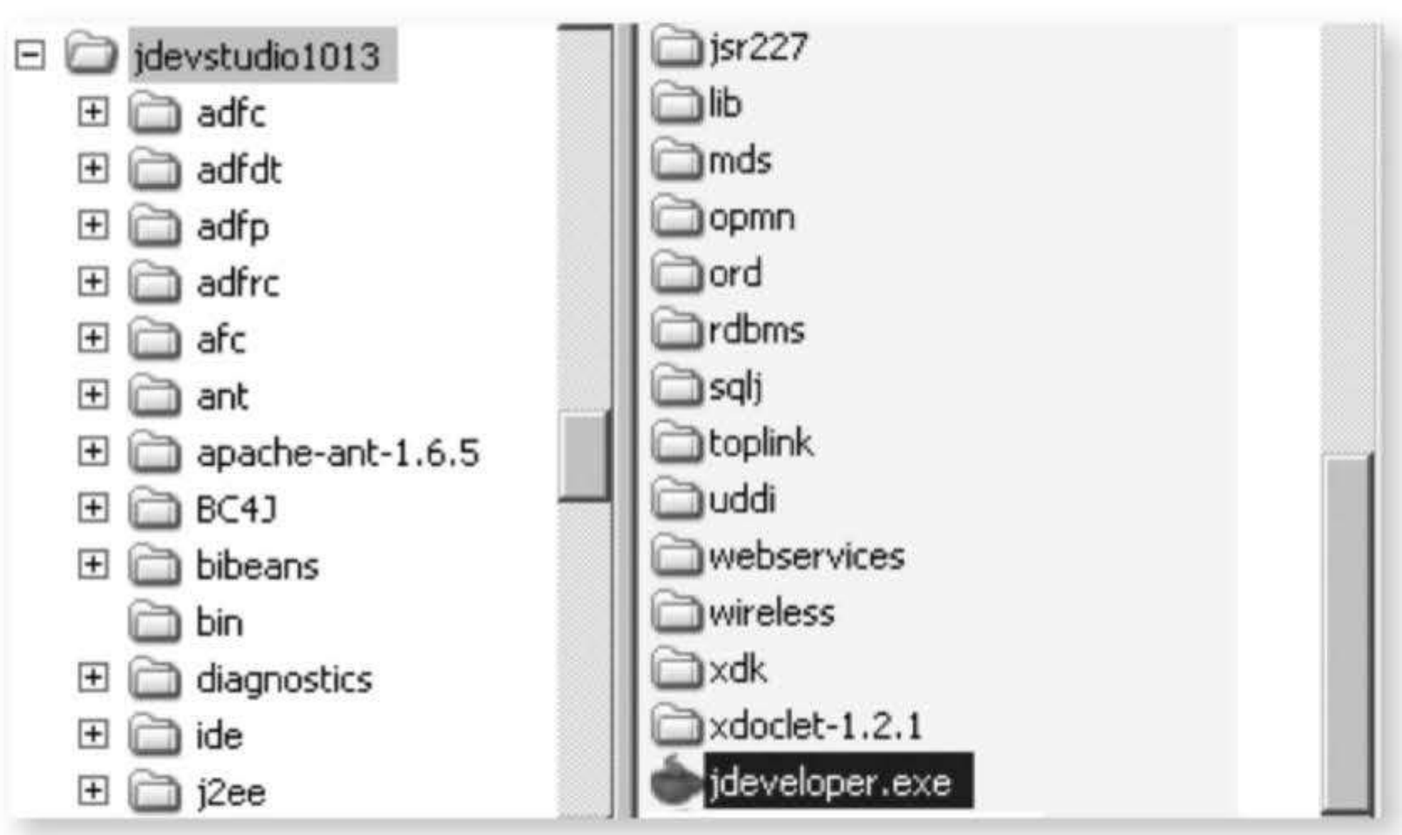

Fig. 3.2 Programa para cargar el IDE JDeveloper

La Fig. 3.3 muestra la pantalla inicial de este IDE.

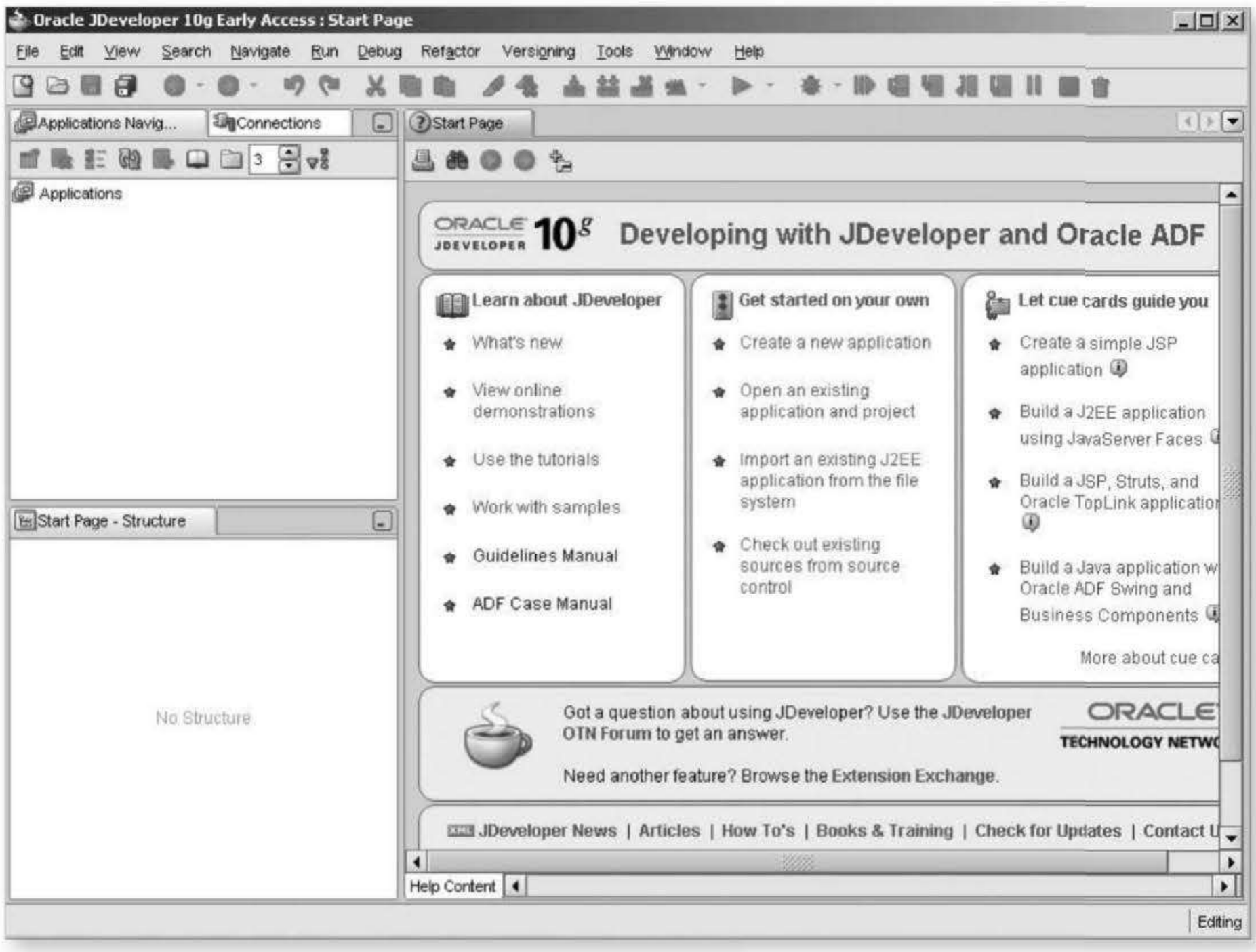

Fig. 3.3 Pantalla Inicial de JDeveloper

3.4 NETBEANS

NetBeans es un IDE bastante completo e interesante, además es *free*.

Para obtener una copia los datos son los siguientes:

Sitio Web	http://www.netbeand.org/
Nombre de Archivo	– netbeans-4_1-windows.exe – jdk-1_5_0_05-nb-4_1-win-ml.exe (Incluye el JDK 5.0)
Tamaño	– netbeans-4_1-windows.exe (46.43 Mb) – jdevstudiobase1013.zip (127.96 Mb)

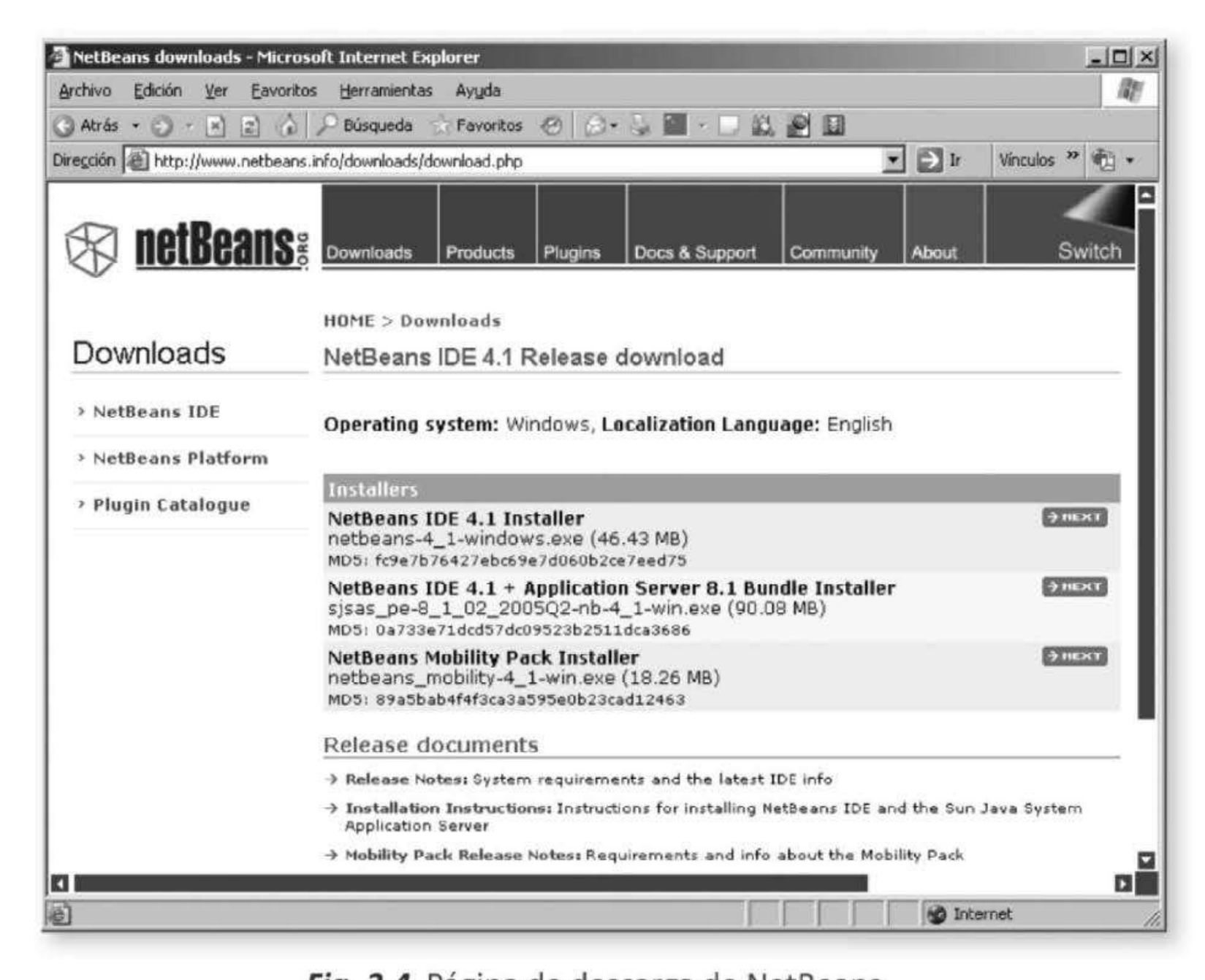

Fig. 3.4 Página de descarga de NetBeans

Si procedemos a instalar, por ejemplo el **netbeans 4_1 windows.exe**, es necesario tener instalado el JDK. Durante el proceso de instalación detecta automáticamente los JDK que podemos tener instalados en nuestro equipo, solo debemos seleccionar uno de ellos y la instalación es automática.

La carga del IDE es muy sencilla (ver Fig. 3.5). Tenemos una opción dentro del menú programas.

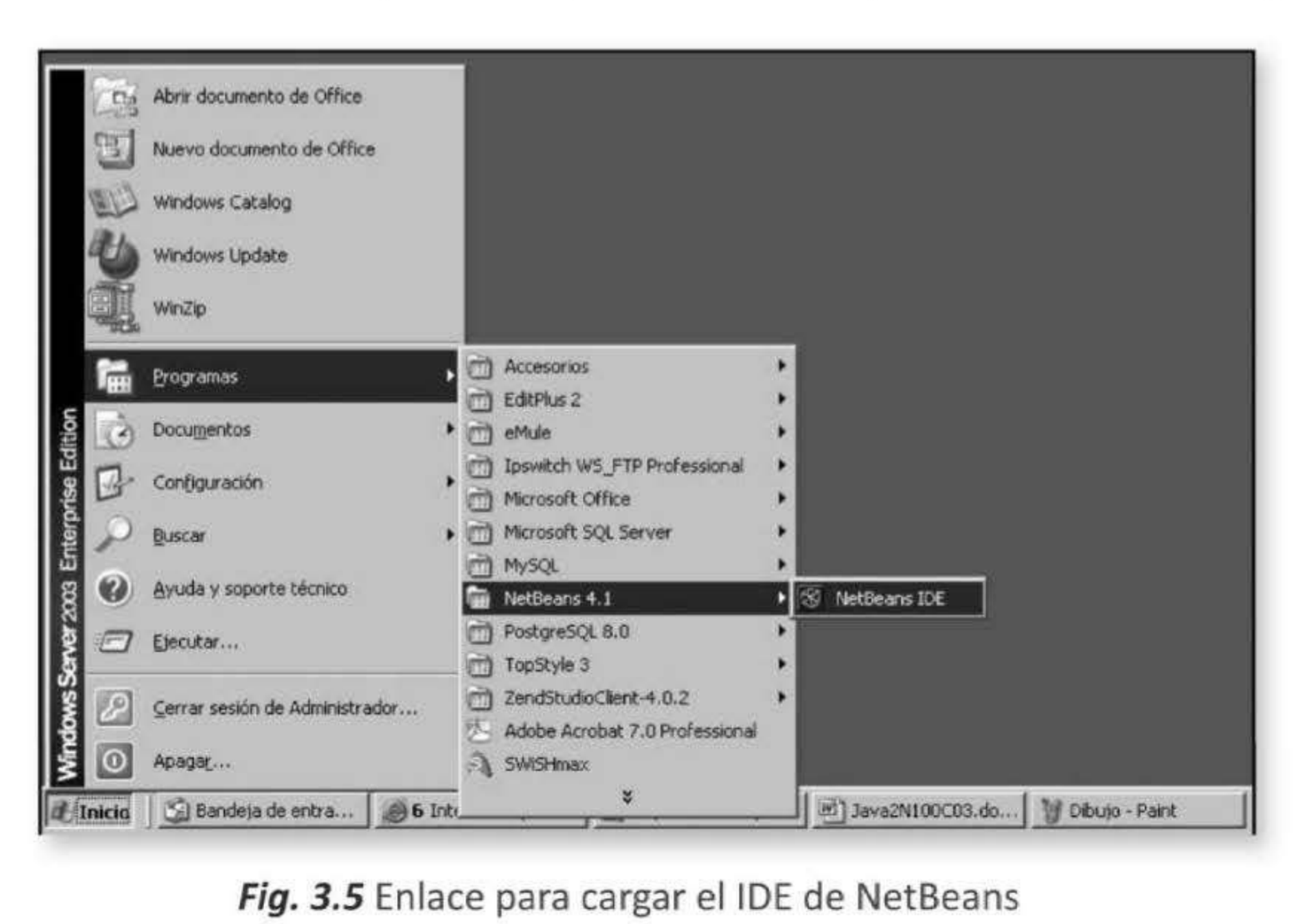

Fig. 3.5 Enlace para cargar el IDE de NetBeans

La presentación de este IDE se muestra en la Fig. 3.6.

Fig. 3.6 Presentación del IDE de NetBeans

3.4.1 Inicio de un nuevo proyecto

El procedimiento es el siguiente:

1. Haga clic en el botón **New Project**.
2. En el diálogo **New Project**:
   a. De la lista **Categories**, seleccione **General**.
   b. De la lista **Projects**, seleccione **Java Application**.
   c. Haga clic en el botón **Next**.
3. En el diálogo **New Java Application**:
   a. En el campo **Project Name** ingrese el nombre del proyecto, por ejemplo **EjmNBeans**.
   b. En el **Project Location** debe ingresar la carpeta donde se grabará el proyecto; por ejemplo **C:\java2n100\cap3**.
   c. Los demás campos los deja con los valores por defecto.
   d. Finalmente hace clic en el botón **Finish**.

De esta manera se ha iniciado un proyecto en NetBeans. Puede observar que se ha creado una clase **Main**, y dentro de esta clase tenemos el método **main**, tal como se aprecia en la Fig. 3.7.

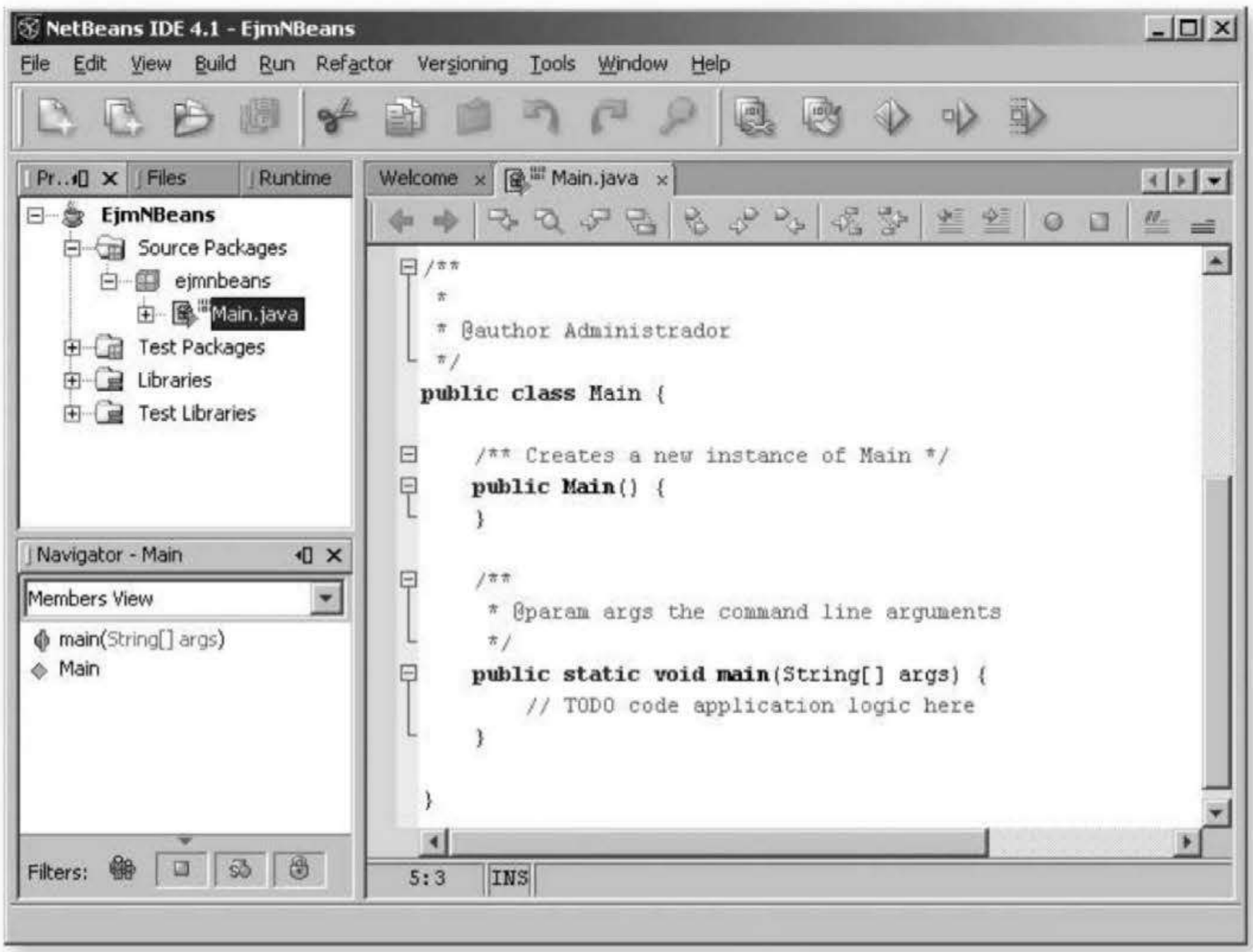

Fig. 3.7 Inicio de un nuevo proyecto en NetBeans

Modifiquemos el método **main** con el código que se muestra a continuación:

Método:main

```
public static void main(String[] args) {
        System.out.println("Primer Contacto con NetBeans");
        System.out.println("Y no será el último");
    }
```

Para ejecutar nuestro proyecto tenemos el comando **Run/Run Main Project** o simplemente presionar la tecla **F6**; el resultado lo puede apreciar en el panel **Output** (ver Fig. 3.8).

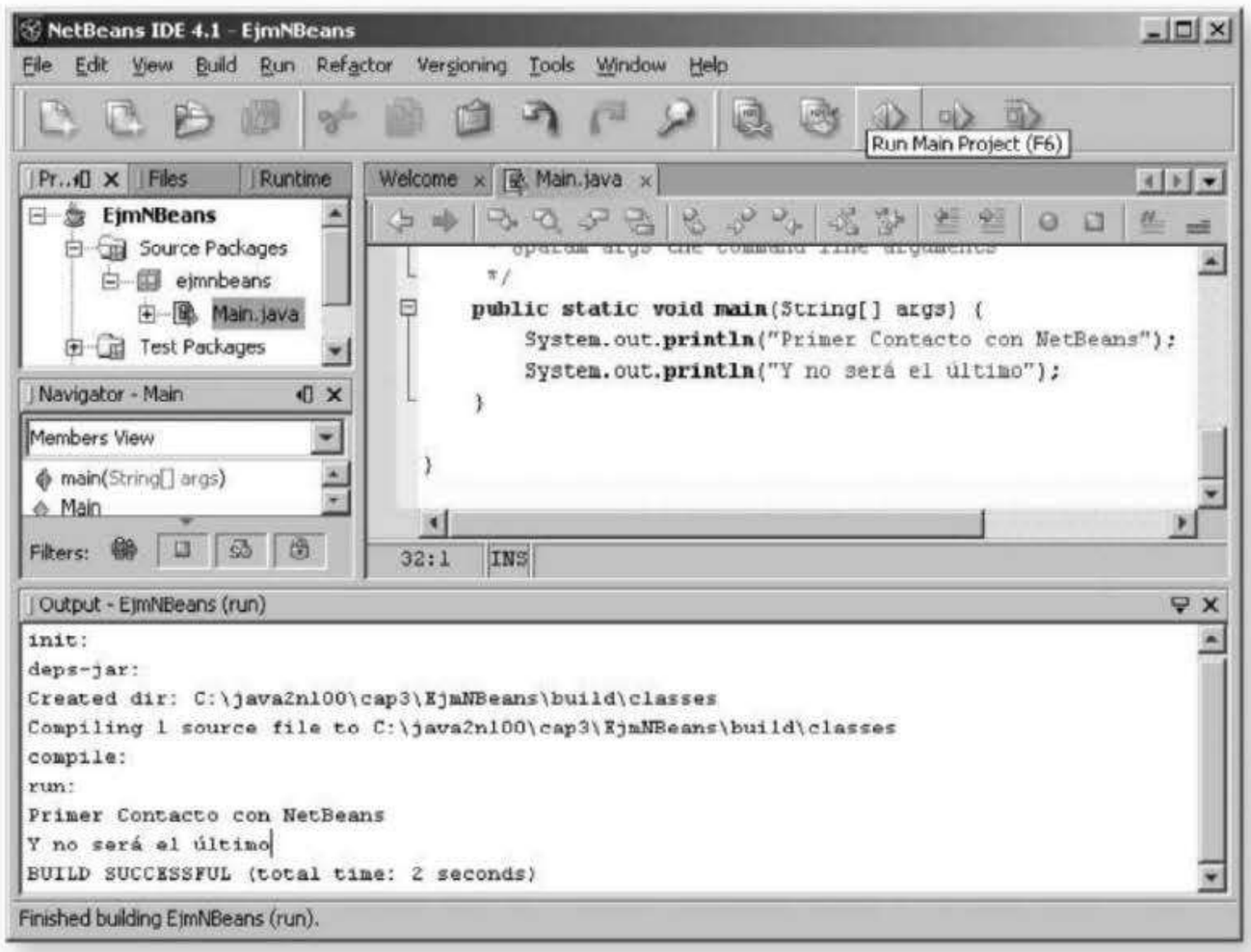

Fig. 3.8 Ejecución de un proyecto

3.5 JCREATOR PRO

JCreator Pro es un IDE bastante ligero e interesante.

Para obtener una copia los datos son los siguientes:

Sitio Web	http://www.jcreator.com
Nombre de Archivo	jcpro350.zip
Tamaño	3709 Kb

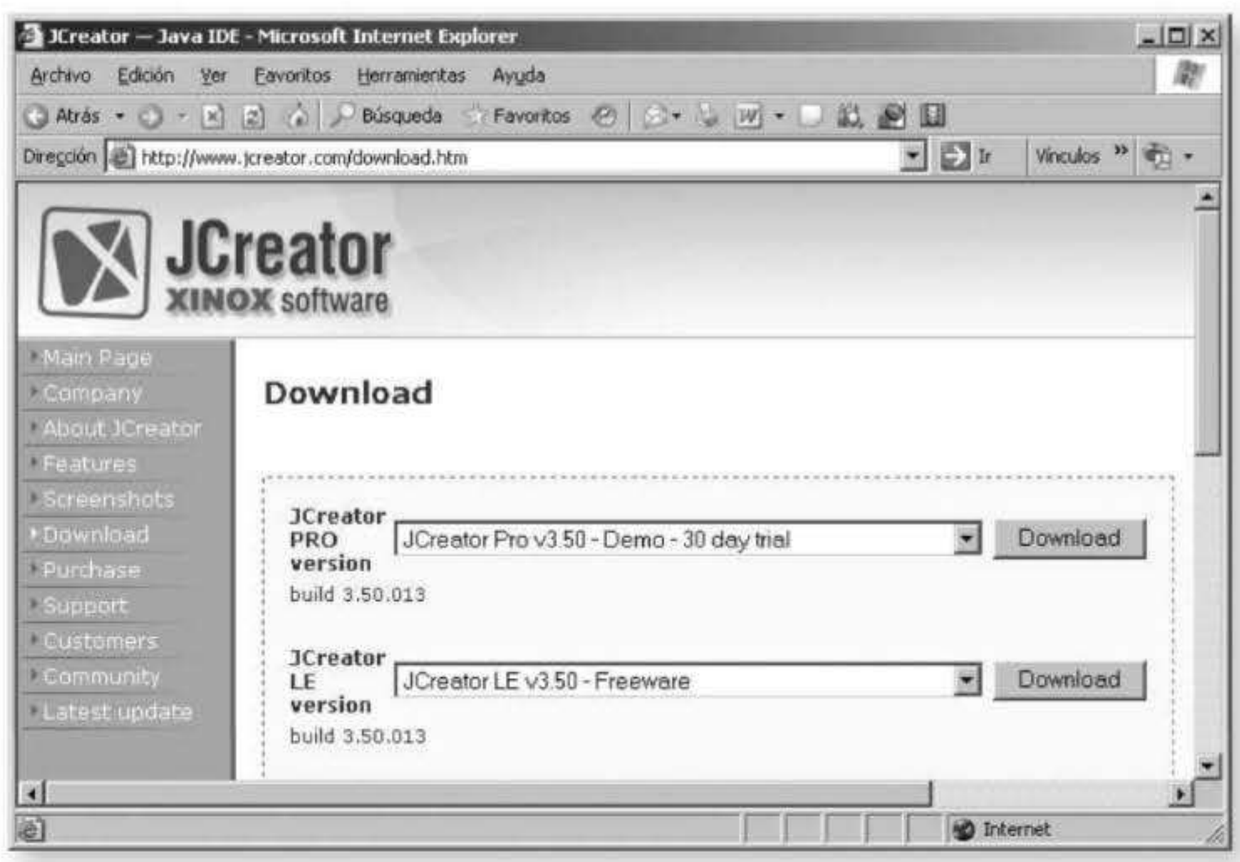

Fig. 3.9 Página de descarga de JCreatorPro

Proceda a instalar el programa después de la instalación para cargar el IDE, es muy sencillo (ver Fig.3.10). Tenemos una opción dentro del menú **Programas**.

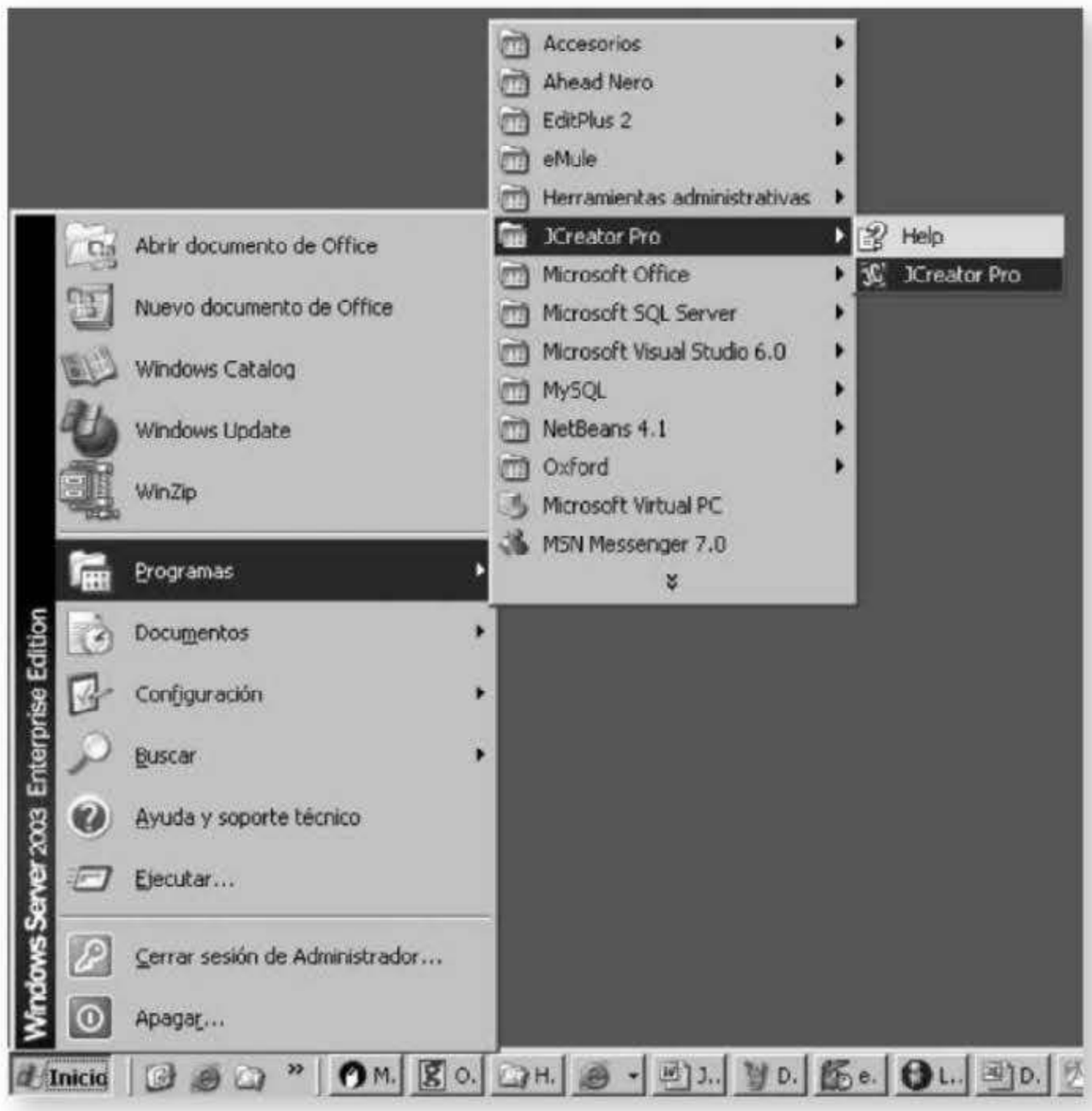

Fig. 3.10 Enlace para cargar el IDE de JCreator

La presentación de este IDE se muestra en la Fig. 3.11, como puede apreciarse es bastante sencilla, pero funcional.

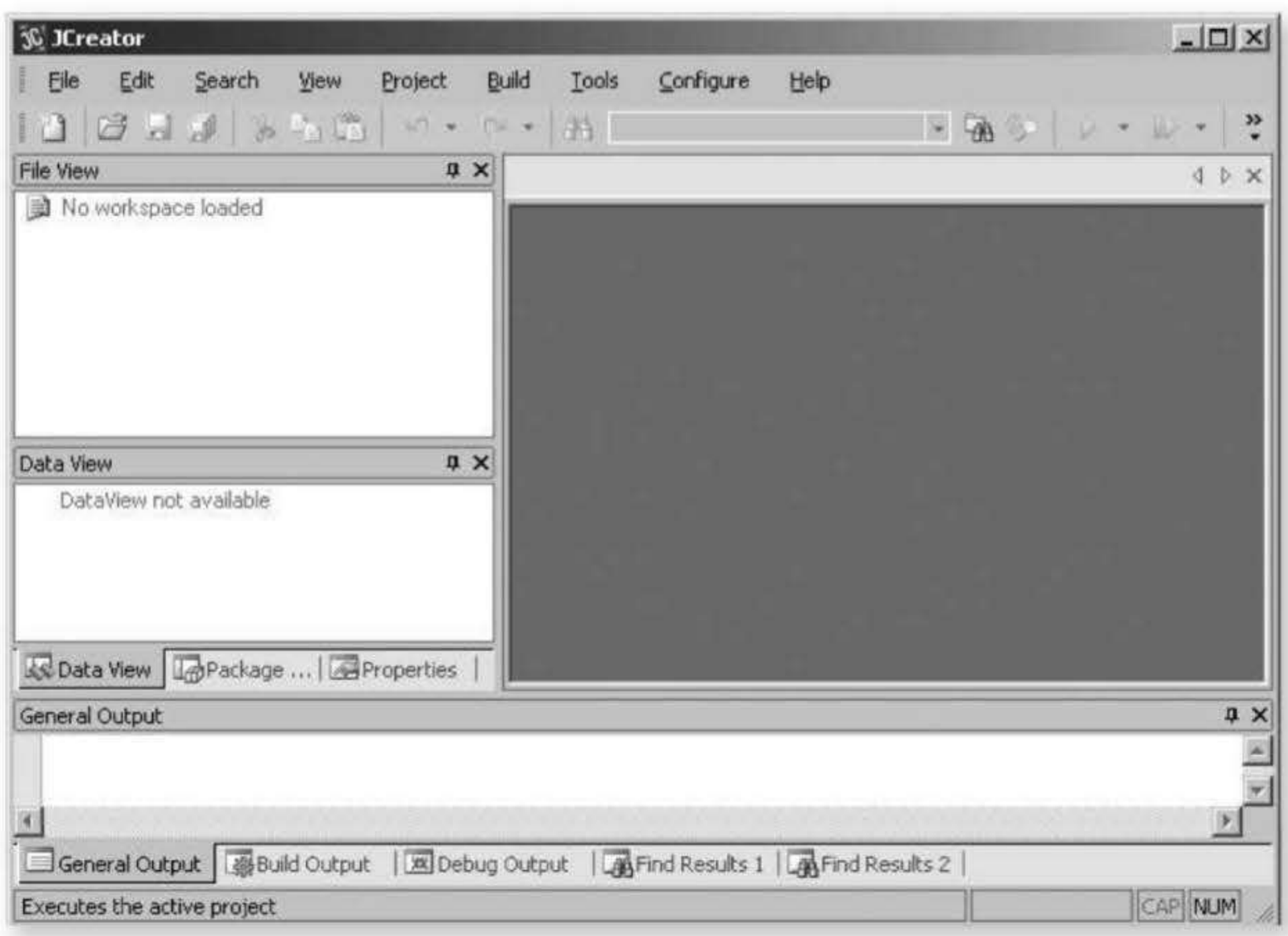

Fig. 3.11 Presentación inicial del IDE de JCreator

3.5.1 Inicio de un nuevo proyecto

El procedimiento para iniciar un nuevo proyecto es el siguiente:

1. Desde el menú **Project** ejecute el comando **New Project**, se carga el asistente a través del cual seguiremos una serie de ventanas donde nos solicitará datos para el nuevo proyecto.
2. Lo primero que debemos seleccionar es la plantilla del proyecto. Vamos a crear un proyecto vacío, para lo cual debe seleccionar **Empty Project** y hacer clic en el botón **Next**.
3. Ahora debe ingresar el nombre del proyecto y las rutas donde se guardarán los archivos (ver Fig. 3.12), luego deberá hacer clic en el botón **Next**.

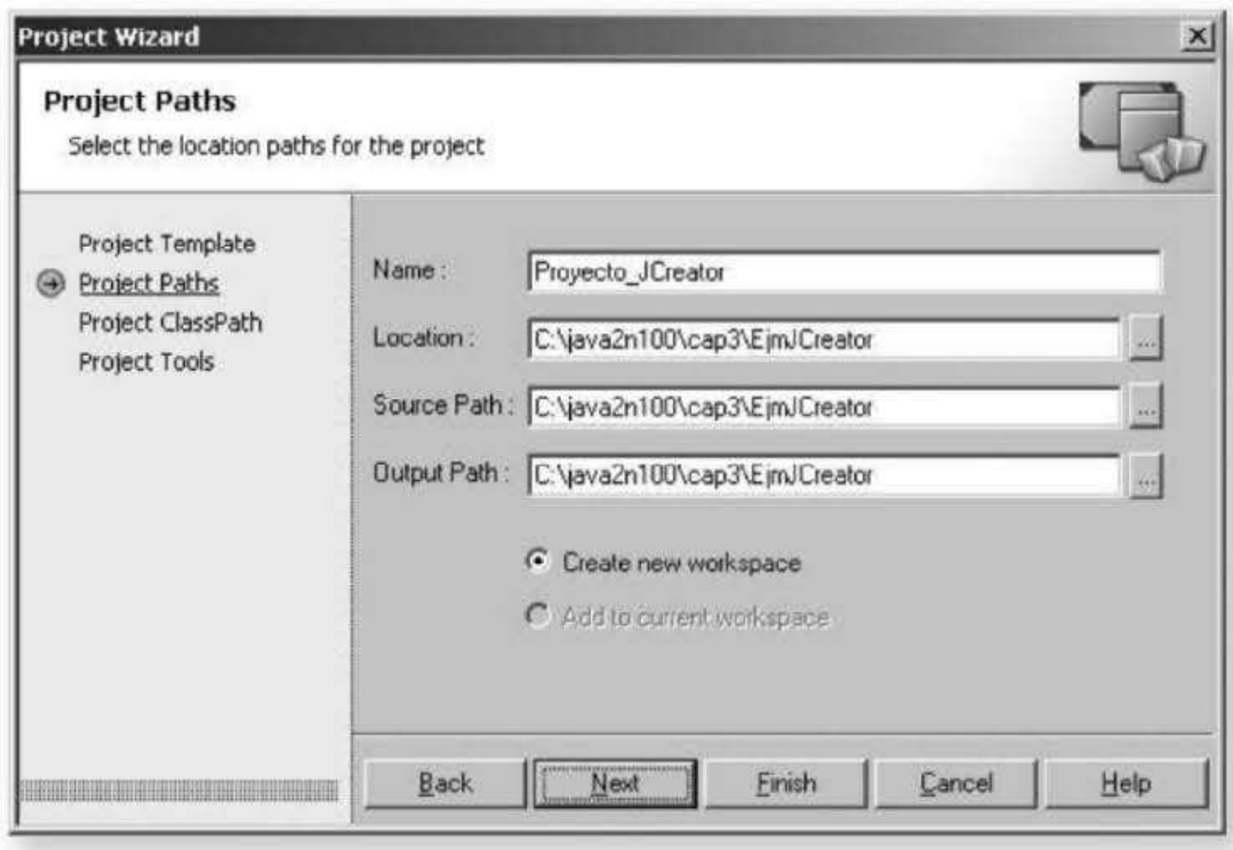

Fig. 3.12 Ubicación y nombre del proyecto

4. En la siguiente ventana debe seleccionar el JDK que utilizará, esto en caso de que tenga instaladas varias versiones del JDK. Luego hacer clic en el botón **Finish**.

   De esta manera ya se ha iniciado un proyecto, que en este momento se encuentra vacío, según la plantilla seleccionada.

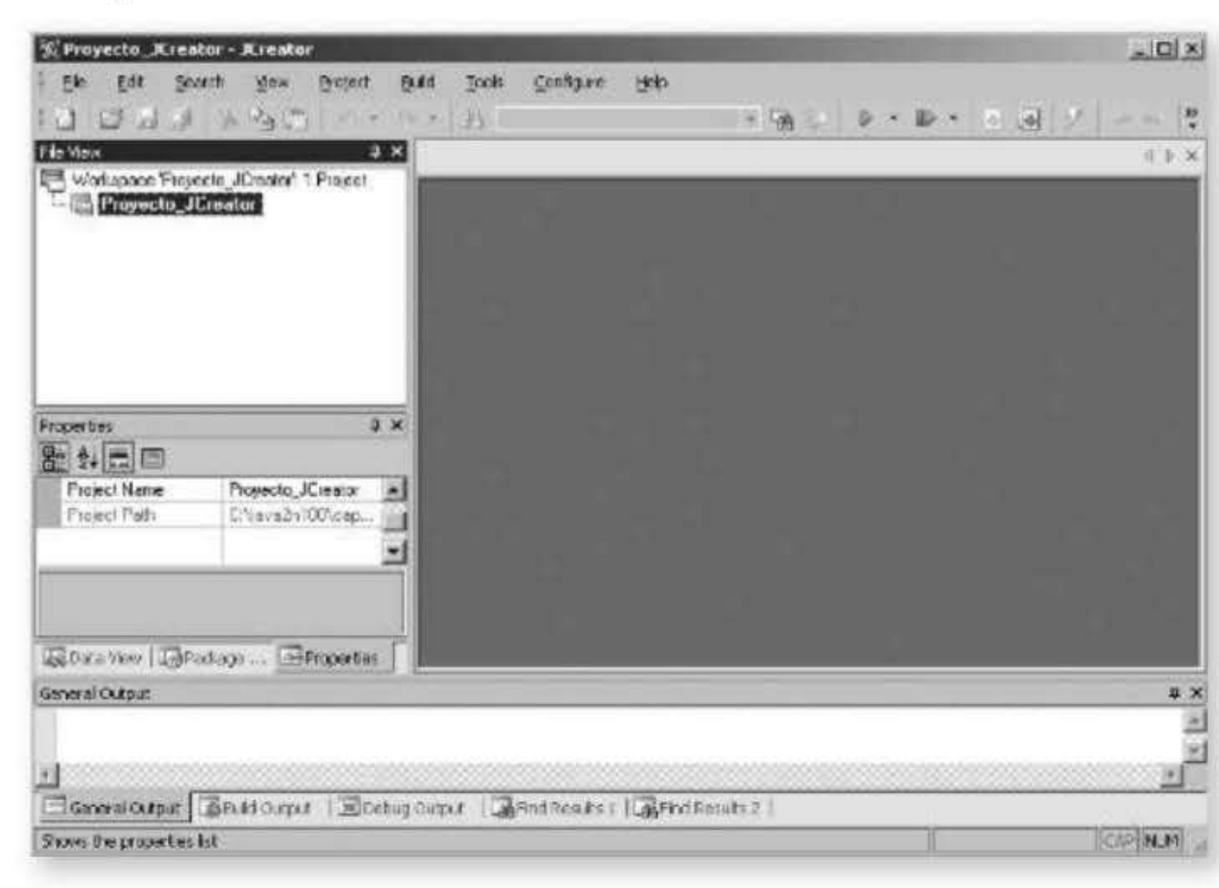

Fig. 3.13 Nuevo proyecto en JCreator

3.5.2 Agregar una nueva clase

Para agregar una nueva clase ejecute desde el menú **Proyect** el comando **New Class**, para cargar el asistente de nuevas clases.

Ingrese como nombre de clase **Test** y marque las opciones **Public** y **Generate main method**; las demás opciones déjelas por defecto,y finalmente haga clic en el botón **Finish**.

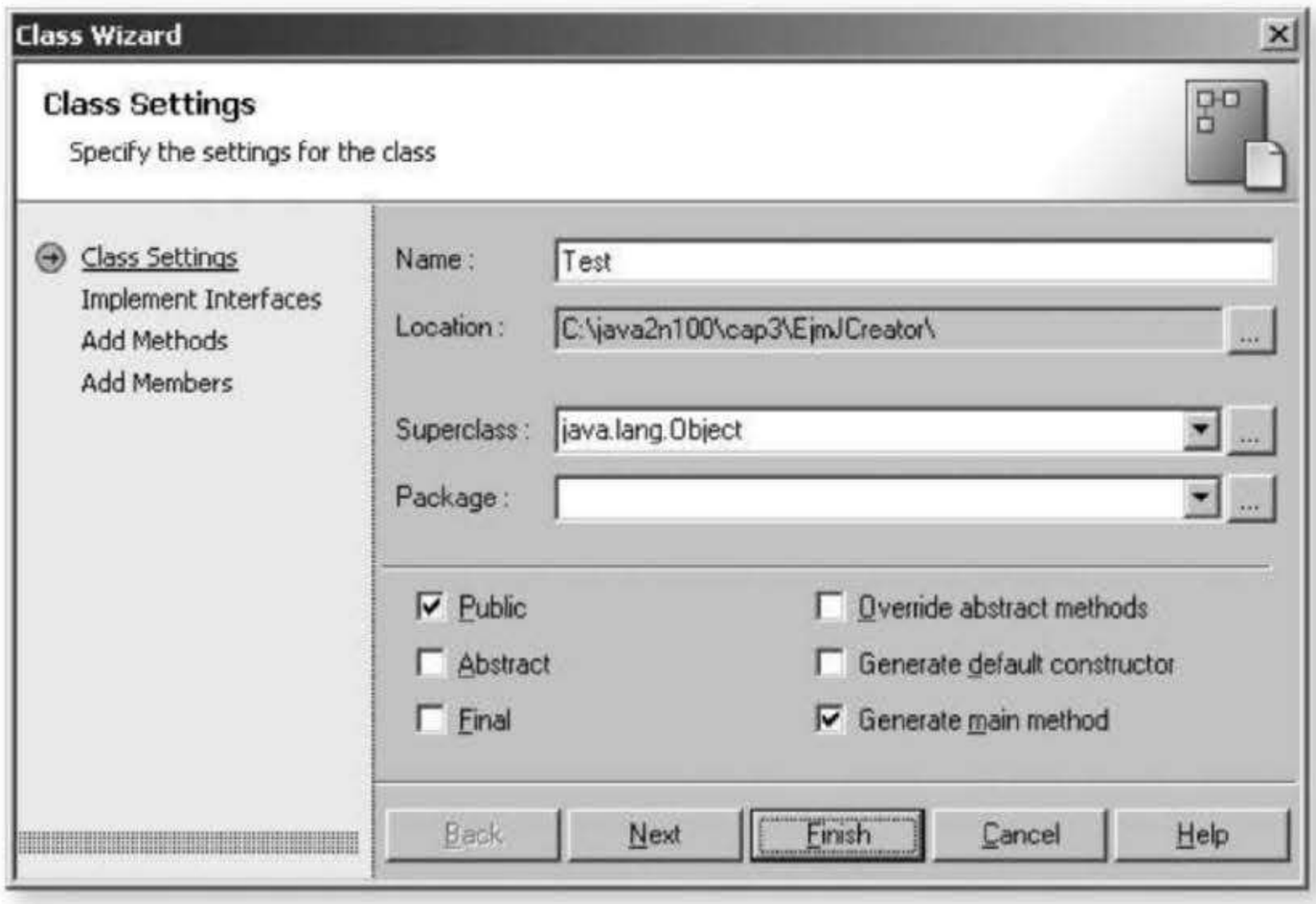

Fig. 3.14 Creación de nuevas clases

De esta manera hemos agregado una clase al nuestro proyecto.

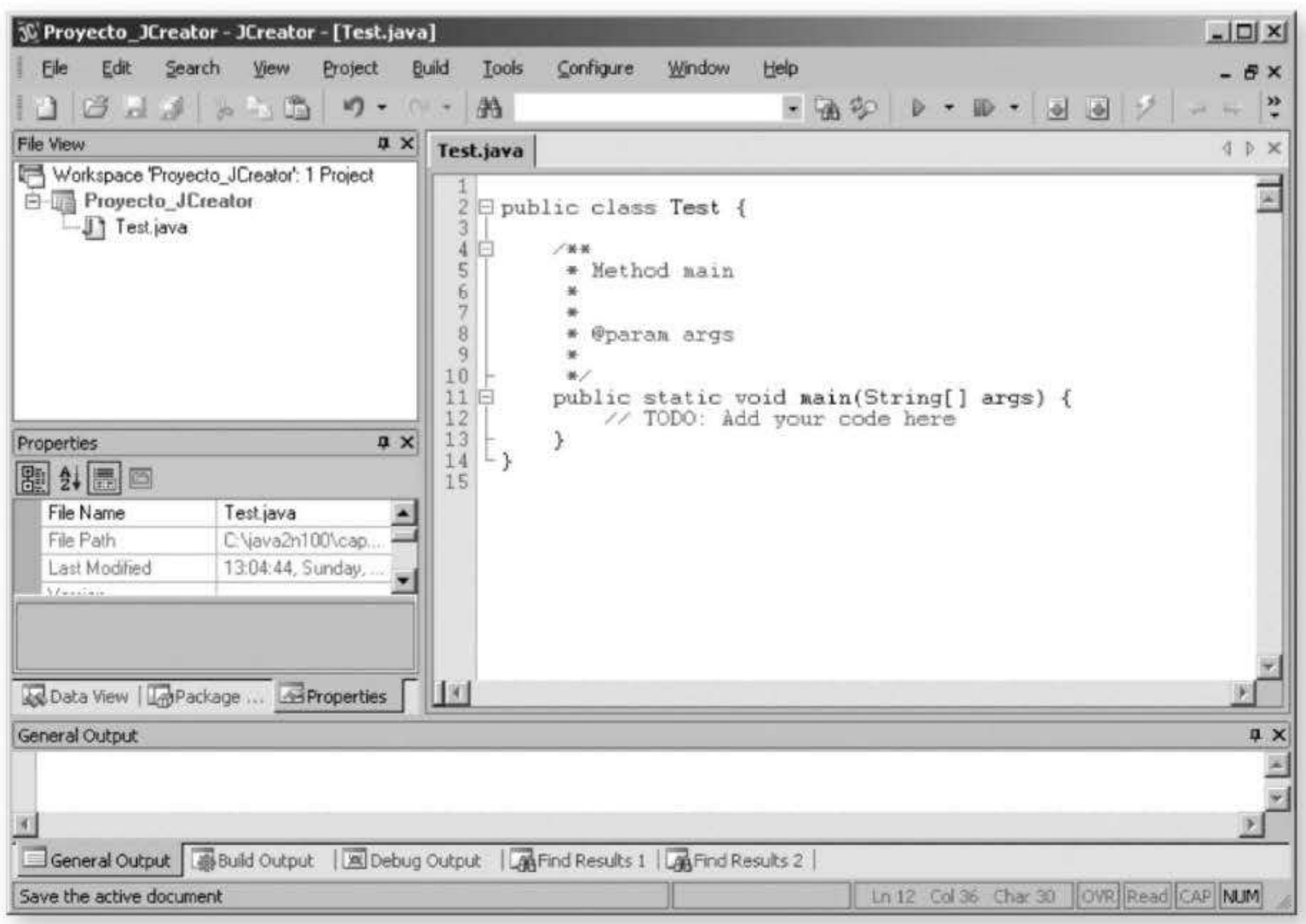

Fig. 3.15 Nueva clase en el proyecto

Modifique el método **main**, tal como se indica a continuación:

Método: main

```
public static void main(String[] args) {
            // TODO: Add your code here
            System.out.println("Bienvenidos a JCreator.");
            System.out.println("Pronto seremos buenos amigos.");
    }
```

Para compilar el archivo ejecute **Build / Compile File** y para realizarlo **Build / Execute File**. El resultado se aprecia en la Fig. 3.16.

Fig. 3.16 Ejecución de la clase Test

También puede ejecutarlo como proyecto, en este caso el comando a ejecutar es **Build / Execute Project**, en este caso el resultado es el mismo.

3.6 ECLIPSE

Eclipse es otro de los IDE muy completo y free con los que cuenta el mercado de los IDE's para Java.

Para obtener una copia los datos son los siguientes:

Sitio Web	http://www.eclipse.org
Nombre de Archivo	eclipse-SDK-3.1.1-win32.zip
Tamaño	103 Mb

Fig. 3.17 Página de descarga de Eclipse

Para instalar **Eclipse** solo debemos desempaquetar el archivo eclipse **SDK 3.1.1 win32.zip** en la carpeta donde quiere tenerlo instalado; por ejemplo en la raíz de **C:**.

Para empezar a trabajar con Eclipse solo debe ejecutar el archivo **eclipse.exe**, que lo encontrará en **C:\eclipse**; luego nos solicitará la ruta donde se guardarán nuestros proyectos por defecto. Por ejemplo podría ser **C:\java2n100\cap3** (ver Fig. 3.18).

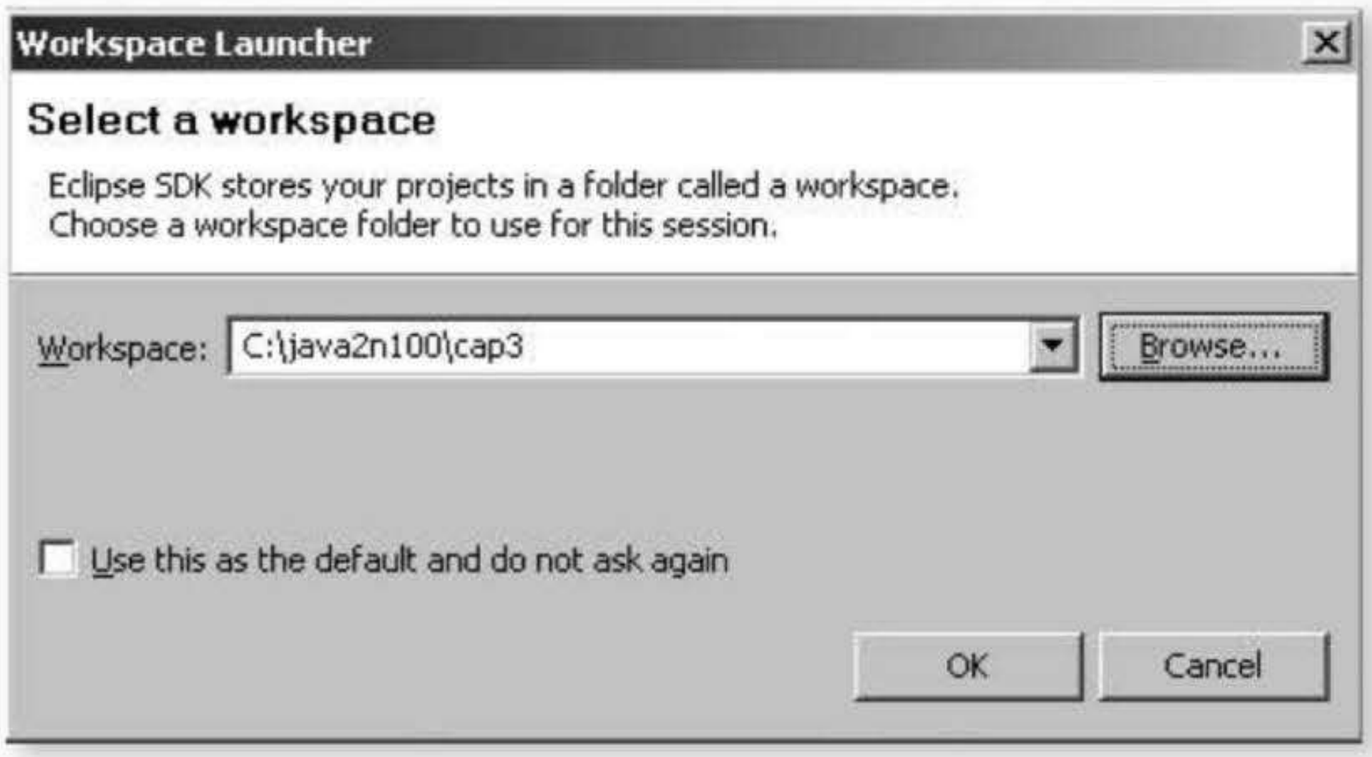

Fig. 3.18 Ingresando la ruta donde se grabarán los proyectos

tenemos la interfaz principal de Eclipse donde presenta una ventana de bienvenida (ver Fig. 3.19).

Fig. 3.19 Interfaz de Eclipse

Luego de cerrar la ventana de bienvenida tenemos la interfaz de trabajo de Eclipse (ver Fig. 3.20).

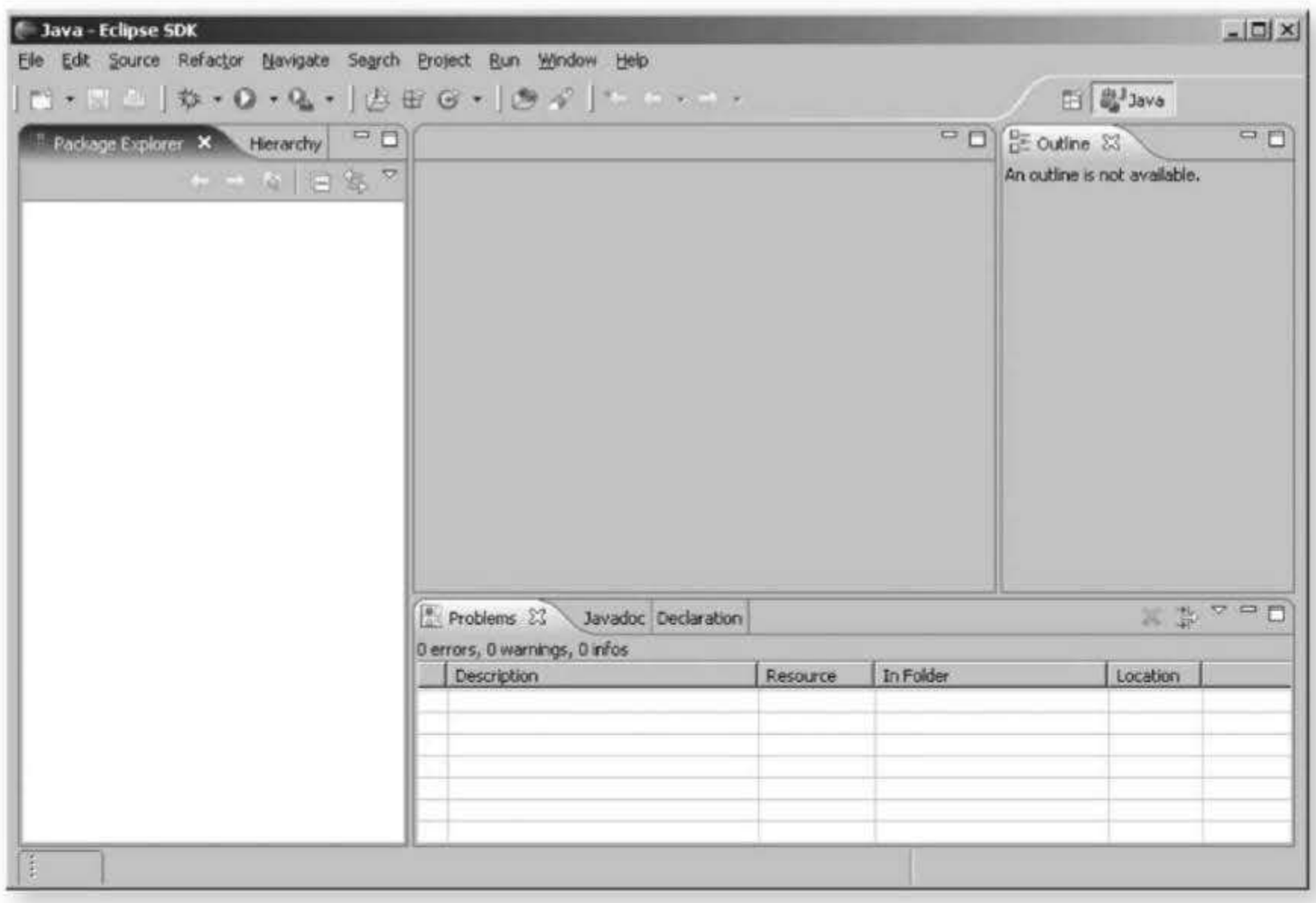

Fig. 3.20 Interfaz de trabajo de Eclipse

3.6.1 Inicio de un nuevo proyecto

Para iniciar un nuevo proyecto siga los siguientes pasos:

1. Ejecute el comando **File/New/Project**, de esta manera iniciamos el asistente para la creación de un nuevo proyecto.
2. En la primera ventana del asistente debe seleccionar el tipo de proyecto. Seleccione **Java Project**.
3. En la segunda ventana debemos ingresar el nombre del proyecto, en este caso será **EjmEclipse**, note que la ruta es **C:\java2n100\cap3\EjmEclipse** y luego haga clic en el botón **Finish**.

3.6.2 Agregar una nueva clase

Para agregar una clase siga los siguientes pasos:

1. Ejecute el comando **File/New/Class**, de esta manera estamos cargando el diálogo donde ingresaremos los datos para la creación de una nueva clase.
2. En el campo **Name** ingrese el nombre de la clase. Para nuestro ejemplo será **Test**, seleccione la opción **Public static void main(String[] args)**, y finalmente haga clic en el botón **Finish**.

En estos momentos ya tenemos la clase **Test** en nuestro proyecto, modifique el método **main**, tal como se ilustra a continuación:

Método: main

```
public static void main(String[] args) {
            // TODO Auto-generated method stub
            System.out.println("Bienveniso a Eclipse");
            System.out.println("Es un excelente IDE");
    }
```

Para ejecutar nuestro proyecto ejecute el comando **Run/Run**, así tenemos la ventana **Run**. Aquí seleccionamos la configuración **Java Application** y la clase **Test** en el campo **Main class** (ver Fig. 3.21), y finalmente haga clic en el botón **Finish**.

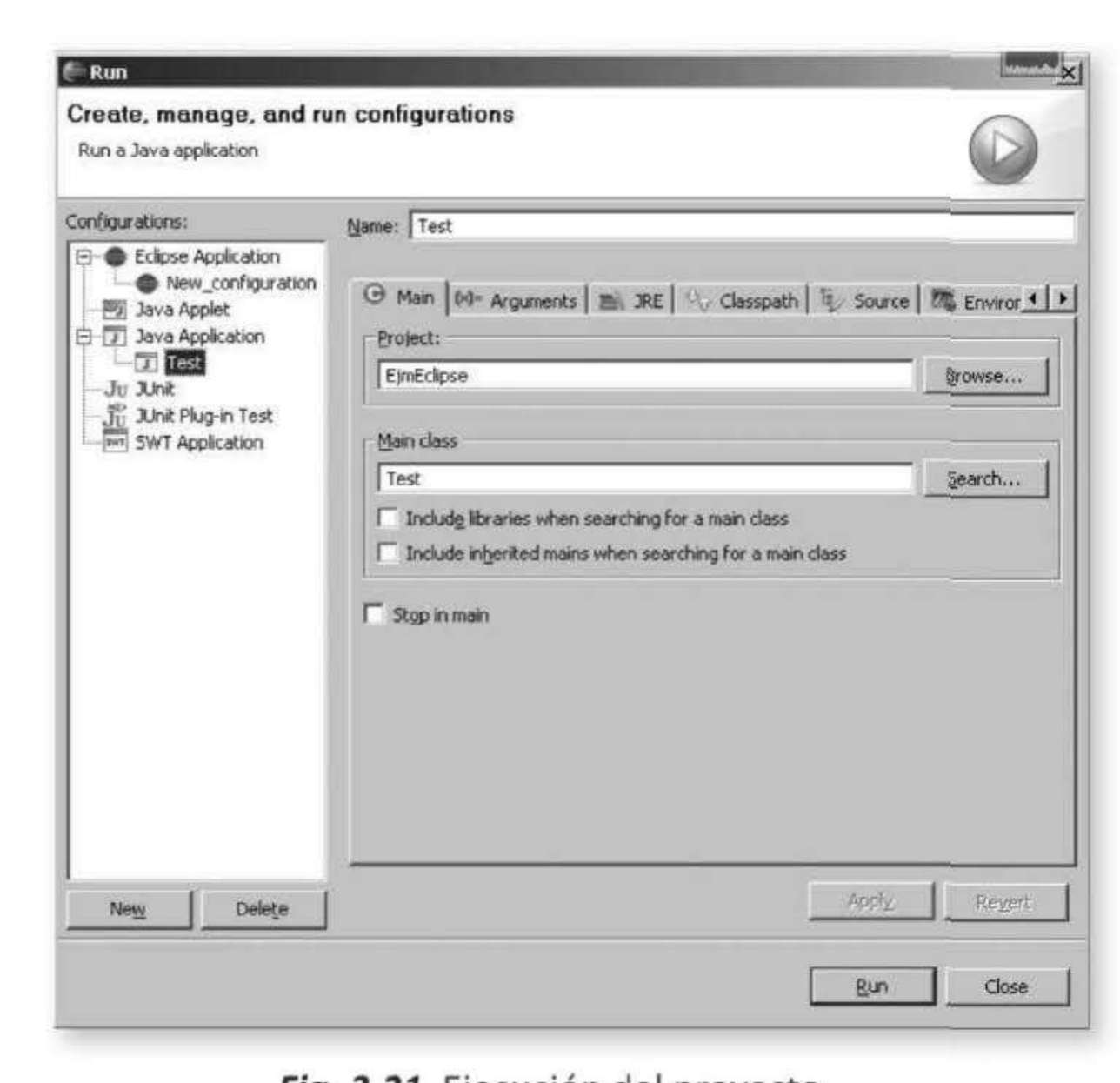

Fig. 3.21 Ejecución del proyecto

El resultado se obtiene en el panel **Console** (ver Fig. 3.22).

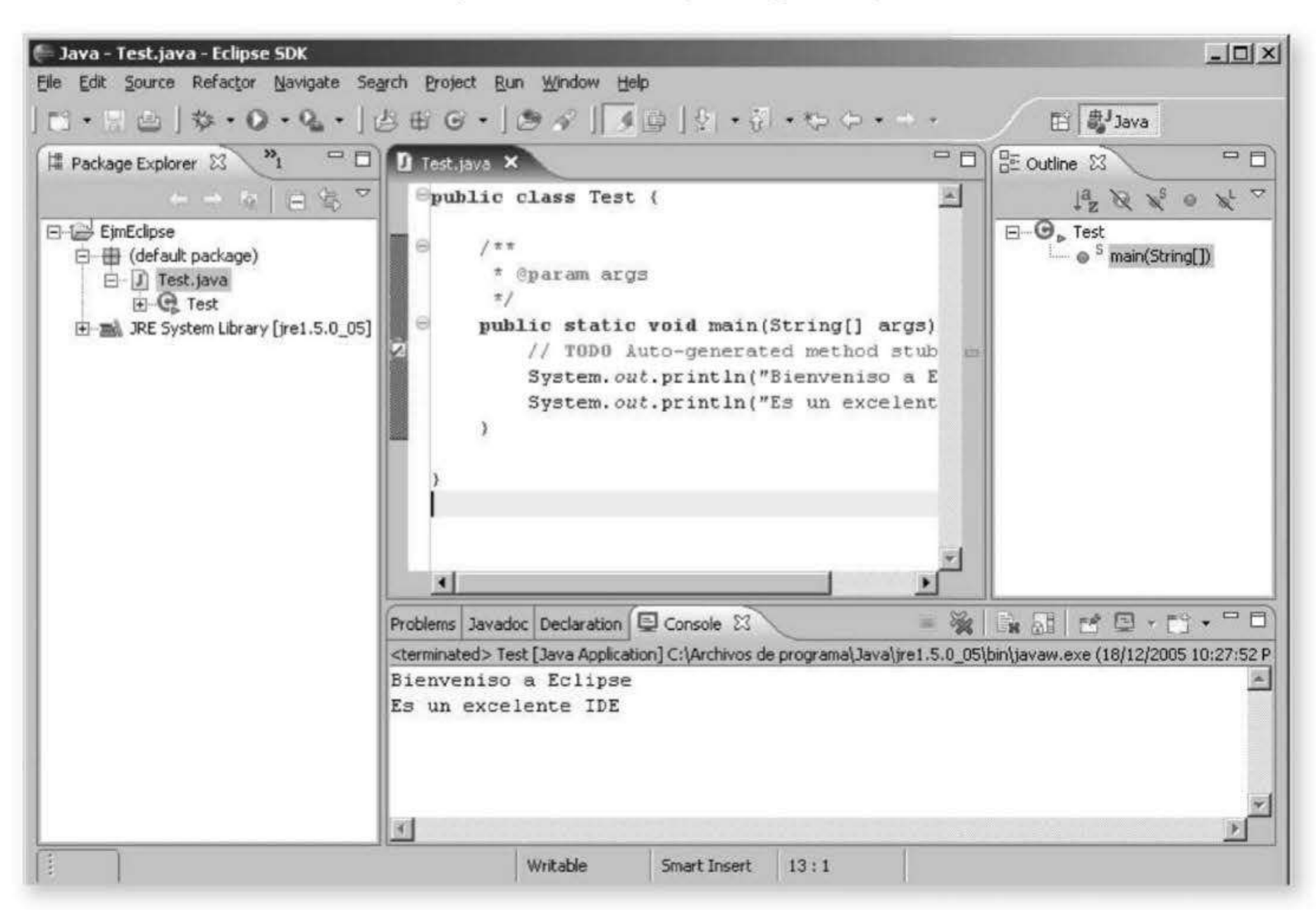

Fig. 3.22 Resultado de la ejecución del proyecto

3.7 JBUILDER

JBuilder es uno de los IDE más conocidos, es muy completo y permite desarrollar todo tipo de proyectos. de una manera muy fácil. Para obtener una copia los datos son los siguientes:

Sitio Web	http://www.oracle.com
Nombre de Archivo	jb2005_devtrialdl_win_mac.zip
Tamaño	421 Mb

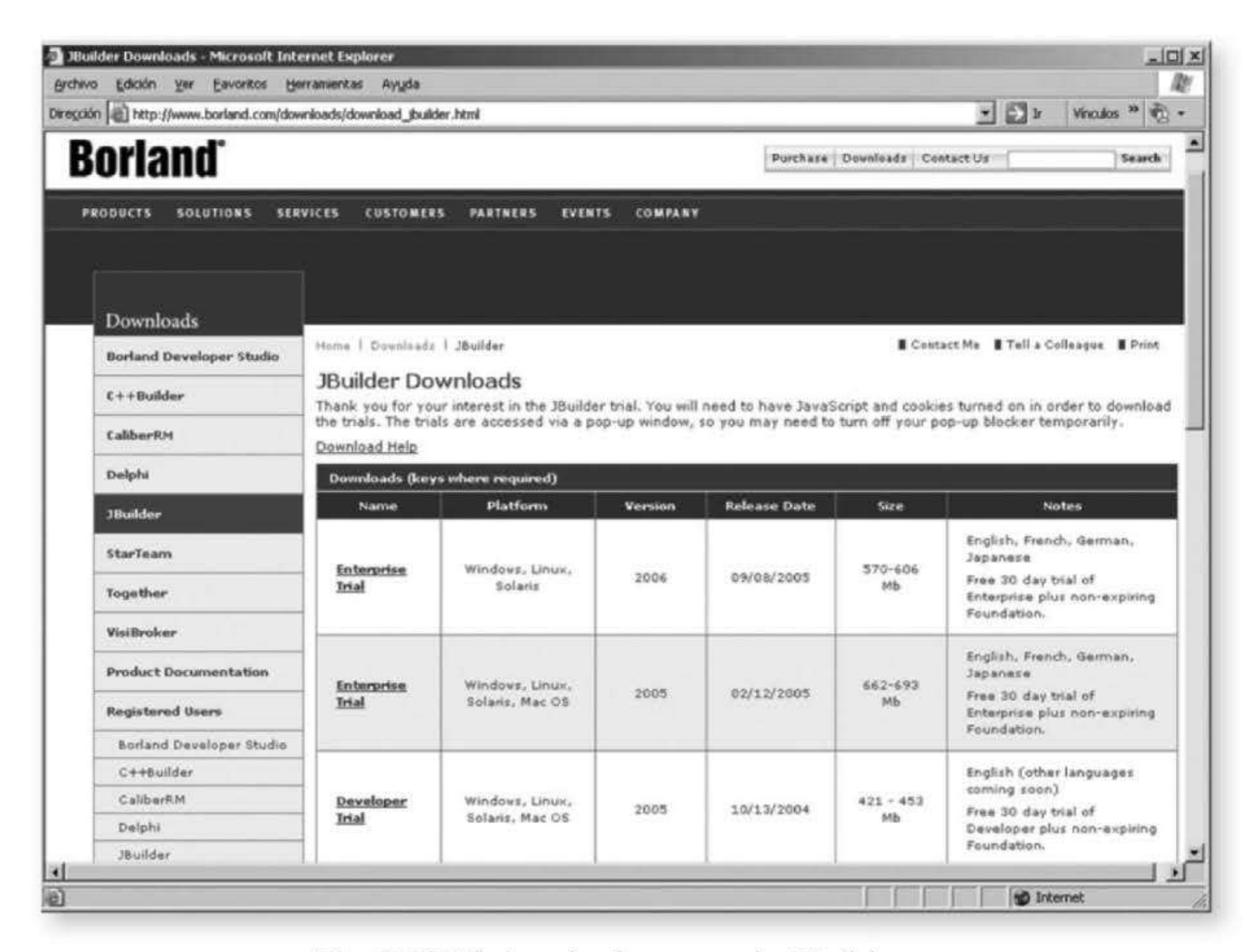

Fig. 3.23 Página de descarga de JBuilder

En esta misma página encontrará el enlace para obtener la clave de activación, se le enviará automáticamente un archivo con la clave de activación a su correo.

Para instalar este IDE debe desempaquetar el archivo ZIP y ejecutar el programa **install_windows.exe**. Se presenta una ventana con varias opciones, ejecute la opción **Borland JBuilder 2005 Developer** y siga los pasos del asistente.

Después de la instalación ya podemos cargar el IDE desde el menú programas (ver Fig. 3.24).

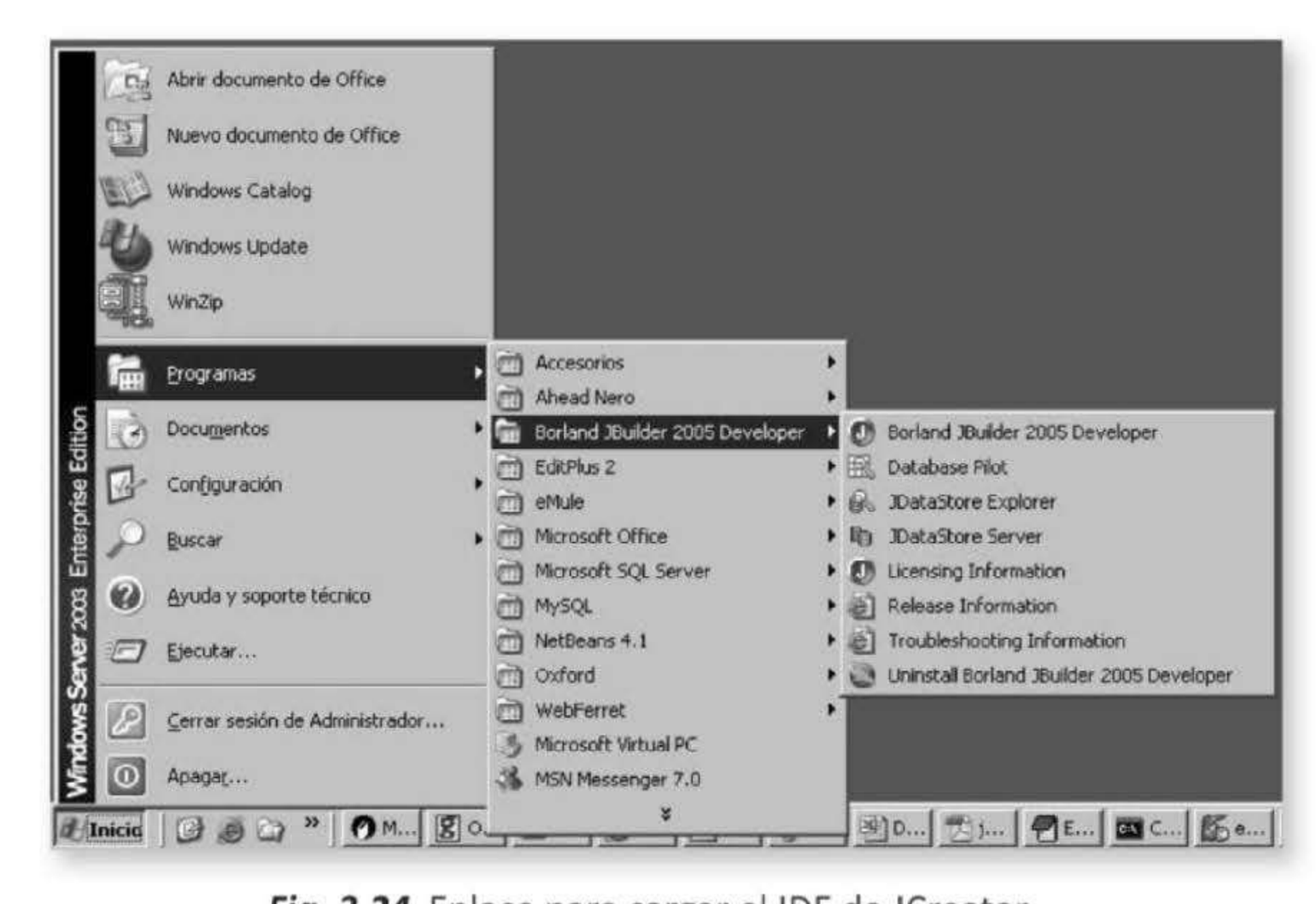

Fig. 3.24 Enlace para cargar el IDE de JCreator

La primera vez que ejecute este producto le mostrará la ventana de activación, seleccione **Have Activation File** y haga clic en el botón **Next**. En la segunda ventana le solicitará el archivo de activación que le ha sido enviado a su correo (tiene que solicitarlo en la página de Borland), y finalmente tendrá el producto activado y listo para empezar a trabajar.

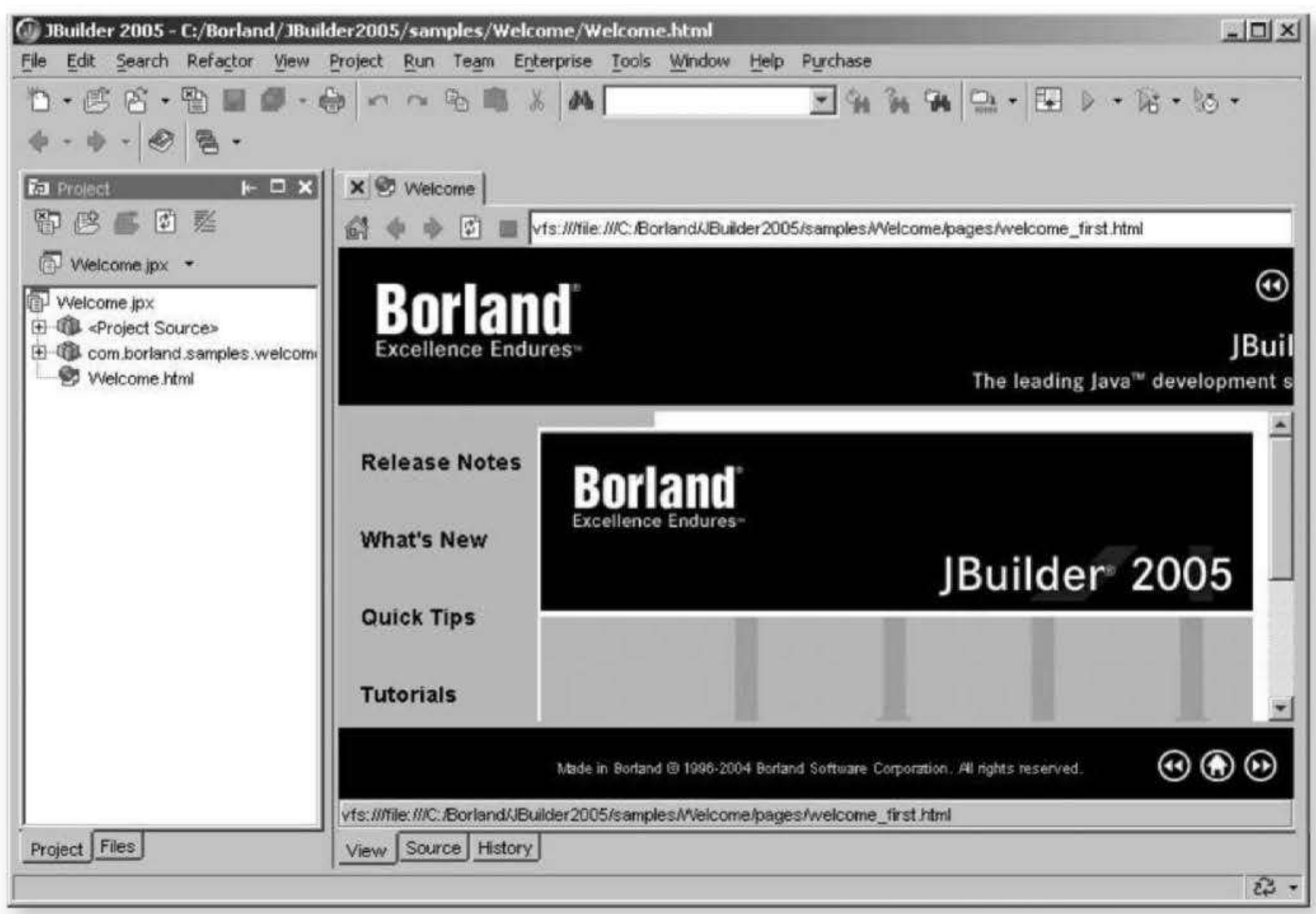

Fig. 3.25 Presentación inicial del IDE de JBuilder

3.7.1 Inicio de un nuevo proyecto

El procedimiento para iniciar un nuevo proyecto es el siguiente:

1. Ejecute el comando **File/New Project**, se carga el asistente para crear nuevos proyectos.
2. En la primera ventana del asistente, ingrese el nombre del proyecto: **EjmJBuilder**, y la ruta donde se grabará el proyecto: **C:/java2n100/cap3/EjmJBuilder**. Finalmente haga clic en el botón **Finish**.

3.7.2 Agregar una nueva clase

Para agregar una nueva clase debe ejecutar el comando **File/New Class** y tenemos el diálogo **Class Wizard**, aquí deberá ingresar el nombre de la clase: **Test**; habilite las opciones (ver Fig. 3.26).

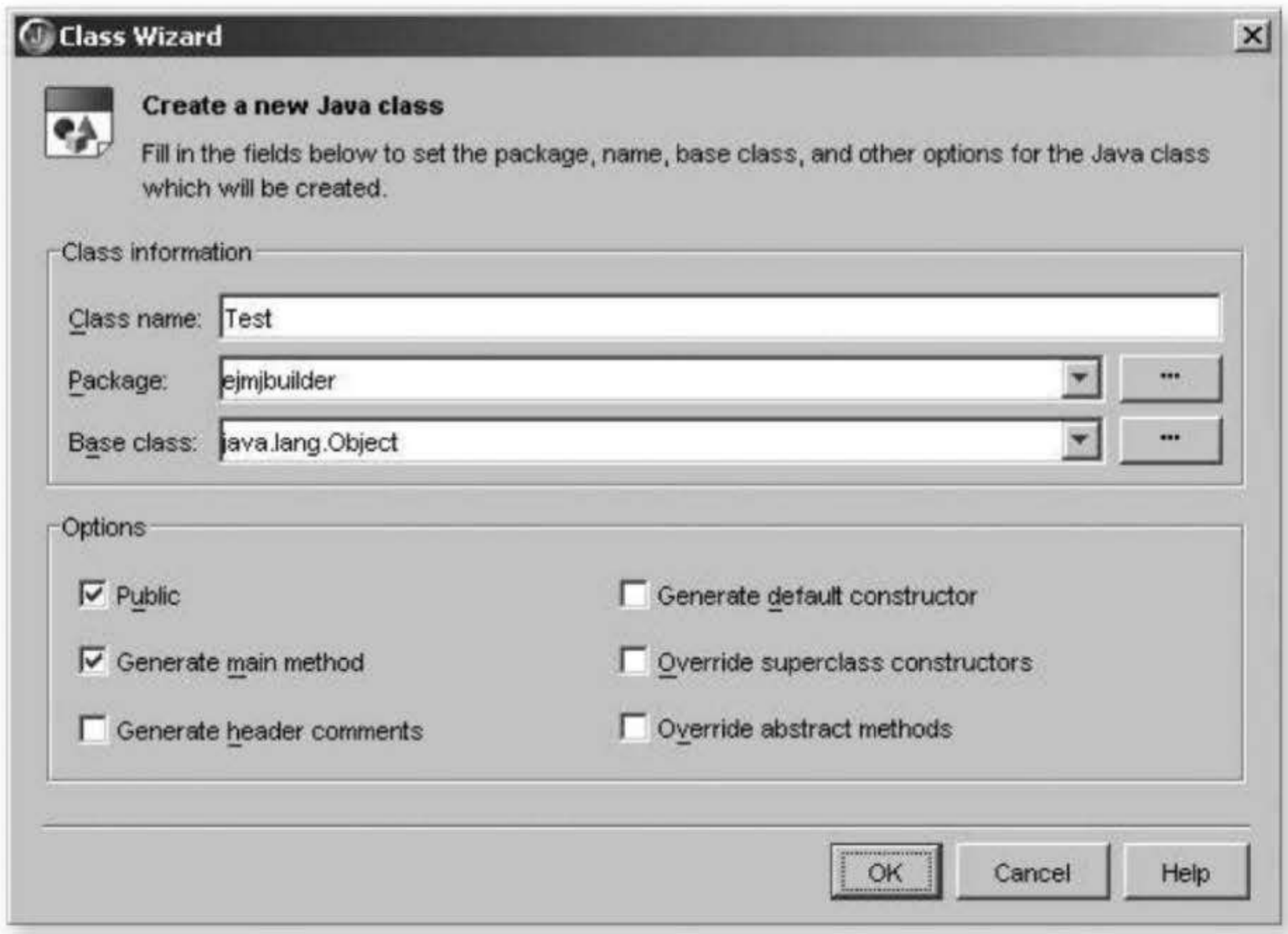

Fig. 3.26 Creación de nueva clase

De esta manera hemos agregado una nueva clase a nuestro proyecto (ver Fig. 3.27).

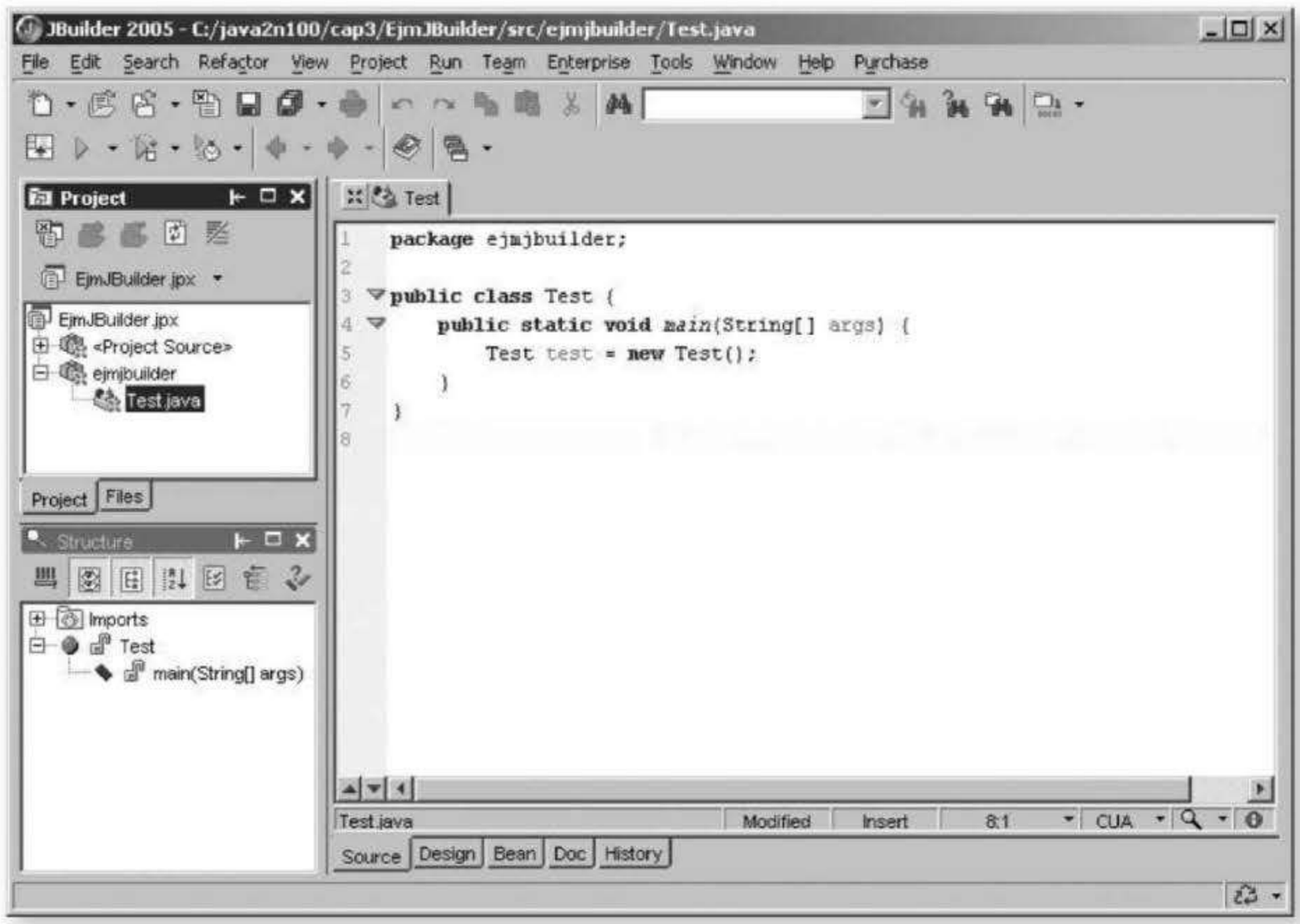

Fig. 3.27 Nueva clase en el proyecto

Modifique el método **main** tal como se ilustra a continuación:

Método: main

```
public static void main(String[] args) {
        Test test = new Test();
        System.out.println("Bienvenido a JBuilder 2005");
        System.out.println("El IDE de los Profesionales");
    }
```

Para ejecutar esta clase debe ejecutar el comando **Run/Run "Test .java"**. El resultado lo puede apreciar en la Fig. 3.28.

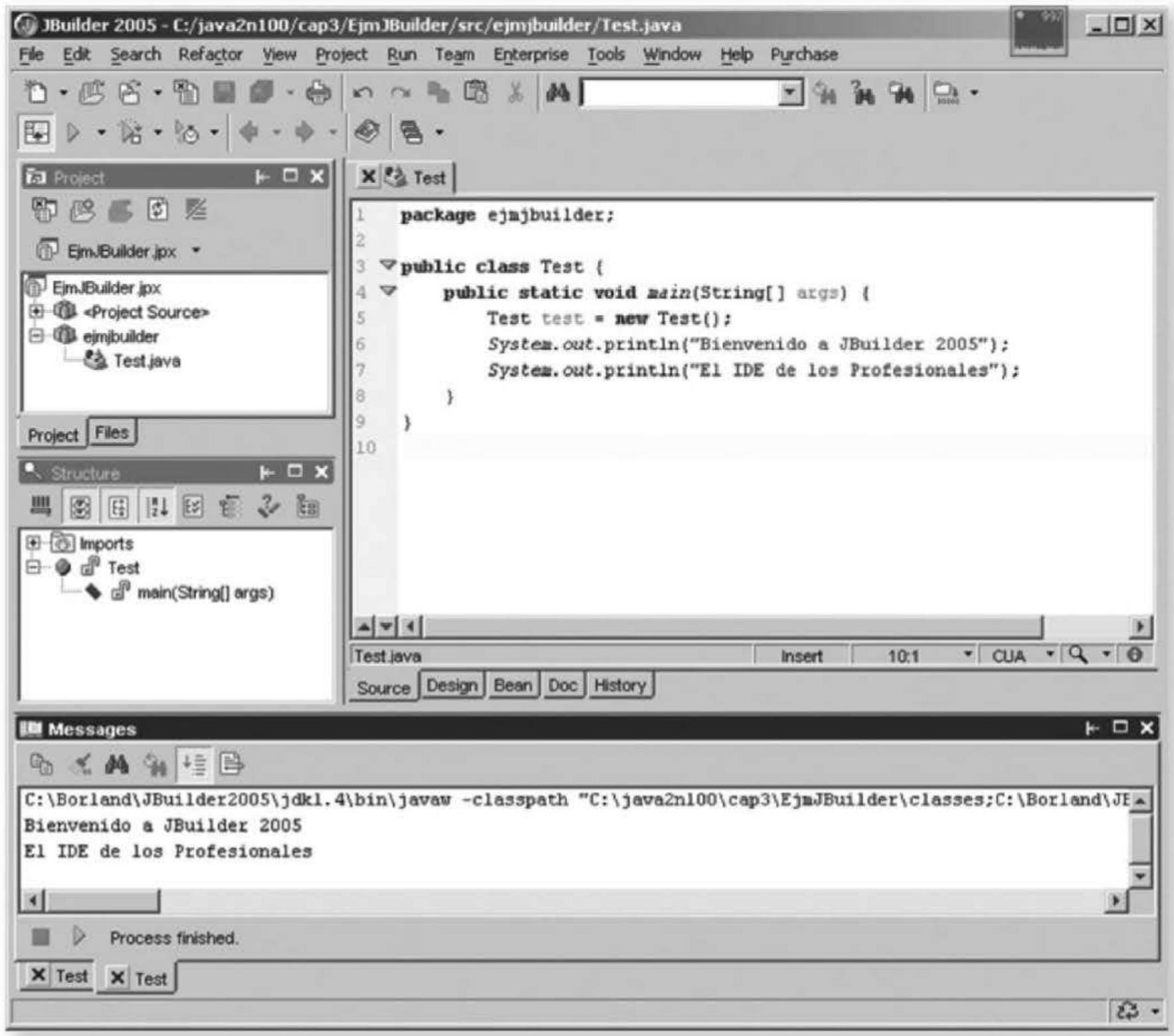

Figura 3.28 Ejecución de la clase Test.

Como puede comprobar existen varios IDE's para trabajar con Java, esta en usted elegir el que más se adecua a su estilo de trabajo, de hecho que existen otros que no he mencionado, pero el objetivo no es mostrar los IDE's que existen en el mercado, sino más bien estudiar el lenguaje.

Fundamentos de la Programación Orientada a Objetos

CAPÍTULO 4

Capacidad: En este capítulo se expondrán todos los fundamentos teóricos de la Programación Orientada a Objetos. Además de las características que posee POO.

La Programación Orientada a Objetos (POO) es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación.

Con la POO tenemos que aprender a pensar las cosas de una manera distinta. Escribir nuestros programas en términos de objetos, propiedades, métodos y otras cosas que veremos rápidamente para aclarar conceptos y dar una pequeña base que nos permita estar relajados con este tipo de programación.

4.1 INTRODUCCIÓN

Si usted tiene los fundamentos claros de la Programación Orientada a Objetos puede prescindir de este capítulo. Lo que se pretende en este apartado es que comprenda los conceptos, no necesariamente que los ejecute en la práctica o dicho de otra manera, implementarlos en un lenguaje como Java.

El concepto de Programación Orientada a Objetos (*Object Oriented Programming*) también conocida como OOP (algunos autores la conocen como POO), agrupa un conjunto de técnicas que nos permiten desarrollar y mantener fácilmente aplicaciones de una gran complejidad.

Estudiaremos los conceptos de Clase, Objeto, Herencia, Encapsulación y Polimorfismo, básicos para toda persona que trabaja o pretende trabajar en OOP; es por lo tanto de suma importancia que los entienda muy claramente.

Es posible que al finalizar la lectura del presente capítulo tenga algunas dudas sobre dichos conceptos, pero no debe preocuparse, ya que se irán afianzando en los siguientes capítulos cuando empiece a crear sus propias clases y jerarquías de clases.

4.2 CONCEPTOS BÁSICOS SOBRE POO

4.2.1 Definición de clase

Una clase es una abstracción que hacemos de nuestra experiencia sensible. El ser humano tiende a agrupar seres o cosas (objetos) con características similares (clases). Así por ejemplo, cuando existe una multitud de vasos diferentes, podemos reconocer un vaso cuando lo vemos, incluso cuando ese modelo concreto de vaso no lo hayamos visto nunca. El concepto de vaso es una abstracción de nuestra experiencia sensible.

Quizás el ejemplo más claro para expresar esto lo tenemos en las taxonomías; los biólogos han dividido a todo ser (vivo o inerte) sobre la Tierra en distintas clases. Tenemos como ejemplo una pequeña porción del inmenso árbol taxonómico en la Figura 4.1.

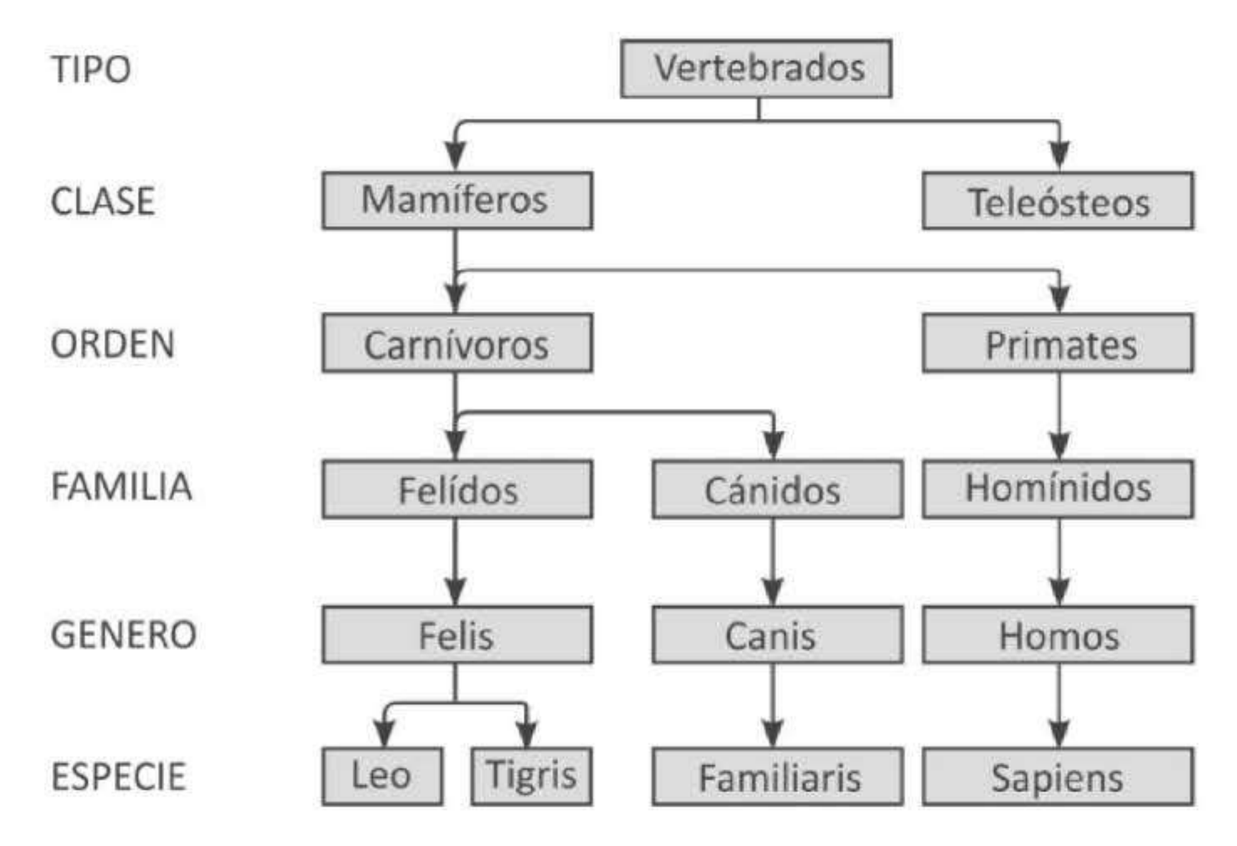

Fig. 4.1 Clasificación taxonómica de los Vertebrados

A cada una de estas parcelas se les denomina reino, tipo, clase, especie, orden, familia, género, etc.; sin embargo, nosotros a todas las llamamos del mismo modo: clase. Así nos referimos la clase animal, vegetal y mineral. Cada clase posee unas cualidades que las diferencian de otras. Por ejemplo, los vegetales se diferencian de los minerales (entre otras muchas cosas) en que los primeros son seres vivos y los minerales no; por otro lado, se diferencian de los animales porque son capaces de sintetizar clorofila a partir de la luz solar.

Como observamos, el ser humano clasifica los objetos del mundo que le rodea en clases; son definiciones estructuralistas de la naturaleza.

Pues bien, situémonos en la clase felinos (*felis*), aquí tenemos varias subclases (géneros en palabras de los biólogos): león, tigre, pantera, gato, etc. Cada una de estas subclases tienen características comunes (por ello los identificamos a todos ellos como felinos) y características que los diferencian (por ello distinguimos a un león de una pantera); sin embargo, ni el león ni la pantera en abstracto existen, existen leones y panteras particulares, pero hemos realizado una abstracción de esos rasgos comunes a todos los elementos de una clase para llegar al concepto de león, pantera o felino.

La clase león se diferencia de la clase pantera en el color de la piel y comparte ciertos atributos con el resto de los felinos (uñas retráctiles, por ejemplo) que lo diferencian del resto de los animales. Pero la clase león, también hereda de las clases superiores ciertas cualidades: columna vertebral (de la clase Vertebrados) y es alimentado en su infancia por leche materna (de la clase mamíferos).

Analizamos cómo las clases superiores son más generales que las inferiores y cómo se va definiendo cada vez más (dotando de cualidades) a las nuevas clases.

Hay cualidades que ciertas clases comparten con otras, pero no son exactamente iguales en las dos. Por ejemplo, la clase hombre también deriva de la clase Vertebrado, pues ambos poseen columna vertebral; sin embargo, mientras que la clase hombre se halla en posición vertical, en la clase león la columna vertebral está en posición horizontal.

En POO existe otro concepto muy importante asociado al de clase, el de «clase abstracta». Una clase abstracta es aquella que construimos para derivar otras clases a partir de ella, pero de la que no se puede instanciar. Por ejemplo, la clase mamífero, no existe como tal en la naturaleza, no existe ningún ser que tan solo sea mamífero (no hay ninguna instanciación directa de esta clase), existen humanos, gatos, conejos, etc., todos ellos son mamíferos, pero no existe un animal que sea solo mamífero.

Del mismo modo, la clase que se halla al inicio de la jerarquía, normalmente es creada solo para que contenga aquellos datos y métodos comunes a todas las clases que derivan de ella, son denominadas clases abstractas. En árboles complejos de jerarquía de clases, suele haber más de una clase abstracta.

Este es un concepto muy importante: el de «clase abstracta». Como se mencionó, una clase abstracta es aquella que construimos para derivar otras clases, pero de la que no se puede instanciar.

Por último, adelantemos algo sobre el concepto de objeto. El león como se explicó, no existe, igual que no existe el hombre; existen leones en los circos, en los zoológicos y, según tenemos entendido, aún quedan algunos en la sabana africana. También existen hombres (hombre en un sentido neutro, ya sea de la subclase mujer o varón) o cada uno de los que nos encontramos a diario en todas partes.

Todos estos hombres comparten las características de la clase hombre, pero son diferentes entre sí, en estatura, pigmentación de la piel, color de ojos, etc. A cada uno de ellos los denominanos «objetos de la clase hombre». Decimos que son objetos de tipo hombre o que pertenecen a la clase hombre. Más técnicamente Ricardo, Marcelo o Sergio Matsukawa son instanciaciones de la clase hombre; instanciar un objeto de una clase es crear un nuevo elemento de esa clase. Cada niño que nace es una nueva instanciación de la clase hombre.

4.2.2 Definición de objeto

En POO, un objeto es un conjunto de datos y métodos. Los datos son los que antes llamamos «características» o «atributos»; los métodos son los comportamientos que pueden realizar. Lo importante de un sistema POO es que ambos, datos y métodos están tan intrínsecamente ligados, que forman una misma unidad conceptual y operacional. En POO no se pueden desligar los datos de los métodos de un objeto, eso mismo ocurre en el mundo real.

Vamos ahora a dar una serie de ejemplos en los que nos acercaremos paulatinamente a los objetos informáticos. Los últimos ejemplos son para los que ya conocen Java; sin embargo, estos exigen conocimientos informáticos pero no son imprescindibles para entender plenamente el concepto de clase y el de objeto.

Observe que aunque los datos y los métodos se han enumerado verticalmente, no existe ninguna correspondencia entre un dato y el método que aparece a su derecha, es una simple enunciación.

- **Ejemplo 1**
  Tomemos la clase león que ya mencionamos y vemos cuáles serían algunos de sus datos y métodos.

Datos	Métodos
Color	Desplazarse
Tamaño	Masticar
Peso	Digerir
Uñas retráctiles	Respirar
Colmillos	Secretar hormonas
Cuadrúpedo	Parpadear
Etc.	Etc.

- **Ejemplo 2**
  Nuestros objetos (los informáticos) se parecen mucho a los del mundo real, porque al igual que estos poseen propiedades (datos) y comportamientos (métodos). Retomemos para nuestro ejemplo un grabador de DVD. Veamos cómo lo definiríamos al estilo POO.

Datos	Métodos
Peso	Abrir puerta
Dimensiones	Cerrar puerta
Color	Grabar
Patencia	Reproducir
Etc.	Etc.

- **Ejemplo 3**
  Pongamos otro ejemplo algo más próximo a los objetos que se suelen tratar en Informática; un recuadro en la pantalla. El recuadro pertenecería a una clase, a la que llamaremos «Marco». Veamos sus datos y sus métodos.

Datos	Métodos
Coordenada superior izquierda	Mostrarse
Coordenada inferior derecha	Ocultarse
Tipo de línea	Cambiar de posición
Color de línea	
Color de relleno	
Etc.	Etc.

- **Ejemplo 4**
  Una clase es un nuevo tipo de dato y objeto a cada una de las asignaciones que hacemos de ese tipo.

```
// Variables
int iEdad;
...
iEdad = 30 ;
```

Hemos creado un objeto que se llama **iEdad** y pertenece a la clase **integer** (entero).

Para crear un objeto de la clase Marco lo haríamos de forma muy parecida, llamando a la función creadora de la clase marco:

```
// Variables
Marco oCajaMsg;
...
oCajaMsg = mew Marco();
```

Para crear un objeto tipo Marco, previamente hemos creado este nuevo tipo de dato. Incluso cuando int no sea internamente un objeto o bien podría serlo; de hecho no hay ninguna diferencia conceptual entre la clase int y la clase Marco. La primera es una de las clases que viene con cualquier lenguaje y la segunda la creamos nosotros. Pero del mismo modo como **iEdad** es una instancia de la clase **int**, o **CajaMsg** es una instancia de la clase Marco. Como se ve, básicamente la POO nos permite ampliar los tipos de datos con los que vamos a trabajar, dicho así resulta simple, aunque si lo piensa un poco, verá que es de una potencia incalculable.

Un objeto lo podemos considerar como un arreglo multidimensional donde se almacenan los datos de ese objeto (coordenadas, color, etc.) y sus métodos, que no serían mas que punteros a funciones que trabajan con esos datos.

Para ser más precisos, en el arreglo de un objeto no se guardan los punteros a los métodos de ese objeto, sino los punteros a otros arreglos donde se hallan estas funciones, para de este modo ahorrar memoria. También se guardan punteros a las clases superiores si las hubiese para buscar allí los métodos que no encontremos en la clase a la que el objeto pertenece: Si un método no es hallado en una clase, se supone que debe pertenecer a la clase padre (clase superior), a esto le llamamos «herencia».

4.2.3 Mensaje

Para que un objeto haga algo, le enviamos una solicitud. Esta hace que se produzca una operación. La operación ejecuta el método apropiado y, de manera opcional, produce una respuesta. El mensaje que constituye la solicitud contiene el nombre del objeto, el nombre de la operación y, opcionalmente, un grupo de parámetros.

La programación orientada por objetos es una forma de diseño modular en donde el mundo se piensa en términos de objeto, operaciones, métodos y mensajes que se transfieren entre tales objetos, como se muestra en la Figura 4.2. Un mensaje es una solicitud para que se lleve a cabo la operación indicada y se produzca el resultado. En consecuencia, la POO se refiere a los mensajes como solicitudes.

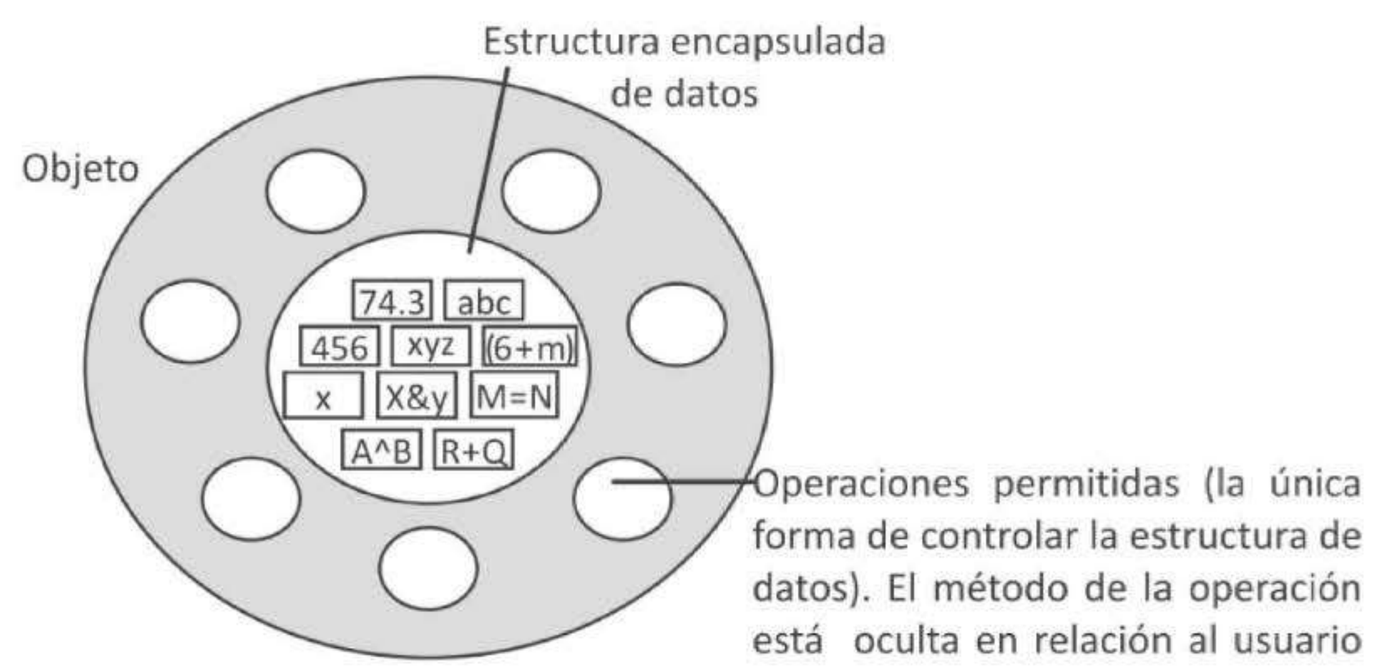

Fig. 4.2 Esquema de un objeto

Una solicitud invoca una operación específica, con uno o más objetos como parámetros.

Los objetos pueden ser muy complejos, puesto que pueden contener muchos subobjetos, estos a su vez pueden contener otros elementos, etc. La persona que utilice un objeto no tiene que conocer su complejidad interna, sino la forma de comunicarse con él y la forma en que responde.

4.3 ENCAPSULACIÓN

Hemos definido antes un objeto como un conjunto de datos y métodos. Dijimos también, que los métodos son procedimientos que trabajan con los datos del objeto. Veamos esto ahora con más detenimiento.

Cuando definimos el concepto de objeto, dimos varios ejemplos de objetos; uno de ellos se refería a un marco (un recuadro) que podría visualizarse en nuestra pantalla. La clase Marco tenía las siguientes características:

Datos	Métodos
Coordenada superior izquierda	Mostrarse
Coordenada inferior derecha	Ocultarse
Tipo de línea	Cambiar de posición
Color de línea	
Color de relleno	

Evidentemente, podríamos (y así deberíamos hacerlo) derivar la clase Marco de la superclase VisualObject, pero para simplificar lo más posible y centrarnos en lo que ahora tratamos, vamos a considerar esta clase como totalmente independiente de las demás.

Los datos de la clase son siempre los mismos para todos los objetos de esa clase, e igualmente los métodos, pero para cada instanciación de la clase (cada uno de los objetos perteneciente a esa clase) los valores de esos datos serán distintos; los métodos trabajarán con cada uno de estos valores de los datos, dependiendo del objeto que se trate.

Veámoslo gráficamente, supongamos que tenemos un monitor en modo texto 80 x 24 y dos objetos de la clase Marco.

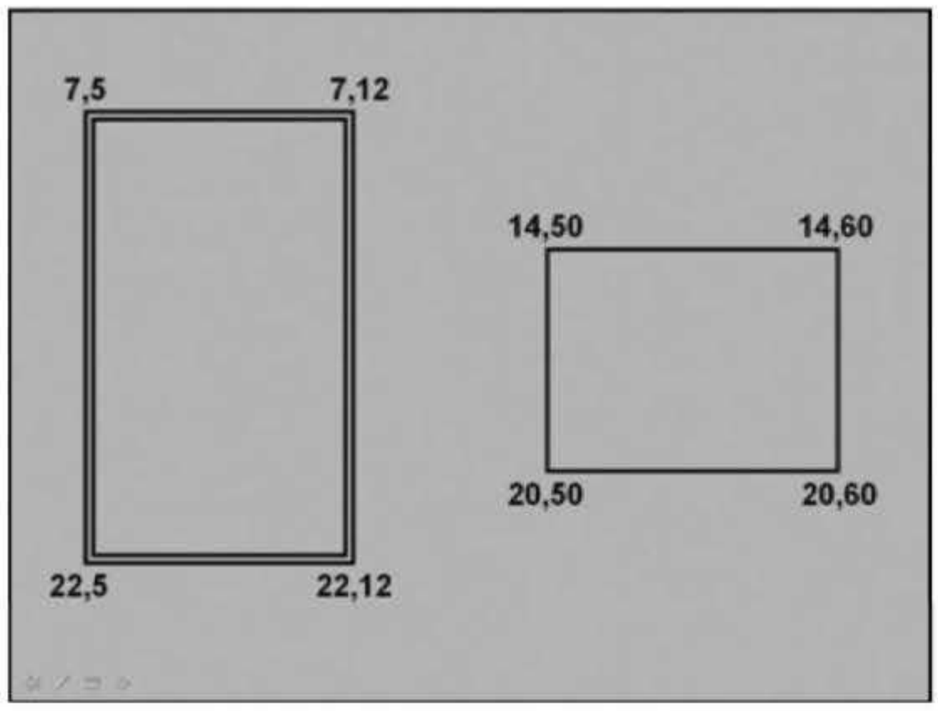

Fig. 4.3 Instanciación de la clase Marco

Hemos puesto en cada una de las esquinas las coordenadas de los vértices de los objetos marco. Estas coordenadas son, desde luego, aproximadas y se las hemos escrito en el siguiente formato: primero la ordenada y luego la abcisa (**<Ypos>,<Xpos>**); situando el origen de coordenadas en el ángulo superior izquierdo de la pantalla con origen 0,0. De hecho, para definir un cuadrado en un plano cartesiano solo es necesario definir los dos vértices superior izquierdo e inferior derecho.

De este modo, podemos apreciar cómo ambos objetos poseen los mismos datos:

marco1	marco2	Datos
7,5	14,50	Coordenada superior izquierda
22,12	20,60	Coordenada inferior derecha
verde	verde	Color de línea
doble	simple	Tipo de línea del recuadro

Los datos son los mismos, pero los valores que toman esos datos son diferentes para cada objeto.

Ahora, podemos aplicar los métodos a los datos. Estos métodos producirán distintos resultados según los datos que se apliquen. Así el objeto **marco1** al aplicarle el método **Mostrarse**, aparece en la parte izquierda, rectangular verticalmente y con línea doble; mientras que el objeto **marco2**, al aplicarle el mismo método, aparece en la parte derecha, cuadrado y con línea simple.

Si quisiéramos ahora aplicar el método **Cambiar de posición** al objeto **marco1**, este debería seguir los siguientes pasos y en el siguiente orden:

1. Llamar al método **Ocultarse** para este objeto.
2. Cambiar los datos de coordenadas para este objeto.
3. Llamar al método **Mostrarse** para este objeto.

Vemos así como un método de una clase puede llamar a otros métodos de su misma clase y cómo puede cambiar los valores de los datos de su misma clase. De hecho así debe hacerse: los datos de una clase solo deben ser alterados por los métodos de su clase y no de forma directa (que es como cambiamos los valores de las variables de un programa).

Esta es una regla de oro que no debe olvidarse:

Todos los datos de una clase son privados y se accede a ellos mediante métodos públicos.

Veamos como se realiza esta acción según los dos modos comentados. Tomemos como ejemplo un objeto perteneciente a la clase **Marco**, modifiquemos su dato **nY1**

(coordenada superior izquierda) de dos modos distintos: directamente y mediante el método poner Y1().

- Cambio directo

```
marco1.nY1 = 12 ;
```

- Cambio mediante invocación a un método

```
marco1.ponerY1( 12 ) ;
```

El primer método es más conveniente, ya que hay que escribir menos para cambiar el valor del dato y además, a la hora de construir la clase, no es necesario crear un método para cambiar cada uno de los datos del objeto. Sin embargo, la POO recomienda que se utilice el segundo procedimiento, por una simple razón: una clase debe ser una estructura cerrada, no debe accederse a ella si no es a través de sus métodos definidos. Si hacemos **nY1** público (para que pueda ser accedido directamente), estaremos violando el principio de encapsulación.

Esta forma de trabajo tiene su razón de ser; ya que en algunos casos puede ser que el método que cambia un dato realiza ciertas acciones o comprobaciones que el programador que está usando un objeto creado por otra persona no conoce, con lo que al manipular los datos del objeto directamente, puede provocar un mal funcionamiento del mismo.

Para evitar que se modifiquen datos que el usuario del objeto no debe tocar, algunos de ellos se hacen de solo lectura, es decir, se puede saber su valor, pero no alterarlo. A estos datos los llamamos *ReadOnly*.

Sin embargo, hay una excepción a esta regla: Pueden hacerse públicos todos los datos que la clase no utiliza. Y usted se preguntará, si la clase no los utiliza: ¿Para qué los quiere? Hay un caso especial en el que al usuario de la clase se le proporciona una «bolsa» (*tag*) para que almacene allí lo que quiera. De hecho, este recurso no es muy ortodoxo, pues la teoría afirma que hay que heredar de la clase y añadir lo que uno necesite; es un recurso muy práctico y cómodo. Dejemos de lado los datos para centrarnos en otros aspectos de los métodos: ¿Cómo requerimos la actuación de un método?

- **Enviando un mensaje al objeto**
  Al enviar un mensaje se ejecuta un método, el cual puede citar a su vez a otros métodos o clases o bien cambiar los valores de los datos de ese objeto. Si el programador tiene que alterar los valores de los datos de un objeto deberá mandar un mensaje a ese objeto; lo mismo sucede cuando un objeto tiene que cambiar los valores de los datos de otro objeto.

Como podemos apreciar, un objeto es como una caja negra a la que se le envía un mensaje y este responde ejecutando el método apropiado; el cual producirá las acciones deseadas. Un objeto una vez programado es solo manipulable a través de mensajes.

A este intrínseco vínculo entre datos y métodos y al modo de acceder y modificar sus datos le llamamos «Encapsulación». Gracias a la encapsulación, una clase cuando ha sido programada y probada hasta comprobar que no tiene fallas, la usamos sin temor a que al programar otros objetos, estos puedan interferir con los primeros produciendo efectos colaterales indeseables que arruinen nuestro trabajo; esto también nos permite depurar (eliminar errores de programación) con suma facilidad, ya que si un objeto falla, el error solo puede estar en esa clase y no en ninguna otra. Si programamos con técnicas tradicionales sabremos apreciar lo que esto vale.

4.4 HERENCIA

Esta es la cualidad más importante de un sistema POO, ya que nos dará mayor potencia y productividad, permitiéndonos ahorrar horas y horas de codificación y de depuración de errores. Es por ello que muchos especialistas en este campo consideran que un lenguaje es POO si incluye esta característica.

La clase león mencionada anteriormente (ver Fig. 4.1), hereda cualidades (métodos, en lenguaje POO) de todas las clases predecesoras (padres, en POO) y posee métodos propios, diferentes a los del resto de las clases. Es decir, las clases van especializándose según se avanza en el árbol taxonómico. Cada vez que creamos una clase heredada de otra (la clase padre) añadimos métodos a la clase padre o modificamos alguno de los métodos de la clase padre.

Veamos qué hereda la clase león de sus clases padre:

Clase	Qué hereda
Vertebrado	Espina dorsal
Mamíferos	Se alimenta con leche materna
Carnívoros	Al ser adultos se alimentan de carne

La clase león hereda todos los métodos de la clase padre y añade nuevos métodos que forman su clase distinguiéndola del resto de las clases, por ejemplo el color de piel.

Observe ahora algo crucial que ya se mencionó: dos subclases distintas, que derivan de una misma clase padre común pueden heredar sus métodos y como estos han sido definidos en ella, o pueden modificar todos o algunos de estos métodos para adaptarlos a sus necesidades.

En el ejemplo expuesto, la alimentación de la clase león es carnívora, mientras que en la clase hombre se ha modificado este dato, siendo su alimentación omnívora.

Pongamos ahora otro ejemplo: Supongamos que hemos construido una clase que permite leer números enteros desde el teclado con un formato determinado. Calcular su IGV y almacenarlo en un fichero. Si deseamos poder hacer lo mismo con números reales (para que admitan decimales), debemos crear una nueva subclase que herede de la clase padre todos sus métodos y redefinir el método de lectura del teclado. Esta nueva clase almacena y muestra los números con formato porque su padre lo sabe.

Las cualidades comunes que comparten las distintas clases, pueden y deben agruparse para formar una clase padre (también llamada «superclase»). Por ejemplo, usted podría derivar las clases presupuesto, guía y factura de la superclase pedidos, ya que estas clases comparten características comunes. De este modo, la clase padre poseerá los métodos comunes a todas ellas y solo se añadirá aquellos métodos propios de cada una de las subclases, reutilizándose el código escrito en la superclase desde cada una de las clases derivadas. Entonces si enseñamos a la clase padre a imprimirse, cada uno de los objetos de las clases inferiores sabrán imprimirse automáticamente (sin escribir ni una sola línea más de código). Esto es muy interesante.

La herencia es la cualidad más importante de la POO, debido a que le permite reutilizar todo el código escrito para las superclases reescribiendo aquellas diferencias que existen entre estas y las subclases.

A la clase heredada se le llama «subclase» o «clase hija», y a la clase de la que se hereda «superclase» o «clase padre». Al heredar, la clase heredada toma directamente el comportamiento de su superclase; una clase toma indirectamente el comportamiento de todas las clases de la rama del árbol de la jerarquía de clases a la que pertenece.

Se heredan los datos y los métodos, por lo tanto, ambos pueden ser redefinidos en las clases hijas, aunque lo más común es redefinir métodos y no datos.

Muchas veces las clases (especialmente aquellas que se encuentran próximas a la raíz en el árbol de la jerarquía de clases) son abstractas; es decir existen para proporcionar una base para la creación de clases más específicas. Por lo tanto, no puede instanciarse de ellas: son las clases virtuales.

Una subclase hereda de su superclase aquellos miembros visibles desde la clase hija y, por lo tanto, puede redefinir estos. Tiene forzosamente que redefinir aquellos métodos que han sido definidos como abstractos en la clase padre. Normalmente, es necesario redefinir datos de las clases superiores, sustituir el funcionamiento del método de la clase padre o bien ampliarlo.

En el primer caso (sustituirlo) no hay ningún problema porque a la clase hija la dotamos con un método de igual nombre que el método que queremos redefinir en la clase padre

y lo implementamos según las necesidades de la clase hija. De este modo, cada vez que se invoque este método se ejecutará su código, y no el código escrito para el método homónimo de la clase padre.

Pero si queremos ampliar el funcionamiento de un método existente en la clase padre (lo que suele ser lo más habitual), entonces tiene que ejecutarse el método de la clase padre, y luego la clase hija. Pero como los dos métodos tienen el mismo nombre, se hace necesario habilitar alguna forma de distinguir cuando nos referimos a un método de la clase hija y cuando es el mismo nombre de la clase padre.

Esto se hace mediante el uso de dos palabras reservadas, las cuales pueden variar dependiendo del lenguaje que se utilice, pero que normalmente son: **this** y **super**.

4.4.1 This

Se refiere a los miembros de la clase. De hecho, siempre que dentro del cuerpo de un método aludimos a cualquier miembro de la clase, ya sea una variable u otro método, podemos anteponer **this**; aunque en caso de no existir duplicidad, el compilador asume que nos referimos a un miembro de la clase.

Algunos programadores prefieren utilizar **this** para dejar claro que se está mencionando a un miembro de la clase.

4.4.2 Super

Hace referencia a miembros de la clase padre (o a los ancestros anteriores, que no hayan sido ocultados por la clase padre) que se hayan redefinidos en la clase hija.

Si un método de una clase hija redefine un miembro (ya sea variable o método) de su clase padre, es posible mencionar al miembro redefinido anteponiendo **super**.

Veamos un ejemplo, suponemos que solo los datos especificados en la clase A son visibles desde la clase B:

Clase A			
Datos	**Visible**	**Métodos**	**Visible**
AD1	No	AM1	No
AD2	Sí	AM2	Sí
AD3	Sí	AM3	Sí

La clase B hereda de la clase A:

Clase B			
Datos	Heredados	Métodos	Heredados
AD2	Sí	AM2	Sí
AD3	Sí	AM3	Sí
BD1	No	BM1	No
BD2	No	BM2	No

Si en la clase B quisiéramos señalar al método **AM2** de la clase A ampliándolo tendríamos que referirnos a él mediante **super**, tal como se aprecia en el siguiente **script**:

```
public class ClaseA
{
    public void AM2()
    {
            . . .
            . . .
    }

    . . .
    . . .
}

public class ClaseB extends ClaseA
{
    public void AM2()
    {
            super.AM2(); // Invoca la ejecución del método AM2 de la clase
padre Clase A

            . . .  // Resto de la implementación de AM2 en la Clase B
            . . .
    }

    . . .
    . . .
}
```

4.5 POLIMORFISMO

Por polimorfismo entendemos aquella cualidad que poseen los objetos para responder de distinto modo al mismo mensaje.

Por ejemplo las clases hombre, vaca y perro; si a todos damos la orden (envíanos el mensaje) **Come**, cada uno de ellos sabe cómo hacerlo y realizará este comportamiento a su modo.

Veamos otro ejemplo. Tomemos las clases barco, avión y coche; todas derivadas de la clase padre Vehículo. Si les enviamos el mensaje **Desplázate**, cada una sabe cómo hacerlo.

Para ser exactos, los mensajes no se envían a las clases, sino a los objetos o algunos de los objetos instanciados de la clases. De este modo, por ejemplo, podemos decirle a los objetos Juan Sebastián, El Cano y Kontiqui, de la clase barco que se desplacen, con lo que el resto de esa clase permanecerá inmóvil.

Del mismo modo, si tenemos en pantalla cinco recuadros (marcos) y tres textos, podemos decirle a tres de los recuadros y a dos de los textos que cambien de color y no decírselo a los demás objetos. Todos estos sabrán cómo hacerlo porque hemos redefinido para cada uno de ellos su método **Pintarse** que bien podría estar en la clase padre **VisualObject** (conjunto de objetos que pueden visualizarse en pantalla).

En programación tradicional, debemos crear un nombre distinto para la acción de pintarse, si se trata de un texto o de un marco; en POO el mismo nombre nos sirve para todas las clases creadas si así lo queremos. El mismo nombre suele usarse para realizar acciones similares en clases diferentes.

Si enviamos el mensaje **Imprímete** a objetos de distintas clases, cada uno se imprime como le corresponda, ya que todos saben cómo hacerlo. El polimorfismo nos facilita el trabajo, debido a que el número de nombres de métodos que tenemos que recordar disminuye visiblemente.

La mayor ventaja la obtendremos en métodos con igual nombre aplicados a las clases que se encuentran próximas a la raíz del árbol de clases, ya que estos afectan a todas las clases que derivan de ellas.

- **Sobrecarga**
  La sobrecarga puede ser considerada como un tipo especial de polimorfismo que casi todos los lenguajes de POO incluyen.

  Varios métodos (incluido los «constructores») pueden tener el mismo nombre siempre y cuando el tipo de parámetros que recibe o el número de ellos sea diferente. De este modo, por ejemplo la clase **File** puede tener tantos métodos **write()** como tipos de datos queramos escribir.

oFile.write(int i) ;	Escribe un integer
oFile.write(long l) ;	Escribe un long
oFile.write(float f) ;	Escribe un single
oFile.write(String str) ;	Escribe una cadena
oFile.write(String str, boolean b) ;	Escribe una cadena pasándola a mayúsculas

Cuando invocamos el método write, el mismo lenguaje sabrá a cuál de ellos nos estamos refiriendo en función al parámetro ingresado.

4.6 PLANTEAMIENTO DE LA IMPLEMENTACIÓN

Cuando desarrollamos una aplicación y deseamos utilizar POO, lo más importante es planificar bien el árbol (jerarquía) de clases.

Una buena planificación de cada uno de los objetos y métodos que debe incluir cada una de las clases, quién deriva de quién y cómo se interrelacionan es lo más importante para que un sistema de clases funcione correctamente. Respecto a esto no hay ninguna regla invariable que seguir, pues es más arte que ciencia.

Para que tenga una idea de la importancia que tiene este diseño, es preciso saber que en los equipos de programadores que trabajan en POO suelen participar varias personas que se dedican exclusivamente al diseño de esta estructura de clases (los datos que cada una de ellas contenga, los métodos que trabajarán con esos datos y las interrelaciones de unas clases con otras). Estas personas trabajan con herramientas especializadas en facilitar el desarrollo de jerarquías de clases, siendo la más conocida **Rational XDE** (antes Rational Rose). Pero incluso con la ayuda de herramientas de este tipo, la definición de clases y sus relaciones sigue siendo más un arte que una ciencia. Y en cualquier caso, nunca el modelo diseñado sobre el papel o la herramienta es el modelo finalmente implementado; de hecho lo más común es definir un modelo inicial e ir rehaciéndolo, una y otra vez, según se va implementando.

Sin embargo, y pese a todo lo dicho, hay unos recomendaciones muy generales que pueden ayudar a la hora de crear estos modelos previos a la implementación de las clases.

La mayor parte de las personas que estudia en un área tecnológica cualquiera, busca técnicas para aplicar inmediatamente en la resolución de nuestros problemas. Sin embargo, son valiosos los consejos de personas que llevan muchos años trabajando en esas áreas, incluso cuando no sepamos apreciar su valor. Dichas recomendaciones las encontramos también en la mayoría de textos y son producto de la experiencia de los autores; por lo tanto están perfectamente verificados.

Señalaré tres consejos muy simples que son necesarios:

- **Divide y vencerás**
  Procure construir un método para realizar cada pequeña tarea que necesite. Cuanto menor sea el ámbito de un método, le resultará reutilizable en otro momento y más fácil de codificarlo. Haga lo mismo con las clases; no escatime en su número ante la duda, siempre es mejor dividir una clase en dos que agrupar dos en una.

- **No piense en forma procedural**
  En POO no se puede pensar en forma procedural, una clase no es un conjunto de funciones relacionadas, es un objeto tan real como los demás que están fuera del ordenador. Los objetos son de verdad, no son cajas de almacenamiento de códigos.
- **Los métodos no son funciones**
  No defina métodos como si fueran funciones, sino como acciones inherentes al objeto; así cambiará el funcionamiento interno completamente sin que cambie ni el nombre de la clase ni los nombres de los métodos. Si al cambiar el funcionamiento de una clase, el nombre de la clase y de los métodos no tienen sentido; entonces estaba mal diseñada.

Ahora indicaré los pasos a seguir para plantear el diseño de las clases:

1. Identificar el ámbito de trabajo; pues verá qué clases están en la raíz de la jerarquía (las más abstractas).
2. Identificar los distintos subámbitos de trabajo.
3. Especificar los objetos finales, las relaciones entre clases del mismo ámbito y las relaciones con clases de los otros ámbitos.

4.7 DISEÑO DE LA ESTRUCTURA Y EL COMPORTAMIENTO DE UN OBJETO

El diseño de la estructura de objetos tiene dos aspectos: el diseño de la estructura del objeto (DEO) y el diseño del comportamiento del objeto (DCO). Los lenguajes de programación OO tienen estructuras de datos y métodos, ambos están enlazados, y se describirán en forma conjunta.

Se identifican los siguientes componentes:

- ¿Qué clases se implantarán?
- ¿Qué estructura de datos utilizará cada clase? Se puede hacer un diagrama para representar la estructura de datos.
- ¿Qué operaciones ofrecerá cada clase y cuáles serán sus métodos? Se enumeran las operaciones y se especifican sus métodos en determinado momento.
- ¿Cómo se implementará la herencia de clases y cómo afectará esta a las especificaciones de los datos y las operaciones?
- ¿Cuáles son las variantes? Se identifican las probables variantes de las clases : «Igual que, excepto...», aplicada a la mayoría de los comportamientos reutilizables.

4.7.1 Clase

Clase es la implantación de un tipo de objeto. Especifica las estructuras de datos y los métodos operativos permitidos aplicados a cada uno de sus objetos.

La clase especifica la estructura de datos en cada uno de sus objetos y las operaciones que se utilizan para tener acceso a los objetos (Fig. 4.4). La especificación de cómo se llevan a cabo las funciones de una clase se llama «método». Los objetos se pueden utilizar exclusivamente como métodos específicos.

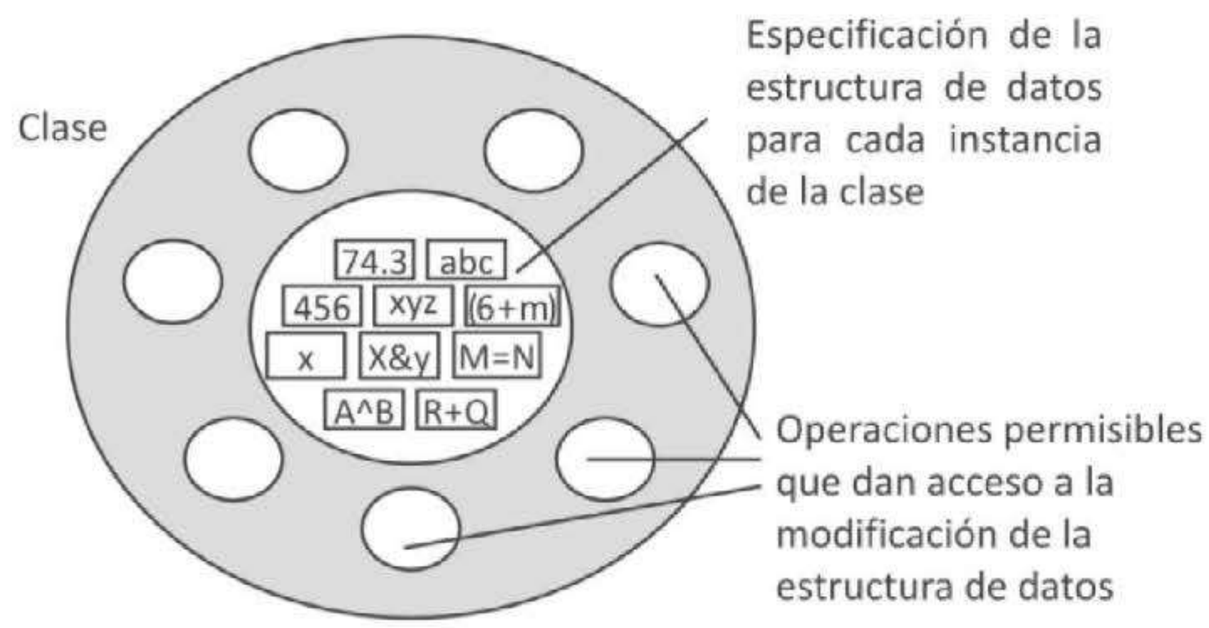

Fig. 4.4 Cada clase especifica una estructura de datos, las operaciones permisibles y los métodos operativos

La Fig. 4.5 muestra un objeto que es instancia de la clase de la Fig. 4.4. Los datos y operaciones que encapsula quedan especificados por su clase. Los datos del objeto se almacenan dentro de él y se tiene acceso a ellos modificándolos mediante las operaciones permisibles. Esta restricción al acceso se debe al encapsulado, que protege los datos del uso arbitrario o no pretendido. El acceso o la actualización directa de los datos de un objeto por parte del usuario violarían el encapsulado. Los usuarios observan el comportamiento del objeto en términos de las operaciones que se pueden aplicar a los objetos, así como los resultados de tales operaciones, las cuales forman el interfaz del objeto con sus usuarios.

4.7.2 Diferencia entre operación y método

Las operaciones son procesos que se pueden solicitar como unidades; mientras que los métodos son especificaciones del procedimiento de una operación dentro de una clase. Es decir, la operación es el tipo de servicio solicitado y el método es su código de programación.

- Una operación es un proceso que se puede solicitar como unidad.
- Un método es la especificación de una operación.

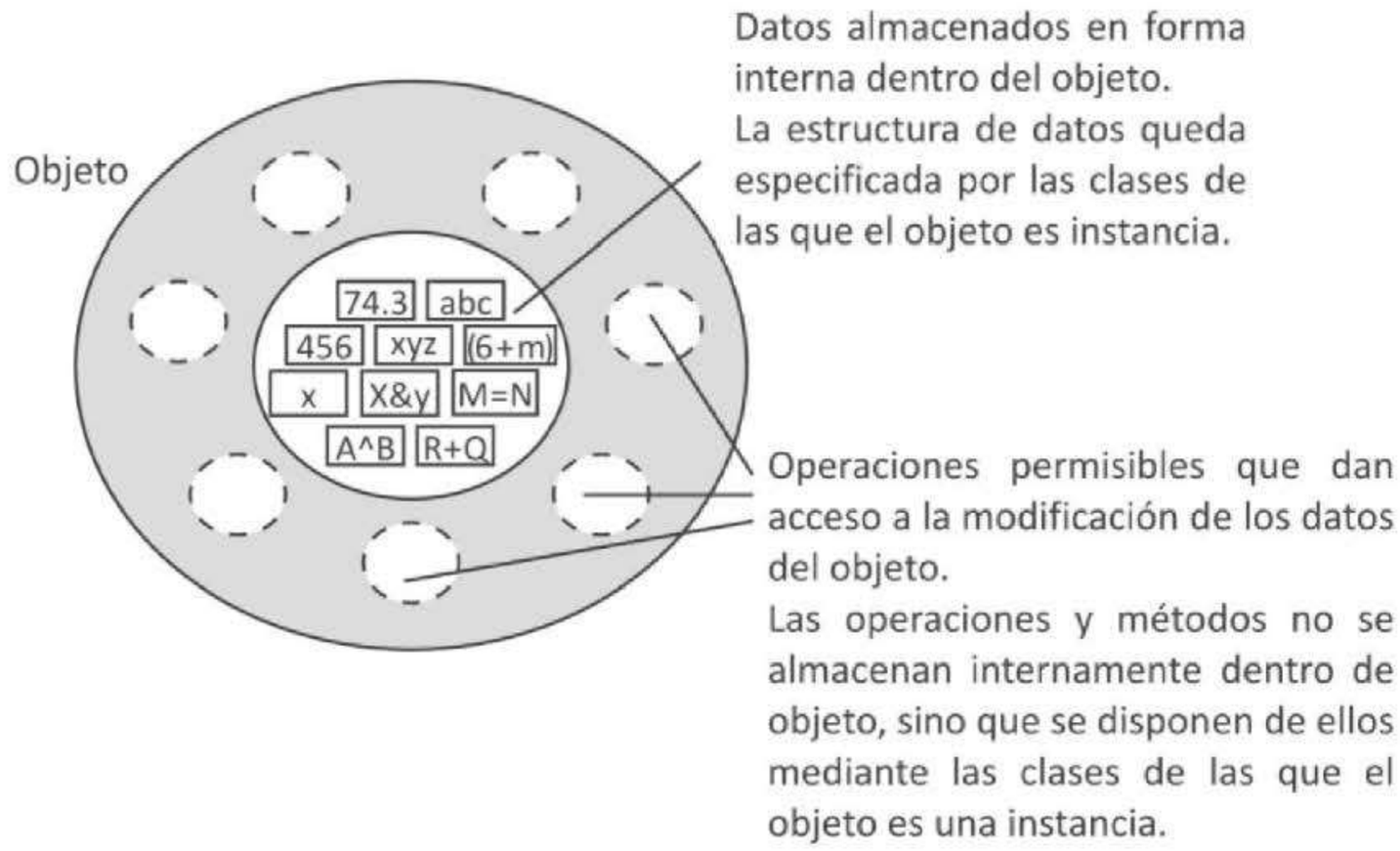

Fig. 4.5 El encapsulado oculta la implementación de la estructura de datos y métodos de cada objeto. Los usuarios solo conocen las operaciones permisibles del objeto, la interfaz

Por ejemplo, una operación asociada con la clase pedido podría ser aquella que calcule el total del pedido. El método especifica la forma de calcular el total. Para esto, el método podría obtener el precio de cada artículo del pedido al enviar una solicitud a los objetos artículos asociados. A su vez, cada objeto artículo regresaría su precio al objeto pedido mediante un método de la clase artículo.

Los métodos de una clase controlan solamente a los objetos de esa clase. No pueden tener acceso directo a las estructuras de datos de un objeto en una clase distinta. Para utilizar las estructuras de datos de una clase diferente deberán enviar solicitudes a ese objeto.

El encapsulado significa que los métodos de una clase protegen a todos los objetos que se encuentran dentro de esa clase.

Fundamentos de lenguaje

CAPÍTULO 5

Capacidad: En este capítulo usted empezará a crear pequeños programas en Java, utilizando los diferentes operadores y sentencias de asignación.

Para el desarrollo de los ejemplos no es necesario un IDE, puede desarrollarlos con un editor de texto simple como el bloc de notas, **EditPlus** u otro de su preferencia.

5.1 ESQUEMA DE UN PROGRAMA JAVA

Un programa básico en Java tiene el siguiente esquema:

```
public class  NombreClase
{
   public static void main(String[] args)
   {
    Instrucciones;
   }
}
```

Existen dos reglas que debe tener en cuenta:

1. El nombre del programa debe ser igual al nombre de la clase seguido de la extensión **.java**.
2. El método main inicia la ejecución del programa.

- **Ejemplo 1**

Se observa la aplicación de las dos reglas enunciadas anteriormente.

prog0501.java

```
public class prog0501
{

     public static void main(String[] args)
     {
             int a = 10;
             int b = 18;
             int c;
c = prog0501.suma(a,b);
             System.out.println("a = " + a);
             System.out.println("b = " + b);
             System.out.println("c = " + c);
     }

     static int suma( int n1, int n2 ){
             return (n1 + n2);
     }

} // Fin de class
```

A continuación, la compilación y ejecución del programa desde la línea de comando:

```
C:\java2n100\cap5>javac prog0501.java          [Enter]

C:\java2n100\cap5>java prog0501                [Enter]
a = 10
b = 18
c = 28
```

5.1.1 Instanciación de una clase

Para que una clase pueda ser utilizada debe ser instanciada, es decir debemos crear el objeto.

Para crear la instancia de una clase, primero se debe definir la variable, y luego crear la instancia, la sintaxis se especifica a continuación:

```
NombreClase NombreVariable;
NombreVariable = new NombreClase();
```

También puede definir la variable e instanciar la clase en la misma instrucción, tal como se ilustra en la siguiente sintaxis:

```
NombreClase Nombre Variable = new NombreClase();
```

En el siguiente ejemplo se ilustra la instanciación de la clase **Msg** y luego la ejecución del método say().

```
NMsg obj = new Msg();
obj.say( "Bienvenido al mundo de Java." );
```

- **Ejemplo 2**
  Veremos la instanciación de la clase **Msg** para poder ejecutar el método **say**.

  El siguiente es el listado de la clase **Msg**.

Msg. java

```
public class  Msg
{
    public void say(String cad)
    {
            System.out.println( cad );
    }
}
```

El siguiente es el listado de la clase **prog0502.java**.

prog0502.java

```
public class prog0502
{
    public static void main(String[] args)
    {
            Msg obj = new Msg();
            for(int n = 1, m = 10; n <= m; n++, m-- ) {
                  String cad;
             cad = n + " * " + m + " = " + producto(n,m);
             obj.say( cad );
            }
    }

    static int producto( int n1, int n2 ){
            return (n1 * n2);
    }
}
```

Compilación de la clase **prog0502.java**:

```
C:\java2n100\cap5>javac prog0502.java          [Enter]

C:\java2n100\cap5>dir                          [Enter]
 El volumen de la unidad C es Main
 El número de serie del volumen es: 6038-0176

 Directorio de C:\java2n100\cap5

08/11/2005  12:32 a.m.    <DIR>          .
08/11/2005  12:32 a.m.    <DIR>          ..
08/11/2005  12:32 a.m.               367 Msg.class
08/11/2005  12:28 a.m.                94 Msg.java
07/11/2005  12:32 a.m.               786 prog0501.class
08/11/2005  12:22 a.m.               316 prog0501.java
08/11/2005  12:33 a.m.               681 prog0502.class
08/11/2005  12:33 a.m.               308 prog0502.java
               6 archivos          2,552 bytes
               2 dirs   2,058,637,312 bytes libres
```

Podemos apreciar que no ha sido necesario compilar la clase **Msg.java**, se ha compilado automáticamente; debido a que dentro de la programación de la clase **prog0502** hacemos referencia a la clase **Msg**.

Ejecución de la clase **prog0502.class**:

```
C:\java2n100\cap5>java prog0502       [Enter]
1 * 10 = 10
2 * 9 = 18
3 * 8 = 24
4 * 7 = 28
5 * 6 = 30
```

5.1.2 Argumentos del método Main

El método **main** dentro de una clase indica que podemos ejecutar esta clase. Toda aplicación en Java inicia ejecutando un método **main**,

A continuación presentamos la estructura del método **main**:

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

Además, esta es la descripción de las partes de la estructura del método **main**:

public	Indica que el método **main** puede ser invocado por cualquier objeto.
static	Indica que el método **main** es un método estático, es decir, un método propio de la clase, y puede ser invocado sin necesidad de instanciar la clase.
void	Indica que el método **main** no devolverá ningún valor
String[] args	**args** es un arreglo de strings a través del cual el método **main** puede aceptar varios parámetros.

Los argumentos se pasan en la misma línea de comando, como se ilustra a continuación:

```
C:\>java programa arg1 arg2 arg3 . . .                    [Enter]
```

- **Acceso a los argumentos de main**
  Podemos acceder en forma directa:

```
args[n]
```

En este caso n es el índice, que toma valor 0 para el primer argumento, 1 para el segundo y así sucesivamente.

También se utiliza un bucle for para acceder a los argumentos. A continuación tenemos el esquema para recorrer los argumentos:

```
for (int j = 0; j < args.length; j++)

      // Instrucciones

}
```

- **Ejemplo 3**
  Se pasará un parámetro a un programa correspondiente al nombre de una persona y luego el programa imprimirá un saludo.

  El siguiente es el listado del programa **prog0503.java**.

prog0503.java

```
public class prog0503
{
    public static void main(String[] args)
    {
            prog0503 obj = new prog0503();
            System.out.println("Hola " + args[0]);
    }
}
```

Compilación y ejecución del programa:

```
C:\java2n100\cap5>javac prog0503.java          [Enter]

C:\java2n100\cap5>java prog0503 Gustavo        [Enter]
Hola Gustavo
```

- **Ejemplo 4**

El siguiente programa recibe «**n**» números y después de calcular la suma de ellos, imprime el resultado obtenido.

prog0504.java

```
public class prog0504
{
     public static void main(String[] args)
     {
           int suma = 0;
           for( int j = 0; j < args.length; j++ ){
             System.out.println("Numero " + (j+1) + ": " + args[j]);
             suma += Integer.parseInt( args[j] );
           }
           System.out.println("La suma es: " + suma);
     }
}
```

Compilación y ejecución del programa:

```
C:\java2n100\cap5>javac prog0504.java          [Enter]

C:\java2n100\cap5>java prog0504 15 17 13 20    [Enter]
Numero 1: 15
Numero 2: 17
Numero 3: 13
Numero 4: 20
La suma es: 65
```

Note que el índice del arreglo **args** inicia en 0; de esta manera, el elemento 0 corresponde al número 1, el elemento 1 corresponde al número 2, y así sucesivamente.

5.2 ELEMENTOS BÁSICOS DEL LENGUAJE

5.2.1 IDENTIFICADORES

Un identificador es el nombre que se le da a una variable, clase o método. Para crear sus identificadores debe tener en cuenta las siguientes reglas:

1. Deben iniciar con una letra, el carácter subrayado <_> o el signo dólar <$>.
2. Los siguientes caracteres pueden también contener dígitos.
3. Los identificares son **case-sensitive**; esto quiere decir que hay diferencia entre mayúsculas y minúsculas.
4. No se pueden utilizar palabras claves.

Los siguientes son ejemplos de identificadores válidos:

- **nombre_usuario**
- **$tipo**
- **nombreUsuario**
- **articulo**
- **_sist_fecha**

Los siguientes son ejemplos de identificadores no válidos:

- **56_tema**
- **nombre@articulo**
- **tema-siguiente**

5.2.2 PALABRAS CLAVES E IDENTIFICADORES RESERVADOS

Las palabras claves e Identificadores reservados que son predefinidos en Java no pueden ser utilizados para definir nombres de variables, clases o métodos. Todas estas palabras claves están en minúsculas y su uso incorrecto generan errores de compilación.

Tabla 5.1 *Palabras claves en Java*

abstract	default	implements	protected	throw
assert	do	import	public	throws
bolean	double	instanceof	return	transient
break	else	int	short	try
byte	extends	interface	static	void
case	final	long	strictfp	volatile
match	finally	native	super	while
char	flota	new	switch	
class	for	package	synchronized	
continue	if	private	this	

Tabla 5.2 *Identificadores reservados en Java*

null	true	false		

- **Ejemplo 5**
  Se puede apreciar el mensaje de error que muestra Java cuando tratamos de redefinir una palabra clave.

prog0505.java

```
public class prog0505
{
    public static void main(String[] args)
    {
            int new;
            new = "Viva el Peru";
            System.out.println("Mensaje: " + new);
    }
}
```

Compilación del programa:

```
C:\java2n100\cap5>javac prog0505.java          [Enter]
prog0505.java:5: not a statement
                int new;
                ^
prog0505.java:5: ';' expected
                int new;
                    ^
prog0505.java:6: <identifier> expected
                new = "Viva el Peru";
                    ^
prog0505.java:6: '(' or '[' expected
                new = "Viva el Peru";
                                    ^
prog0505.java:7: <identifier> expected
                System.out.println("Mensaje: " + new);
                                                     ^
prog0505.java:7: '(' or '[' expected
                System.out.println("Mensaje: " + new);
                                                      ^
6 errors
```

Note usted que en el proceso de compilación se muestran los errores.

5.2.3 Comentarios

Ayudarán a documentar de manera muy precisa sus programas.

Los tres estilos permitidos para insertar comentarios son:

1. Comentarios en línea

```
a// Este es un comentario
```

2. Comentarios en una o más líneas

```
/* Este es un comentario
en varias líneas */
```

3. Comentarios de documentación

```
/*
 *  Esta clase implementa un mantenimiento de tabla
 *  @autor Gustavo Coronel
 *  @version 1.0
 */
```

5.2.4 Punto y coma; bloques y espacios en blanco

En Java una sentencia es una o más líneas de código finalizadas con punto y coma (;). Por ejemplo:

```
importe = a + b + c + d + e + f + g;
```

Es igual que:

```
importe = a + b + c +
      d + e + f + g;
```

Un bloque (a veces llamado «declaración compuesta») es un grupo de sentencias delimitadas por llaves { }.

El siguiente es un ejemplo de un bloque simple:

```
{   // Inicio del bloque

    c = a + b;
    a = b;
    b = c

}   // Finalización del bloque
```

La definición de una clase está contenida dentro de un bloque, tal como se ilustra a continuación:

```
public class MiClase
{   Inicio del bloque

    private int cod_prod;
    private int pre_prod;
    . . .

}   Finalización del bloque
```

Un bloque de sentencias puede estar contenido dentro de otro bloque, tal como se observa en el siguiente ejemplo:

```
while ( j < limite ) { // Inicio de bloque externo

    total = total + j;

    if ( total == maximo ) { // Inicio del bloque anidado
            acumulado = acumulado + total;
            total = 0;
    } // Fin del bloque anidado

    j = j + 1;

} // Fin del bloque externo
```

También puede usar espacios en blanco entre los elementos del código fuente:

```
{ int        x;x=15*20;}
```

Es lo mismo que;

```
{
    int x;
    x = 15 * 20;
}
```

5.2.5 Tipos de datos primitivos

Java define ocho tipos de datos primitivos que pueden ser considerados en cuatro categorías (ver Fig. 5.1).

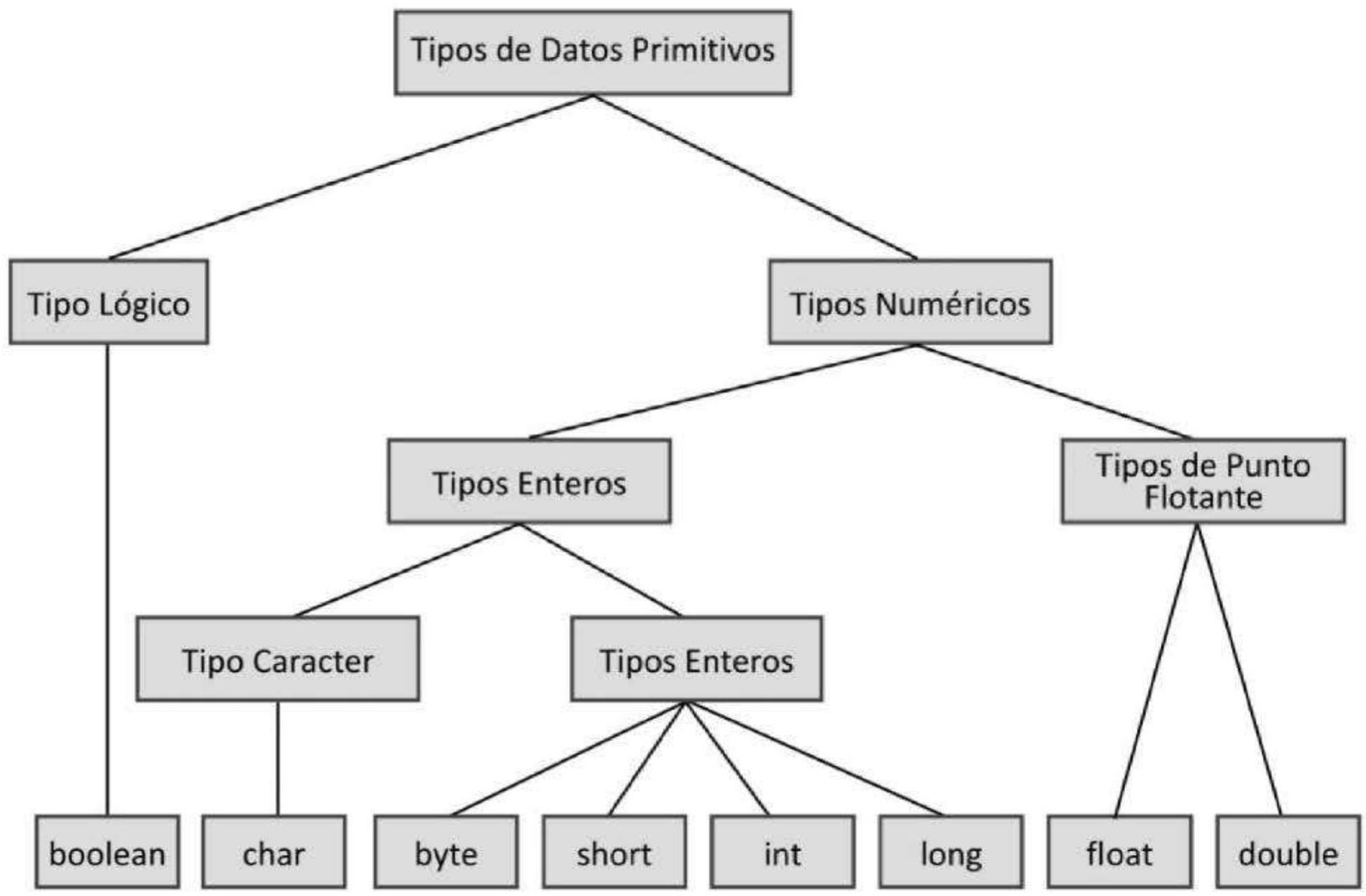

Fig. 5.1 Tipos de datos primitivos de Java

El tamaño efectivo de estos tipos de datos se observa en la siguiente tabla:

Tabla 5.3 *Tamaño efectivo en bits de los tipos de datos primitivos*

Tipo	Bits	Tipo	Bits
boolean	1	**char**	16
byte	8	**short**	16
int	32	**long**	64
float	32	**double**	64

Señalamos un resumen de los tipos de datos primitivos donde se aprecia sus clases envolventes (ver Tabla 5.4).

Las clases envolventes de tipos primitivos describen el tipo de dato correspondiente; se trata de una clase estática. Por lo tanto no necesitamos instanciarla para poder utilizar sus métodos.

Tabla 5.4 *Resumen de tipos de datos primitivos*

Tipo de dato	Ancho en bits	Valor mínimo, Valor máximo	Clase envolvente
boolean	No aplicable	true, false (valores discretos)	Boolean
byte	8	-2^7, 2^7-1	Byte
short	16	-2^{15}, $2^{15}-1$	Short
char	16	0x0, 0xffff	Character
int	32	-2^{31}, $2^{31}-1$	Integer
long	64	-2^{63}, $2^{63}-1$	Long
float	32	±1.40129846432481707e-45f, 3.40282347663852886 0e+38f	Float
double	64	\'14.9406564584124654 4e-324, \'11.7976931348623157 0e+308	Double

a. Tipo boolean

Tipo lógico que representa dos valores: **true** y **false**. Una variable de este tipo solo puede tomar valor **true** o **false**.

- **Ejemplo 6**

  En el siguiente ejemplo se puede apreciar el uso del tipo **boolean**.

prog0506.java

```
ppublic class prog0506
{
    public static void main(String[] args)
    {
            boolean pase = true;
            System.out.println("pase = " + Boolean.toString(pase));
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0506.java        [Enter]
C:\java2n100\cap5>java prog0506              [Enter]
pase = true
```

Observamos que se está utilizando el método **toString** de la clase **Boolean** para convertir el valor lógico a una cadena.

b. Tipo char

Representa un caracter simple. Para representarlo debe encerrarlo entre apóstrofes, tal como se aprecia en los siguientes ejemplos:

'a'	La letra **a**
'\t'	Un **tab**
'\u????'	Un caracter Unicote **????**, es reemplazado con exactamente cuatro dígitos hexadecimales, por ejemplo **'\u03A6'** es la letra Griega **phi (ϕ)**

El tipo String no es un tipo primitivo, es una clase que se utiliza para representar una secuencia de caracteres.

- **Ejemplo 7**

Se observa el uso del tipo **char**.

prog0507.java

```
public class prog0507
{
    public static void main(String[] args)
    {
        char c1 = 'A', c2 = '\t', c3 = '\u03A6';
        String msg;
        msg = Character.toString(c1) +
              Character.toString(c2) +
              Character.toString(c3) ;
          System.out.println(msg);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0507.java          [Enter]

C:\java2n100\cap5>java prog0507                [Enter]
A       ?
```

c. Tipos byte, short, int y long

Tenemos cuatro tipos de datos enteros en Java. Podemos representar tipos enteros en formato decimal, octa y hexadecimal; tal como se aprecia en el siguiente cuadro:

15	Es un número entero en formato decimal. Su valor es 15.
077	Es un número entero en formato octal. Su valor decimal es 63.
0xBAAC	Es un número entero en formato hexadecimal. Su valor decimal es 47788.

Todos los tipos enteros en Java representan números con signos.

Todos los literales enteros son de tipo **int**, a menos que explícitamente se le ponga el sufijo L que indica que se trata de un valor **long**. En el siguiente cuadro se pueden apreciar algunos ejemplos:

15L	Es un número entero en formato decimal tipo long. Su valor es 15.
077L	Es un número entero en formato octal tipo long. Su valor decimal es 63.
0xBAACL	Es un número entero en formato hexadecimal tipo long. Su valor decimal es 47788.

A continuación, señalamos las características fundamentales de los tipos de datos enteros: tamaño y rango (ver Tabla 5.5).

Tabla 5.5 *Tipos de datos enteros, tamaño y rango*

Tipo	Tamaño	Rango
byte	8 bits	-2^7 hasta 2^7-1
short	16 bits	-2^{15} hasta $2^{15}-1$
int	32 bits	-2^{31} hasta $2^{31}-1$
long	64 bits	-2^{63} hasta $2^{63}-1$

- **Ejemplo 8**

Se observa el uso de tipos de datos enteros.

prog0508.java

```
ppublic class prog0508
{
    public static void main(String[] args)
    {
            byte  n1 = 15;
            short n2 = 10000;
            int   n3 = 100000;
            System.out.println("n1 = " + Byte.toString(n1));
            System.out.println("n2 = " + Short.toString(n2));
            System.out.println("n3 = " + Integer.toString(n3));
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0508.java          [Enter]

C:\java2n100\cap5>java prog0508                [Enter]
n1 = 15
n2 = 10000
n3 = 100000
```

d. Tipos flot y double

Tenemos 2 tipos de datos de punto flotante en Java. Un literal numérico es de tipo punto flotante si incluye un punto decimal, una parte exponencial (letra e o E) o esta seguida de las letras F o f (**float**), o D o d (**double**). Los valores de coma flotante son por defecto de tipo double. El siguiente cuadro muestra algunos ejemplos de números de coma flotante.

3.15	Un número de coma flotante simple, tipo double.
6.12E23	Un número de coma flotante grande.
2.718F	Un número de coma flotante simple.
123.4E+306D	Un número de coma flotante grande con D redundante.

En la Tabla 5.6 tenemos el tamaño en bits de cada tipo de dato de coma flotante.

Tabla 5.6 *Tipos de coma flotante*

Tipo	Tamaño
float	32 bits
double	64 bits

- **Ejemplo 9**

En el siguiente ejemplo podemos apreciar el uso de tipos de datos de coma flotante.

prog0509.java

```
public class prog0509
{
    public static void main(String[] args)
    {
            float  n1 = 45.678f;
            double n2 = 1689.3483467;
            System.out.printf("n1 = %1$10.2f\n", n1 );
            System.out.printf("n2 = %1$.6e", n2 );
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0509.java        [Enter]

C:\java2n100\cap5>java prog0509              [Enter]
n1 =      45,68
n2 = 1.689348e+03
```

Aquí apreciamos el uso de la función **printf** para poder imprimir datos con formato.

5.2.6 Variables y operadores

a. Variables

Para declarar variables debemos utilizar la siguiente sintaxis:

```
tipo nombre_variable [ = valor_inicial ], . . .  ;
```

Observamos en el siguiente ejemplo que es posible asignar en la misma declaración un valor inicial.

- **Ejemplo 10**

prog0510.java

```
public class prog0510
{
    public static void main(String[] args)
    {
            int a = 5, b = 10;
            int c;
            c = a * b;
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            System.out.println("c = " + c);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0509.java          [Enter]

C:\java2n100\cap5>java prog0509                [Enter]
n1 =      45,68
n2 = 1.689348e+03
```

b. Operadores aritméticos

1. Operadores unarios

Operador	Descripción	Ejemplo
+	Identificación de signo	int n = -5; n = +n; // n toma valor -5
-	Negación de signo	int n = -5; n = -n; // n toma valor +5

2. Operadores binarios

Operador	Descripción	Ejemplo
+	Suma	int n1 = 5, n2 = 10, n3; n3 = n1 + n2; // n3 toma valor 15
-	Resta	int n1 = 5, n2 = 10, n3; n3 = n1 + n2; // n3 toma valor -5
*	Multiplicación	int n1 = 5, n2 = 10, n3; n3 = n1 * n2; // n3 toma valor 50

/	División	int n1 = 14, n2 = 4, n3;
		float n4;
		n3 = n1 / n2; // n3 toma valor 3
		n4 = (float)n1 / n2; // n4 toma valor 3.5

%	Resto	int n1 = 14, n2 = 4, n3; n3 = n1 % n2; // n3 toma valor 2

El comportamiento del operador de división depende de los operandos. Si ambos son enteros la división será entera, si uno de los operandos es de punto flotante, entonces se tratará de una división real.

- **Ejemplo 11**

A continuación ilustramos el uso de operadores aritméticos.

prog0511.java

```
public class prog0511
{
     public static void main(String[] args)
     {
    int a, b, r1, r2, r3;
    float r4;
    a = Integer.parseInt( args[0] );
    b = Integer.parseInt( args[1] );
    r1 = a * b; /* Multiplicación */
    r2 = a / b; /* División Entera */
    r3 = a % b; /* Resto */
    r4 = (float)a / b; /* División Real */
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("Multiplicacion = " + r1);
    System.out.println("Division Entera = " + r2);
    System.out.println("Resto = " + r3);
    System.out.println("Division Real = " + r4);
     }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0511.java          [Enter]

C:\java2n100\cap5>java prog0511 20 6           [Enter]
a = 20
b = 6
Multiplicacion = 120
Division Entera = 3
Resto = 2
Division Real = 3.3333333
```

3. Operadores de asignación

El siguiente cuadro muestra los diferentes operadores de asignación con los que cuenta Java.

Operador	Descripción	Ejemplo

=	Asignación simple	int a = 5;
		int b = a + 6;
		int c;
		c = a * b;

+=	Suma y asignación	int a = 5, b = 10; b += a; // Equivalente a: b = b + a
-=	Resta y asignación	int a = 5, b = 10; b -= a; // Equivalente a: b = b - a
*=	Multiplicación y asignación	int a = 5, b = 10; b *= a; // Equivalente a: b = b * a
/=	División y asignación	int a = 5, b = 10; b /= a; // Equivalente a: b = b / a
%=	Resto y asignación	int a = 5, b = 10; b %= a; // Equivalente a: b = b % a

- **Ejemplo 12**

A continuación señalamos el uso del operador +=. Observe que el valor de **b** se incrementa en el valor de **a**.

prog0512

```
public class prog0512
{
    public static void main(String[] args)
    {
            int a, b;
            a = Integer.parseInt( args[0] );
            b = Integer.parseInt( args[1] );
```

```
System.out.println("a = " + a);
            System.out.println("b = " + b);
            b += a;
            System.out.println("Nuevo valor de b =  " + b);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0512.java        [Enter]

C:\java2n100\cap5>java prog0512 10 15        [Enter]
a = 10
b = 15
Nuevo valor de b =  25
```

c. Expresión lógica

Una expresión lógica puede tomar solo valores de tipo boolean: **true** o **false**.

Cuando las expresiones lógicas son utilizadas como condicionales en las instrucciones de control, permiten controlar el flujo del programa durante su ejecución. Se construyen con operadores de comparación y lógicos.

1. Operadores de comparación

Permiten construir expresiones lógicas simples, ya que comparan dos valores dando un resultado lógico. La siguiente tabla muestra los diferentes operadores de comparación con los que cuenta Java:

Operador	Descripción	Ejemplo
==	Igualdad	boolean a; a = 6 == 7; // Toma valor false
!=	Diferente	boolean a; a = '6' != 6; // Toma valor true
>	Mayor	boolean a; a = 6 > 7; // Toma valor false
>=	Mayor que	boolean a; a = 'A' >= 'a'; // Toma valor true
<	Menor	boolean a; a = 20 < 50; // Toma valor true
<=	Menor que	boolean a; a = 100 <= 100; // Toma valor true

- **Ejemplo 13**

  Se indica el uso del operador ==. El programa recibe como dato de entrada un número entero y nos informa si es par o impar.

prog0513.java

```
public class prog0513
{
    public static void main(String[] args)
    {
        int n;
        boolean esPar;
        n = Integer.parseInt( args[0] );
        esPar = n % 2 == 0;
        System.out.println("n = " + n);
        System.out.println("Es Par = " + Boolean.toString(esPar));
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0513.java          [Enter]

C:\java2n100\cap5>java prog0513 15             [Enter]
n = 15
Es Par = false
```

2. Operadores lógicos

Permiten crear expresiones lógicas complejas, estos incluyen un operador unario ! (complemento lógico) y tres operadores binarios: & (Y lógico), | (O lógico), ^ (O exclusivo).

Los operadores lógicos se aplican a operandos lógicos y su resultado es otro valor lógico. Se definen en la siguiente tabla:

Operador	Descripción	Ejemplo
!	Complemento lógico	boolean a; a = ! (6 == 7); // Toma valor true
&	Y Lógico	boolean a; a = (6 == 6) & (8<15); // Toma valor true
|	O Lógico	boolean a; a = (6 == 6) | (8<15); // Toma valor true
^	O Exclusivo	boolean a; a = (6 == 6) ^ (8<15); // Toma valor false

Asumiendo que **x** e **y** representan dos expresiones lógicas, la tabla siguiente muestra un resumen del funcionamiento de los operadores lógicos:

x	y	!x	x & y	x | y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

- Ejemplo 14

  Evalúamos si un número entero es par y a la vez es múltiplo de 5. El número a evaluar se ingresa en la línea de comandos.

prog0514.java

```
public class prog0514
{
    public static void main(String[] args)
    {
            int n;
            boolean ok;
            n = Integer.parseInt( args[0] );
            ok = (n % 2 == 0) & (n % 5 == 0);
            System.out.println("n = " + n);
            System.out.println("Condicion = " + Boolean.toString(ok));
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0514.java          [Enter]

C:\java2n100\cap5>java prog0514 15             [Enter]
n = 15
Condicion = false

C:\java2n100\cap5>java prog0514 20             [Enter]
n = 20
Condicion = true
```

3. Operadores de asignación con operadores lógicos

También contamos con operadores de asignación compuestos por operadores lógicos, tal como se ilustra a continuación:

Operador	Descripción	Ejemplo
&=	Lógica "Y" y asignación	boolean a = true, b = false; a &= b; // Equivale a: a = a & b
|=	Lógica "O" y asignación	boolean a = true, b= false; a |= b; // Equivalente a: a = a | b
^=	Lógica "O" exclusiva y asignación	boolean a = true, b = false; a ^= b;; // Equivalente a: a = a ^ b

- **Ejemplo 15**

Verificamos si un número entero es múltiplo de 3 y 5 a la vez.

prog0515.java

```
public class prog0515
{
    public static void main(String[] args)
    {
            int n;
            boolean ok;
            n = Integer.parseInt(args[0]);
            ok = (n % 3 == 0);
            ok &= (n % 5 == 0);
            System.out.println("n = " + n);
            System.out.println("Condicion = " + Boolean.toString(ok));
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0515.java          [Enter]

C:\java2n100\cap5>java prog0515 9              [Enter]
n = 9
Condicion = false

C:\java2n100\cap5>java prog0515 15             [Enter]
n = 15
Condicion = true
```

4. Operadores condicionales

Los operadores **&&** y **||** son similares a sus equivalentes operadores lógicos **&** y **|**, como se observa en la tabla siguiente:

Operador	Descripción	Ejemplo
&&	Lógica "Y"	boolean ok; ok = (5 > 8) && (10 > 5); // Toma valor false
||	Lógica "O"	boolean ok; ok = (5 > 8) || (10 > 5); // Toma valor true

Las expresiones que utilizan estos operadores tienen la ventaja de finalizar en el momento en que se determine el valor de la expresión. Consideremos el siguiente ejemplo:

```
String cadena;
. . .
if( (cadena != null) && (cadena.length() > 0) ) {
    // Instrucciones
}
```

La expresión lógica de la instrucción **if()** es correcta y totalmente segura, ya que cuando la primera subexpresión es falsa, la segunda no es evaluada porque toda la expresión es falsa. De forma similar ocurre con el operador **||**.

5. Operadores a nivel de bits

A continuación, se muestra los operadores que podemos aplicar a nivel de bits.

Operador	Notación	Efecto en cada bit
Complemento	~A	Invierte el valor del bit: 0 a 1, 1 a 0.
AND	A & B	1 si ambos bits son 1; 0 en otros casos.
OR	A | B	1 si cualquiera o ambos bits es 1; 0 en otros casos.
XOR	^	1 si y solo si uno de los bits es 1; 0 en otros casos.

Ilustramos el funcionamiento de los operadores a nivel de bits (ver Fig. 5.2).

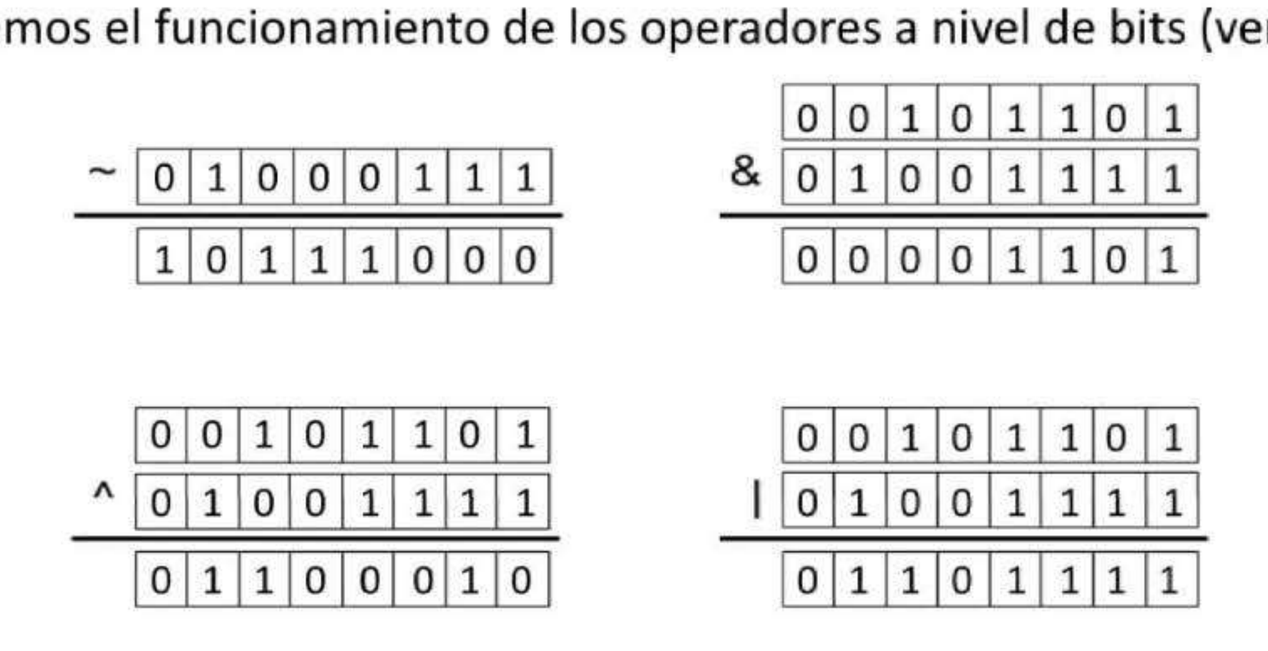

Fig. 5.2 Funcionamiento de los operadores a nivel de bits

- **Ejemplo 16**

  Observamos el uso de los operadores a nivel de bits. El programa recibe dos números enteros y luego se utilizan los operadores **&** y **^** para operarlos a nivel de bits.

prog0516.java

```
public class prog0516
{
    public static void main(String[] args)
    {
            int n1, n2, n3, n4;
            n1 = Integer.parseInt(args[0]);
            n2 = Integer.parseInt(args[1]);
            n3 = n1 & n2;
            n4 = n1 ^ n2;
            System.out.println("n1 = " + n1);
            System.out.println("n2 = " + n2);
            System.out.println("n3 = " + n3);
            System.out.println("n4 = " + n4);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0516.java          [Enter]

C:\java2n100\cap5>java prog0516 15 37          [Enter]
n1 = 15
n2 = 37
n3 = 5
n4 = 42
```

6. Operadores de desplazamiento

Actúan a nivel de bits. La siguiente tabla detalla su funcionamiento.

Operador	Notación	Efecto en cada Bit
<<	N << p	Desplaza los bits de N, p posiciones a la izquierda
>>	N >> p	Desplaza los bits de N, p posiciones a la derecha, incluye el signo
>>>	N >>> p	Desplaza los bits de N, p posiciones de la derecha, incluye el signo

- **Ejemplo 17**

  Señalamos el uso del operador >>. Se puede comprobar que los bits de **n1** se desplazan 1 posición a la derecha.

prog0517.java

```
public class prog0517
{
     public static void main(String[] args)
     {
          int n1 = 160;            // n1 = 10100000
          int n2;
          n2 = n1 >> 1;    // n2 = 01010000
          System.out.println("n1 = " + n1);
          System.out.println("n2 = " + n2);
     }
}
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0517.java        [Enter]

C:\java2n100\cap5>java prog0517              [Enter]
n1 = 160
n2 = 80
```

7. Operador condicional

Permite definir una condición y en función a su resultado obtener un valor.

- **Sintaxis**

```
condición ? expresión1 : expresión2
```

Si la condición es **true**, obtenemos el valor de la **expresión1**; en caso contrario obtenemos el valor de **expresión2**. También es posible anidar este operador.

- **Ejemplo 18**

  En el presente ejemplo se hace uso del operador condicional para obtener el mayor de 2 números.

prog0518.java

```
public class prog0518
{
     public static void main(String[] args)
     {
          int n1, n2, mayor;
          n1 = Integer.parseInt(args[0]);
          n2 = Integer.parseInt(args[1]);
             mayor = (n1>n2)?n1:n2;
```

```
            System.out.println("n1 = " + n1);
            System.out.println("n2 = " + n2);
            System.out.println("mayor = " + mayor);
        }
    }
```

Compilación y ejecución:

```
C:\java2n100\cap5>javac prog0518.java          [Enter]

C:\java2n100\cap5>java prog0518 20 25          [Enter]
n1 = 20
n2 = 25
mayor = 25

C:\java2n100\cap5>java prog0518 45 25          [Enter]
n1 = 45
n2 = 25
mayor = 45
```

8. Concatenación de cadenas con +

El operador **+** permite concatenar objetos **String**, produciendo uno nuevo, tal como se ilustra en el siguiente ejemplo:

```
String equipo = "Alianza";
String condicion = "Campeon";
String frase = equipo + " " + condicion;
```

El resultado de la última línea es:

```
Alianza Campeón
```

Si uno de los operandos del operador **+** es un **String**, entonces el otro operando es convertido automáticamente a String. Todos los objetos pueden ser convertidos a un objeto String aunque el resultado puede ser algo críptico.

5.2.7 Transformación de tipos: casting

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de **int** a **double**, o de *float* a *long*.

La conversión entre tipos primitivos es más sencilla. En Java se realizan de modo automático conversiones implícitas de un tipo a otro de más precisión; por ejemplo de int a long, de float a double, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación, donde la variable a la izquierda del operador de asignación es de un tipo distinto (más amplio) que el resultado de evaluar la expresión a la derecha del operador de asignación. Las conversiones de un tipo de mayor a otro de menor precisión requieren una

instrucción explícita, pues son conversiones inseguras que pueden dar lugar a errores. Por ejemplo, para pasar a **short** un número almacenado como **int**, hay que estar seguro de que puede ser representado con el número de cifras binarias de **short**. A estas conversiones explícitas de tipo se les llama **cast**. El cast se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo:

```
Slong r;
r = (long) (a/(b+c));
```

La expresión (*long*) hace que el resultado de la expresión a la derecha sea convertido a un valor de tipo *long*.

5.3 IMPRESIÓN EN LA CONSOLA

Para poder imprimir en la consola debemos utilizar **System.out** que pertenece a la clase **java.io.PrintStream**, aquí tenemos implementados los siguientes métodos:

- **print()**
- **println()**
- **printf()**

5.3.1 MÉTODO PRINT()

- **Sintaxis**

```
System.out.print( dato );
```

Se trata de un método sobrecargado, es decir que existe una implementación para cada tipo de dato. Simplemente como dato ponemos una expresión de cualquier tipo y listo, el lenguaje se encarga de utilizar el método correcto, veamos algunos ejemplos.

```
int nota = 15;
double igv = 0.19;
System.out.print(nota);
System.out.print("\t");
System.out.print(igv);
```

Aquí estamos usando **print** para imprimir tres datos de tipos diferentes: primero un **int**; luego un **String** y finalmente un **double**, el resultado sería el siguiente:

```
15          0.19 5          0.19
```

La característica de este método es que no realiza un cambio de línea.

5.3.2 Método println()

Sintaxis

```
System.out.println( dato );
```

Al igual que el método **print**, este también se encuentra sobrecargado. Existe una implementación para cada tipo de dato, donde dato puede ser una expresión de cualquier tipo; el lenguaje se encarga de utilizar el método correcto, veamos algunos ejemplos.

```
int nota = 15;
double igv = 0.19;
String msg = "El verdadero discípulo es el que supera al maestro";
System.out.println(nota);
System.out.println(msg);
System.out.println(igv);
```

En este ejemplo estamos usando **println** para imprimir tres datos de tipos diferentes. Primero un **int**; luego un **String** y finalmente un **double**, el resultado sería el siguiente:

```
15
El verdadero discípulo es el que supera al maestro
0.19
```

La característica de este método es que realiza un cambio de línea.

5.3.3 Método printf()

Sintaxis

```
System.out.printf( cadena_formato, Lista_parametros );
```

Este método puede tener varios parámetros, pero el primero de ellos es una cadena con formato, y los demás dependen de los parámetros que tiene la cadena con formato.

La sintaxis general para indicar los formatos es la siguiente:

```
%[nro_parametro$][flags][ancho][.precisión]sinbolo_conversión
```

Las opciones que están entre corchetes son opcionales y se utilizan, según el tipo de dato y necesidades de impresión que se tengan.

La siguiente tabla muestra los diferentes caracteres que pueden ser usados como **flag** y dependen del tipo de dato que se quiere imprimir.

Flag	Descripción
'-'	El resultado se ajustará a la izquierda.
'#'	El resultado debe usar una forma de conversión alterna.
'+'	El resultado incluirá el signo.
' '	El resultado incluirá un espacio para números positivos.
'0'	El resultado será rellenado con ceros.
','	El resultado incluirá separadores.
'('	Los números negativos serán encerrados entre paréntesis.

Veamos el siguiente ejemplo:

```
float precio = 18.567f;
System.out.printf("Precio: %1$f   %1$010.2f", precio);
```

Se está imprimiendo el precio dos veces; en la segunda se está indicando que se rellene con ceros, un ancho de 10 y decimales, el resultado es el siguiente:

```
Precio: 18,566999        0000018,57
```

En la siguiente tabla se muestra los diferentes símbolos que podemos utilizar para dar formatos.

Símbolo	Categoría	Significado
b B	General	Si el argumento es null imprime **false**. Si el argumento es boolean imprime **false** o **true**. para cualquier otro caso imprime.
h H	General	Si el argumento es null imprime null. Cualquier otro caso el valor hexadecimal del argumento.
s S	General	Si el argumento es null imprime null. Cualquier otro caso el argumento es convertido a String.
c C	Caracter	El argumento se trata como número entero y se muestra el caracter cuyo código ASCII se corresponde con el valor.
d	Entero	El argumento se trata como número entero y se imprime su valor decimal.
o	Entero	El argumento se trata como un número entero y se imprime su valor en octal.
x X	Entero	El argumento se trata como un número entero y se imprime su valor en hexadecimal.

e E	Punto flotante	El argumento se trata como un número de tipo double y se imprime en notación científica.
f	Punto flotante	El argumento se trata como un número de tipo double y se imprime como un decimal.
g G	Punto flotante	El argumento se imprime como un decimal o notación científica, dependiendo de la precisión y el valor después del redondeo.
a A	Punto flotante	El argumento se ajusta a un formato de punto flotante hexadecimal.
t T	Fecha/Hora	Este prefijo se utiliza para conversiones de fecha y hora.
%	Porcentaje	Representa el símbolo del porcentaje
n	Letra n	Representa un salto de línea.

Veamos el siguiente ejemplo:

```
float precio = 18.567f;
System.out.printf("Precio: %1$e   %1$.2f", precio);
```

Se está imprimiendo el mismo número dos veces; la primera vez se imprime en notación científica y la segunda vez se especifica con dos decimales, el resultado es el siguiente:

```
Precio: 1.856700e+01   18,57
```

El siguiente ejemplo ilustra el uso de formatos de fecha:

```
import java.util.Date;
public class test
{
    public static void main(String[] args)
    {
            Date fecha = new Date();
            System.out.printf("Fecha de hoy: %1$tY-%1$tm-%1$td", fecha);
    }
}
```

El resultado es el siguiente:

```
Fecha de hoy: 2015-12-18
```

Note usted que se está imprimiendo el mismo argumento tres veces; la primera vez se imprime el año, la segunda vez el mes, y finalmente el día.

Instrucciones de control

CAPÍTULO 6

Capacidad: En este capítulo se desarrollarán todas las instrucciones de control para un adecuado manejo del lenguaje de programación.

Las instrucciones de control permiten realizar la verificación de ciertas condiciones y en función a su resultado seleccionar las instrucciones a ejecutar o repetir su ejecución las veces que sean necesarias.

6.1 INTRODUCCIÓN

Las instrucciones de control permiten bifurcar el flujo del programa y así ejecutar unas partes u otras del código según ciertas condiciones. Java dispone de todas las instrucciones clásicas.

En todos los casos, las instrucciones de control contienen una expresión cuya evaluación como verdadero o falso determinará el flujo a seguir dentro del programa. Estas expresiones pueden ser una variable, una función, una constante o cualquier combinación de estas con los operadores respectivos.

Estas instrucciones están divididas en dos grupos:

- Selectivas
- Repetitivas

6.2 INSTRUCCIONES SELECTIVAS

6.2.1 SELECTIVA SIMPLE: IF

La instrucción **if** es la estructura de control más básica de todas. Es su forma más simple.

Sintaxis:

```
if (condición)
    instrucción;
```

Si **condición** se evalúa como verdadera, se ejecuta **instrucción** y luego se sigue ejecutando el resto del **programa**. Si se evalúa como falsa, no se ejecuta instrucción y continúa con el resto del programa.

Si lo que se quiere es ejecutar varias instrucciones, estas deben ser consideradas como un bloque, tal como se indica a continuación:

```
if (condición)
{
        instrucción;
}
```

- **Ejemplo 1**
  El siguiente programa calcula el importe de la venta de un determinado artículo, los datos a ingresar son: Precio, descuento y cantidad.

  El descuento solo se aplica si la cantidad es mayor o igual a 6 unidades.

prog0601.java

```
public class prog0601
{
    public static void main(String[] args)
    {
            int precio, dcto, cant, total;
            precio = Integer.parseInt(args[0]);
            dcto   = Integer.parseInt(args[1]);
            cant   = Integer.parseInt(args[2]);
            if (cant < 6) dcto = 0;
            total = (precio - dcto) * cant;
            System.out.println("Precio = " + precio);
            System.out.println("Dcto = " + dcto);
            System.out.println("Cant = " + cant);
            System.out.println("Total = " + total);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0601.java                    [Enter]

C:\java2n100\cap6>java prog0601 15 3 10                  [Enter]
Precio = 15
Dcto = 3
Cant = 10
Total = 120

C:\java2n100\cap6>java prog0601 15 3 3                   [Enter]
Precio = 15
Dcto = 0
Cant = 3
Total = 45
```

6.2.2 Selectiva doble: if - else

A una instrucción **if** se le puede añadir instrucciones que se ejecuten cuando la condición es falsa mediante la cláusula **else**.

Sintaxis

```
if (condición)
    instrucción1;
else
    instrucción2;
```

Si **condición** se evalúa como verdadera se ejecuta **instrucción1**; en caso contrario se ejecuta **instrucción2**.

Es importante recordar que si lo que se quiere es ejecutar varias instrucciones, estas deben ser consideradas dentro de un bloque, tal como se ilustra a continuación:

```
if (condición)
{
    instrucción1;
} else {
    instrucción2;
}
```

- **Ejemplo 2**
  El siguiente programa permite determinar si un número entero es par o impar.

  Note que se está verificando el tamaño del argumento **args** (que es un arreglo), de ser cero (0) significa que no ha ingresado ningún número y muestra un mensaje; tal como se comprueba en la ejecución.

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0601.java                    [Enter]

C:\java2n100\cap6>java prog0601 15 3 10                  [Enter]
Precio = 15
Dcto = 3
Cant = 10
Total = 120

C:\java2n100\cap6>java prog0601 15 3 3                   [Enter]
Precio = 15
Dcto = 0
Cant = 3
Total = 45
```

6.2.2 Selectiva doble: if - else

A una instrucción **if** se le puede añadir instrucciones que se ejecuten cuando la condición es falsa mediante la cláusula **else**.

Sintaxis

```
if (condición)
    instrucción1;
else
    instrucción2;
```

Si **condición** se evalúa como verdadera se ejecuta **instrucción1**; en caso contrario se ejecuta **instrucción2**.

Es importante recordar que si lo que se quiere es ejecutar varias instrucciones, estas deben ser consideradas dentro de un bloque, tal como se ilustra a continuación:

```
if (condición)
{
    instrucción1;
} else {
    instrucción2;
}
```

- **Ejemplo 2**
  El siguiente programa permite determinar si un número entero es par o impar.

  Note que se está verificando el tamaño del argumento **args** (que es un arreglo), de ser cero (0) significa que no ha ingresado ningún número y muestra un mensaje; tal como se comprueba en la ejecución.

prog0602.java

```
public class prog0602
{
        public static void main(String[] args)
        {
                int n;
                String msg = "";
                if (args.length == 0)
                {
                        System.out.println( "Debe ingresar un numero
entero" );
                        return;
                }
                n = Integer.parseInt(args[0]);
                if (n % 2 == 0)
                        msg = n + " es par";
                else
                        msg = n + " es el impar";
                System.out.println("n = " + n);
                System.out.println(msg);
        }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0602.java          [Enter]

C:\java2n100\cap6>java prog0602                [Enter]
Debe ingresar un numero entero

C:\java2n100\cap6>java prog0602 15             [Enter]
n = 15
15 es el impar
```

6.2.3 Anidamiento de instrucciones if

Las instrucciones **if** se pueden anidar. Es recomendable demarcar claramente los bloques que comprende cada sección de la instrucción **if**, para una mayor claridad de la lógica que se está programando.

Por ejemplo, en el siguiente *script*:

```
if (condición1)
    if (condición2)
            instrucción1;
    else
            instrucción2;
else
    instrucción3;
```

- instrucción1, se ejecuta cuando condición1 y 2 son verdaderas.
- instrucción2, se ejecuta cuando condición1 es verdadera y condición2 es falsa.
- instrucción3, se ejecuta cuando condición1 es falsa; no se toma en cuenta el valor de condición2.

El mismo *script* puede ser escrito de la siguiente manera:

```
if (condición1)
{
    if (condición2)
            instrucción1;
    else
            instrucción2;
} else {
    instrucción3;
}
```

Consiguiendo mayor claridad en la lógica del programa.

- **Ejemplo 3**
  El siguiente programa recibe dos números enteros y determina cuál de ellos es el mayor, o si son iguales.

prog0603.java

```
public class prog0603
{
    public static void main(String[] args)
    {
            int n1, n2;
            String msg = "";
            if (args.length < 2)
            {
                    System.out.println( "Debe ingresar dos numeros ente-
ros" );
                    return;
            }
            n1 = Integer.parseInt(args[0]);
            n2 = Integer.parseInt(args[1]);
            if (n1 > n2)
            {
                    msg = n1 + " es el mayor";
            } else {
                    if (n2 > n1)
                            msg = n2 + " es el mayor";
                    else
                            msg = "Ambos son iguales";
            }
            System.out.println("n1 = " + n1);
            System.out.println("n2 = " + n2);
            System.out.println(msg);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0603.java          [Enter]

C:\java2n100\cap6>java prog0603 15 10          [Enter]
n1 = 15
n2 = 10
15 es el mayor
C:\java2n100\cap6>java prog0603 15 20          [Enter]
n1 = 15
n2 = 20
20 es el mayor

C:\java2n100\cap6>java prog0603 15 15          [Enter]
n1 = 15
n2 = 15
Ambos son iguales
```

- **Ejemplo 4**

  El siguiente ejemplo califica la nota obtenida por un alumno.

prog0604.java

```
public class prog0604
{
    public static void main(String[] args)
    {
            String alumno, msg;
            int    nota;

            alumno = args[0];
            nota   = Integer.parseInt( args[1] );

            if(nota < 0)
                    msg = "No puede ser negativa";
            else if(nota <=5)
                    msg = "Pesima";
            else if(nota < 11)
                    msg = "Baja";
            else if(nota < 15)
                    msg = "Regular";
            else if(nota < 18)
                    msg = "Buena";
            else if(nota <= 20)
                    msg = "Excelente";
            else
                    msg = "No puede ser mayor que 20";

            System.out.println( "Alumno: " + alumno );
            System.out.println( "Nota: " + nota );
            System.out.println( "Calificacion: " + msg );
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0604.java          [Enter]

C:\java2n100\cap6>java prog0604 Ricardo 15     [Enter]
Alumno: Ricardo
Nota: 15
Calificacion: Buena

C:\java2n100\cap6>java prog0604 Ricardo 25     [Enter]
Alumno: Ricardo
Nota: 25
Calificacion: No puede ser mayor que 20

C:\java2n100\cap6>java prog0604 Ricardo -15    [Enter]
Alumno: Ricardo
Nota: -15
Calificacion: No puede ser negativa.
```

Note que el programa recibe dos parámetros; el primero es el nombre del alumno y el segundo su nota.

6.2.4 Selectiva múltiple: switch

La instrucción **switch** evalúa una expresión, y en función a su resultado selecciona las instrucciones que serán ejecutadas.

Sintaxis

```
switch (expresión) {
        case valor1:
                sentencia1;
                [break;]
        case valor2:
                sentencia2;
                [break;]
        case valor3:
                sentencia3;
                [break;]
        . . .
        . . .
        [ default:
                sentancia_por_defecto; ]
}
```

La expresión debe ser de tipo byte, char, short o int.

El resultado de evaluar la expresión se compara con cada uno de los casos: **valor1**, **valor2**, etc. Si coincide con alguno de ellos, ejecuta la instrucción asociada a ese caso. Finaliza cuando encuentra una sentencia **break** o llega al final de la instrucción.

La opción **default** es opcional y solo se ejecuta cuando no se encuentra ninguna coincidencia con algún valor.

- **Ejemplo 5**

  El siguiente programa simula una calculadora muy básica. Recibe dos números enteros y un operador, y retorna el resultado de la operación.

prog0605.java

```
public class prog0605
{
    public static void main(String[] args)
    {
        int n1 = Integer.parseInt( args[0] );
        int n2 = Integer.parseInt( args[1] );
        char ope = args[2].charAt(0); // Primer caracter del tercer
parámetro
        float rpta;

        switch(ope){
            case '+':
                rpta = n1 + n2;
                break;
            case '-':
                rpta = n1 - n2;
                break;
            case '*':
                rpta = n1 * n2;
                break;
            case '/':
                rpta = (float)n1 / n2;
                break;
            default:
                rpta = (float)0.0;
        }

        System.out.println( n1 + " " + ope + " " + n2 + " = " + rpta
);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0605.java          [Enter]

C:\java2n100\cap6>java prog0605 13 7 +         [Enter]
13 + 7 = 20.0

C:\java2n100\cap6>java prog0605 13 7 "*"       [Enter]
13 * 7 = 91.0

C:\java2n100\cap6>java prog0605 13 7 "/"       [Enter]
13 / 7 = 1.8571428
```

Observe que se pasan tres parámetros; cada uno de ellos se trata como cadena, por lo tanto también podemos encerrar los parámetros entre comillas.

En el caso del operador de multiplicación, si no lo encerramos entre comillas, Windows lo interpretará como si estuviéramos pasando la lista de archivos del directorio actual.

- **Ejemplo 6**
  El siguiente ejemplo recibe como parámetro el número de un mes y nos informa a qué trimestre pertenece.

prog0606.java

```
public class prog0606
{
    public static void main(String[] args)
    {
            int mes = Integer.parseInt( args[0] );
            String msg;

            switch(mes){
                    case 1: case 2: case 3:
                            msg = "Primer Trimestre";
                            break;
                    case 4: case 5: case 6:
                            msg = "Segundo Trimestre";
                            break;
                    case 7: case 8: case 9:
                            msg = "Tercer Trimestre";
                            break;
                    case 10: case 11: case 12:
                            msg = "Cuarto Trimestre";
                            break;
                    default:
                            msg = "Numero de mes fuera de rango";
            }

            System.out.println( "Numero de mes: " + mes );
            System.out.println( "Mensaje: " + msg );
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0606.java        [Enter]

C:\java2n100\cap6>java prog0606 5            [Enter]
Numero de mes: 5
Mensaje: Segundo Trimestre

C:\java2n100\cap6>java prog0606 10           [Enter]
```

```
Numero de mes: 10
Mensaje: Cuarto Trimestre

C:\java2n100\cap6>java prog0606 7            [Enter]
Numero de mes: 7
Mensaje: Tercer Trimestre
C:\java2n100\cap6>java prog0606 15           [Enter]
Numero de mes: 15
Mensaje: Numero de mes fuera de rango
```

Este ejemplo ilustra cómo ejecutar las mismas instrucciones para diferentes casos; por ejemplo para los meses 1, 2 y 3, el mensaje debe ser el mismo porque se trata del primer trimestre.

6.3 INSTRUCCIONES REPETITIVAS

6.3.1 BUCLE WHILE

Esta instrucción de bucle permite repetir la ejecución de un grupo de instrucciones mientras se cumpla cierta condición.

Sintaxis

```
while ( condición ){

    // Bloque de instrucciones

}
```

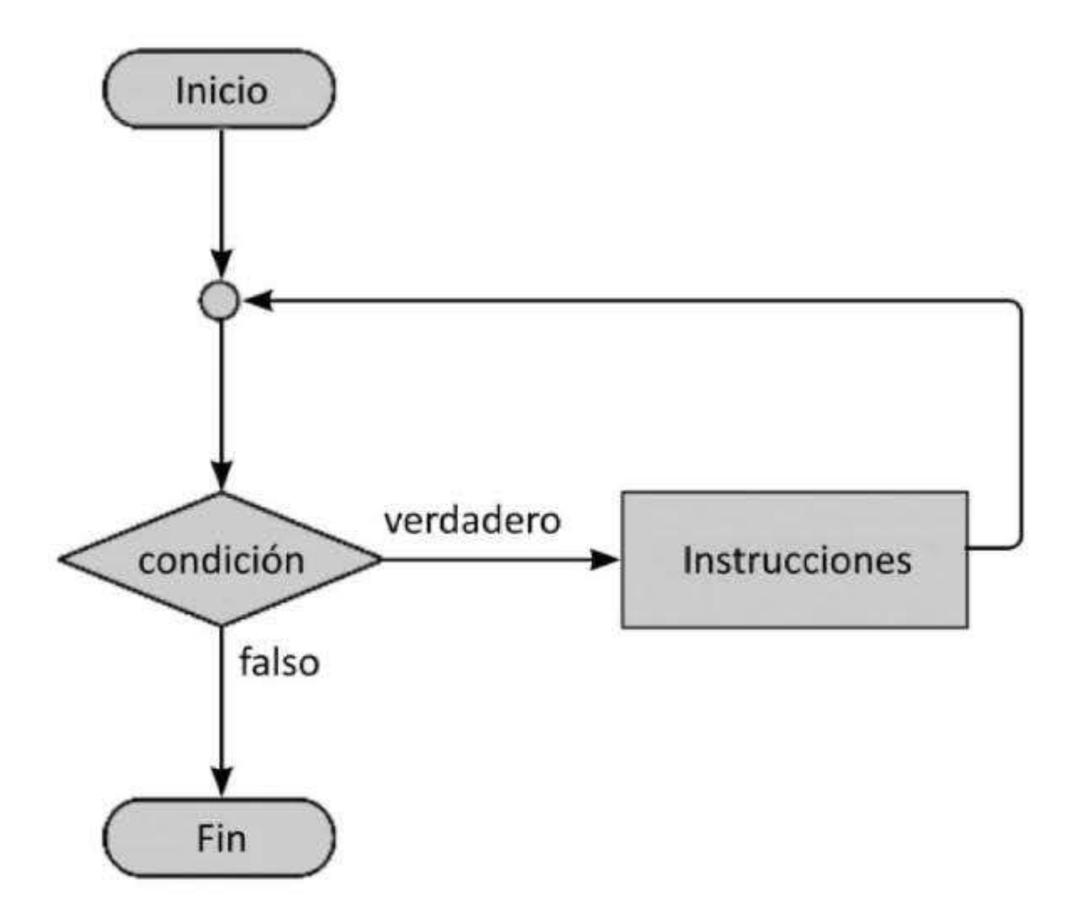

Fig. 6.1 Diagrama de flujo del bucle while

Observe que para iniciar el bucle, la primera vez que se evalúe la condición debe dar verdadero; en caso contrario no habrá ninguna iteración.

Otro aspecto importante es que debe existir alguna instrucción dentro del bucle que cambie la condición, como por ejemplo el valor de un contador; en caso contrario estaríamos en un bucle infinito.

- **Ejemplo 7**
  Se muestran los números pares entre m y n (m < n); los valores de m y n se deben ingresar como parámetros al programa.

prog0607.java

```
public class prog0607
{
    public static void main(String[] args)
    {
            int m = Integer.parseInt( args[0] );
            int n = Integer.parseInt( args[1] );

            while(m <= n){
                    if( m%2 == 0 )
                            System.out.println( m );
                    m++;
            }
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0607.java          [Enter]

C:\java2n100\cap6>java prog0607 15 21          [Enter]
16
18
20
```

Observe que el valor de m es modificado <m++;> dentro del bloque de instrucciones; de no ser así, la condición nunca llegaría a ser falsa, por lo tanto, el bucle no terminaría, sería un bucle infinito.

- **Ejemplo 8**
  El siguiente programa recibe como parámetros un mes (número) y un día del mismo mes, considerando solo el presente año (2025); luego calcula los días transcurridos desde el 1.º de enero hasta el día del mes ingresado y el día de la semana al que pertenece.

Debemos tener en cuenta que el 1.° de enero fue sábado.

prog0608.java

```
public class prog0608
{
    public static void main(String[] args)
    {
            int mes = Integer.parseInt( args[0] );
            int dia = Integer.parseInt( args[1] );
            int dt = 0; // Acumulador de días transcurridos
            int k = 0;  // Contador
            int ids = 0;    // Día de la semana
            String sds = “”; // Nombre del día de la semana

            while(++k <  mes){
                    switch(k){
                            case 1: case 3: case 5: case 7:
                            case 8: case 10: case 12:
                                    dt += 31;
                                    break;
                            case 4: case 6: case 9: case 11:
                                    dt += 30;
                                    break;
                            case 2:
                                    dt += 28;
                                    break;
                    }
            }

            ids = dt % 7; // Día de la semana
            switch(ids){
                    case 0: sds = “Viernes”; break;
                    case 1: sds = “Sábado”; break; // 1 de enero fue sábado
                    case 2: sds = “Domingo”; break;


                    case 3: sds = “Lunes”; break;
                    case 4: sds = “Martes”; break;
                    case 5: sds = “Miércoles”; break;
                    case 6: sds = “Jueves”; break;
            }

            System.out.println( “Días transcurridos: “ + dt );
            System.out.println( “Día de la semana: “ + sds );
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>date /T                    [Enter]
07/12/2005

C:\java2n100\cap6>javac prog0608.java        [Enter]

C:\java2n100\cap6>java prog0608 12 7         [Enter]
Días transcurridos: 334
Día de la semana: Miércoles
```

Observe que primero mostramos la fecha del sistema; luego ingresamos al programa el 7 de diciembre, primero el mes y luego el día, finalmente el programa nos muestra el resultado esperado.

- **Ejemplo 9**
  El siguiente programa permite visualizar la tabla de multiplicar de un número **n**. El valor de **n** se ingresa como parámetro en la línea de comandos.

prog0609.java

```
public class prog0609
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            int k = 0; // Contador
            int r;     // Variable auxiliar

            System.out.println( "Tabla de Multiplicar del numero: " + n
);
            while(++k <= 12){
                    r = n * k;
                    System.out.println( k + " * " + n + " = " + r );
            }
            System.out.println( "--- Fin ---" );

    }
}
```

La variable **k** se utiliza como contador. Observe que el incremento de esta variable se realiza en la misma expresión de condición, lo cual es totalmente válido.

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0609.java          [Enter]

C:\java2n100\cap6>java prog0609 8              [Enter]
Tabla de Multiplicar del número: 8
1 * 8 = 8
2 * 8 = 16
3 * 8 = 24
4 * 8 = 32
5 * 8 = 40
6 * 8 = 48
7 * 8 = 56
8 * 8 = 64
9 * 8 = 72
10 * 8 = 80
11 * 8 = 88
12 * 8 = 96
--- Fin ---
```

Observe que primero mostramos la fecha del sistema; luego ingresamos al programa el 7 de diciembre, primero el mes y luego el día, finalmente el programa nos muestra el resultado esperado.

- **Ejemplo 9**
  El siguiente programa permite visualizar la tabla de multiplicar de un número **n**. El valor de **n** se ingresa como parámetro en la línea de comandos.

prog0609.java

```
public class prog0609
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            int k = 0; // Contador
            int r;     // Variable auxiliar

            System.out.println( "Tabla de Multiplicar del numero: " + n
);
            while(++k <= 12){
                    r = n * k;
                    System.out.println( k + " * " + n + " = " + r );
            }
            System.out.println( "--- Fin ---" );

    }
}
```

La variable **k** se utiliza como contador. Observe que el incremento de esta variable se realiza en la misma expresión de condición, lo cual es totalmente válido.

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0609.java          [Enter]

C:\java2n100\cap6>java prog0609 8              [Enter]
Tabla de Multiplicar del número: 8
1 * 8 = 8
2 * 8 = 16
3 * 8 = 24
4 * 8 = 32
5 * 8 = 40
6 * 8 = 48
7 * 8 = 56
8 * 8 = 64
9 * 8 = 72
10 * 8 = 80
11 * 8 = 88
12 * 8 = 96
--- Fin ---
```

6.3.2 Bucle do – while

Esta instrucción de bucle permite repetir la ejecución de un grupo de instrucciones mientras se cumpla cierta condición. Es similar al bucle **while**, la diferencia está en que la condición se evalúa después de la primera iteración, mientras que en el bucle **while** la condición se evalúa antes de ejecutar cualquier iteración.

Sintaxis

```
do{
    // Bloque de instrucciones
} while ( condición );
```

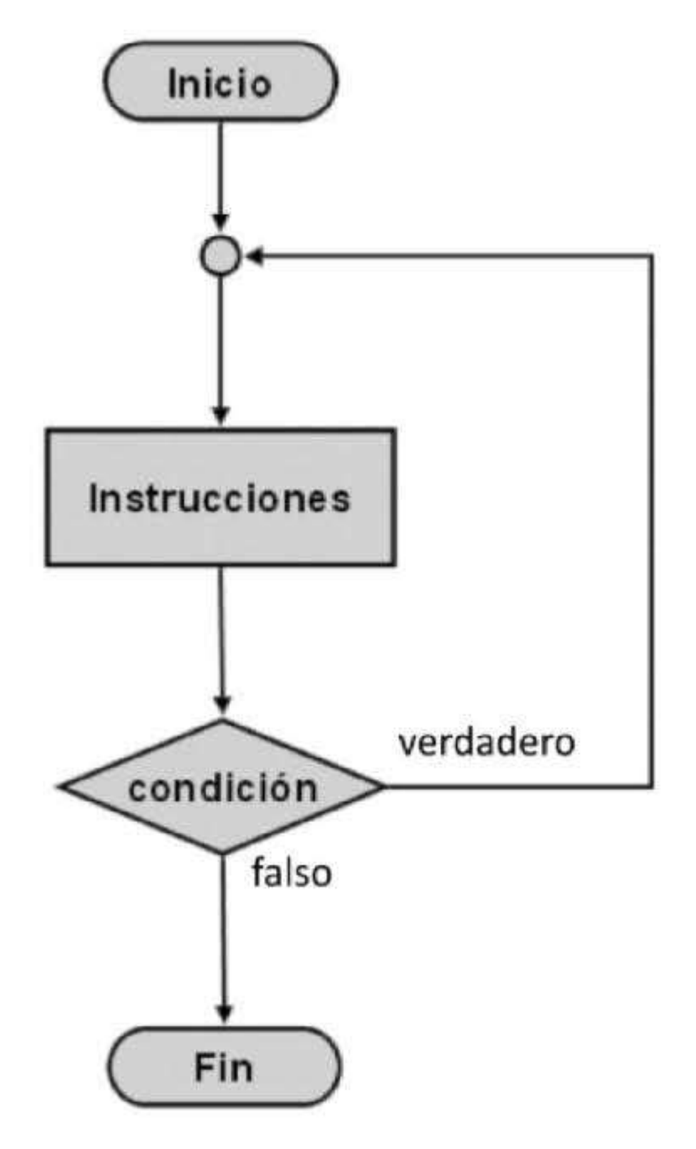

Fig. 6.2 Diagrama de flujo del bucle do - while

Note usted que para ejecutar la primera iteración no es necesario verificar la condición.

- **Ejemplo 10**
  El siguiente ejemplo imprime los **n** primeros términos de la serie de Fibonacci; los dos primeros términos son 0 y 1 respectivamente, para calcular un término k (Tk) se utiliza la siguiente fórmula:

```
Tk = Tk-1 + Tk-2
```

De tal manera que la serie toma la siguiente forma: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . . etc.

prog0610.java

```
public class prog0610
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            int a = 0, b = 1, k = 2, c;

            System.out.println( "Serie de Fibonacci" );
            System.out.print( a + "\t" + b );
            do{
                    k++;
                    c = a + b;
                    System.out.print( "\t" + c );
                    a = b;
                    b = c;
            }while (k < n);
            System.out.println( "\n--- Fin ---" );

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0610.java          [Enter]

C:\java2n100\cap6>java prog0610 8              [Enter]
Serie de Fibonacci
0       1       1       2       3       5       8       13
--- Fin ---

C:\java2n100\cap6>java prog0610 12             [Enter]
Serie de Fibonacci
0       1       1       2       3       5       8       13      21      34
55      89
--- Fin ---
```

Observe usted que el valor de n se pasa como parámetro en la línea de comando.

6.3.3 Bucle for

El bucle for está basado en tres expresiones, tal como se aprecia en su sintaxis.

Sintaxis

```
for( ExpInicialización; Condición; ExpFinalización ){

    // Bloque de instrucciones

}
```

ExpInicialización	Esta sección se ejecuta antes de la primera iteración, normalmente se utiliza para inicializar variables, como por ejemplo contadores.
Condición	Es la condición que determina si se inicia la ejecución de la primera iteración, y posteriormente las siguientes iteraciones.
ExpFinalización	Esta sección se ejecuta después del bloque de instrucciones propias del bucle.

En la Figura 6.3 podemos apreciar en forma gráfica cómo funciona el bucle for.

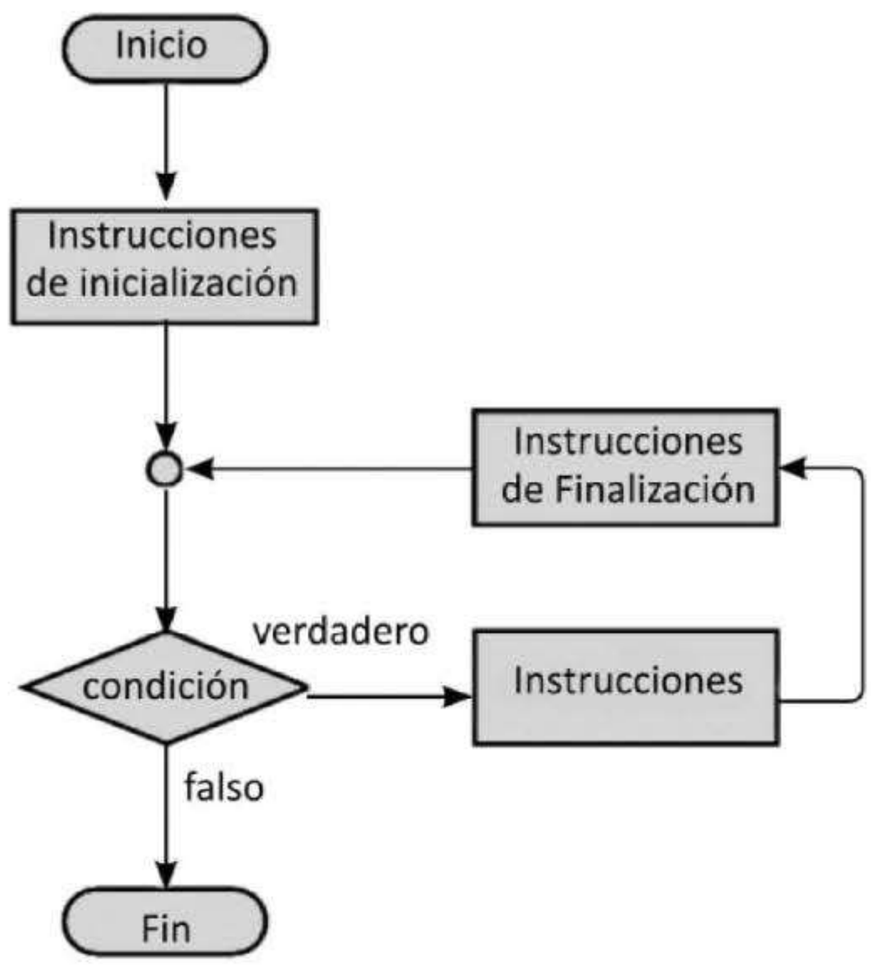

Fig. 6.3 Diagrama de flujo del bucle for

- **Ejemplo 11**
  Observa cómo se calcula el factorial de un número. El número se pasa como parámetro en la línea de comando.

prog0611.java

```
public class prog0611
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            int f = 1;

            for( int k=2; k<=n; k++ ){
                    f *= k;
            }

            System.out.println( "n: " + n );
            System.out.println( "Factorial: " + f );
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0611.java          [Enter]

C:\java2n100\cap6>java prog0611 5              [Enter]
n: 5
Factorial: 120

C:\java2n100\cap6>java prog0611 7              [Enter]
n: 7
Factorial: 5040
```

- **Ejemplo 12**
  A continuación elevamos un número **n** a una potencia **p**. La restricción es considerar solo números enteros, y para p solo positivos.

prog0602.java

```
public class prog0612
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            int p = Integer.parseInt( args[1] );
            int k, r;

            for( k=1, r=1; k<=p; k++ ){
                    r *= n;
            }

            System.out.println( "n: " + n );
            System.out.println( "p: " + p );
            System.out.println( "Resultado: " + r );

    }
}
```

Compilación:

```
C:\java2n100\cap6>javac prog0612.java          [Enter]
```

Para calcular 2 elevado al cubo, la ejecución debe ser así:

```
C:\java2n100\cap6>java prog0612 2 3            [Enter]
n: 2
p: 3
Resultado: 8
```

Para calcular -5 elevado al cubo, la ejecución debe ser así:

```
C:\java2n100\cap6>java prog0612 -5 3 [Enter]
n: -5
p: 3
Resultado: -125
```

Para calcular -5 elevado a la potencia 4, la ejecución debe ser así:

```
C:\java2n100\cap6>java prog0612 -5 4 [Enter]
n: -5
p: 4
Resultado: 625
```

Puede comprobar que los resultados son correctos. Observe que la expresión de inicialización del for tiene dos instrucciones separadas por coma (,); de igual forma la expresión de finalización puede tener más de una instrucción.

Bajo estas consideraciones podría calcular el factorial de 5 de la siguiente manera:

```
int n = 5, k, f;
for(k=2, f=1; k<=n; f*=k, k++ );
System.out.println( f );
```

O también podría ser así:

```
int n = 5, k, f;
for(k=2, f=1; k<=n; f*=k++ );
System.out.println( f );
```

- **Ejemplo 13**
  Aquí se muestra la flexibilidad de la instrucción for, utilizando solo su condición sería muy similar a **while** pero con **for**.

  Este ejemplo calcula el MCD y MCM de dos números enteros.

prog0613.java

```
public class prog0613
{
    public static void main(String[] args)
    {
            int n1 = Integer.parseInt( args[0] );
            int n2 = Integer.parseInt( args[1] );
            int n1_aux = n1, n2_aux = n2, mcd, mcm;
```

```
for( ; n1 != n2; )
                    if(n1 > n2) n1 -= n2;
                    else n2 -= n1;

            mcd = n1;
            mcm = n1_aux * n2_aux / mcd;

            System.out.println( "n1  => " + n1_aux );
            System.out.println( "n2  => " + n2_aux );
            System.out.println( "mcd => " + mcd );
            System.out.println( "mcm => " + mcm );

        }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0613.java          [Enter]

C:\java2n100\cap6>java prog0613 15 20          [Enter]
n1  => 15
n2  => 20
mcd => 5
mcm => 60
```

6.3.4 Bucle for con arreglos

Para procesar los elementos de un arreglo se puede utilizar el siguiente bucle:

```
for( k=0; k<lista.length; k++ ){

    // Instrucciones

}
```

En este caso lista es el arreglo:

```
int lista[] = {12,15,45,23,80}, k;
for( k=0; k<lista.length; k++ )
    System.out.println( lista[k] );
```

En este ejemplo estamos inicializando lista como un arreglo de enteros para luego imprimir dicha lista.

También podemos utilizar el siguiente formato:

```
for( tipo variable: arreglo ){

    // Instrucciones

}
```

Para imprimir los elementos de la lista del ejemplo anterior, tenemos la siguiente solución:

```
int lista[] = {12,15,45,23,80};
for( int n: lista )
System.out.println( n );
```

- **Ejemplo 14**
  Note que se trata de un programa que recibe a través de la línea de comando, una serie de números y lo que se trata es de calcular la suma de todos ellos.

prog0614.java

```
public class prog0614
{
    public static void main(String[] args)
    {
            int suma = 0;

            for( String dato: args )
                    suma += Integer.parseInt( dato );

            System.out.println( "Suma => " + suma );
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0614.java                [Enter]

C:\java2n100\cap6>java prog0614 15 20 23 56          [Enter]
Suma => 114

C:\java2n100\cap6>java prog0614 8 10 15 35 50 25     [Enter]
Suma => 143
```

6.3.5 Bucle for con colecciones

También podemos utilizar el bucle for para recorrer una colección. El formato es el siguiente:

```
for( tipo variable: colección ) {

    // Instrucciones

}
```

Vamos a suponer que amigos es una colección de elementos de tipo *String*. Para recorrer sus elementos e imprimirlos, la solución sería así:

```
for( String dato: amigos )
    System.out.println( dato );
```

- **Ejemplo 15**
  Creamos una colección de nombre **amigos** y utilizamos el bucle for para recorrer sus elementos y mostrarlos en pantalla.

prog0615.java

```
import java.util.ArrayList;

public class prog0615
{
    public static void main(String[] args)
    {
            ArrayList<String> amigos = new ArrayList<String>();

            amigos.add(new String("Sergio"));
            amigos.add(new String("Guino"));
            amigos.add(new String("Ricardo"));
            amigos.add(new String("Julio"));
            amigos.add(new String("Hugo"));

            for( String dato: amigos )
                    System.out.println( dato );

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0615.java          [Enter]

C:\java2n100\cap6>java prog0615                [Enter]
Sergio
Guino
Ricardo
Julio
Hugo
```

6.4 INSTRUCCIONES: BREAK Y CONTINUE

El funcionamiento de estas instrucciones se describe a continuación:

break	Permite salir de una instrucción switch, while, do-while y for.
continue	Permite saltar al inicio de un bucle.

- **Ejemplo 16**

Permite averiguar si un número es primo. Un número es primo cuando es divisible por 1 y por sí mismo.

Por ejemplo 4 no es primo porque es divisible por 2; 5 si es primo porque solo es divisible por 1 y 5.

prog0616.java

```
public class prog0616
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            String msg = "si es primo"; // Se asume que si es primo

            for( int k = 2; k < n; k++ )
                    if( n%k == 0 ) {
                            msg = "no es primo";
                            break;
                    }

            System.out.println( "n: " + n );
            System.out.println( "Resultado: " + msg );
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap6>javac prog0616.java          [Enter]

C:\java2n100\cap6>java prog0616 8              [Enter]
n: 8
Resultado: no es primo

C:\java2n100\cap6>java prog0616 7              [Enter]
n: 7
Resultado: si es primo

C:\java2n100\cap6>java prog0616 11             [Enter]
n: 11
Resultado: si es primo
```

Arreglos

CAPÍTULO 7

Capacidad: En este capítulo se otorgarán las definiciones acerca de los arreglos y los beneficios de su aplicación en el lenguaje de programación.

El uso de variables es la forma más simple de guardar datos en memoria durante la ejecución de un programa, pero resulta inadecuado para algunos procesos, por eso debemos recurrir a una estructura de datos que permita almacenar varios datos como una sola unidad, para luego procesarlos mediante bucles, y una de las posibilidades son los arreglos.

7.1 ¿QUÉ ES UN ARREGLO?

Un arreglo (*array* en inglés) es una estructura de datos conformada por un conjunto de variables del mismo tipo, agrupadas bajo un mismo nombre a las que accedemos mediante un índice.

amigos

0	Hugo
1	Sergio
2	Ricardo
3	Julio
4	Guino
5	César

Fig.7.1 Ejemplo de un arreglo

Tenemos un ejemplo de lo que sería un arreglo de tipo *String*, todos sus elementos son cadenas, en este caso los nombres de mis amigos (ver Fig. 7.1).

El primer elemento tiene índice cero (0), y el último es el tamaño del arreglo disminuido en uno. Si tenemos un arreglo de tamaño 6, el primer elemento tendrá índice 0 y el último índice 5.

Los arreglos pueden ser de una dimensión, se les denomina vectores; de dos dimensiones, se les llama matrices; así se puede construir de más dimensiones pero no es usual.

7.2 ARREGLOS UNIDIMENSIONALES – VECTORES

Los arreglos unidimensionales o vectores, representan una lista de variables contiguas y homogéneas (del mismo tipo), a las cuales podemos acceder mediante un índice (ver Fig 7.2).

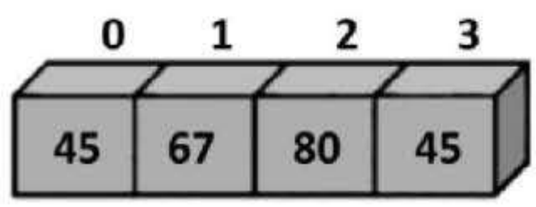

Fig. 7.2 Arreglo unidimensional

7.2.1 Declaración de arreglos

Sintaxis

```
tipo variable_arreglo[];
```

o

```
tipo[] variable_arreglo;
```

En este caso, estamos creando la variable que va a contener el arreglo, más no el arreglo en sí.

Veamos a continuación algunos ejemplos ilustrativos:

```
lista1 = new int[10];          // De tamaño 10 (10 elementos)
lista2 = new char[15];         // De tamaño 15 (15 elementos)
lista3 = new String[8];        // De tamaño 8 (8 elementos)
```

7.2.2 Creación de arreglos

Para la creación de los arreglos debemos utilizar el operador **new**.

Sintaxis

```
variable_arreglo = new tipo[tamaño];
```

Es necesario haber declarado la variable **variable_arreglo** previamente.

Veamos a continuación algunos ejemplos ilustrativos:

```
lista1 = new int[10];                // De tamaño 10 (10 elementos)
lista2 = new char[15];               // De tamaño 15 (15 elementos)
lista3 = new String[8];              // De tamaño 8 (8 elementos)
```

7.2.3 Acceso a elementos de un arreglo

Para acceder a los elementos de un arreglo debemos usar el índice del elemento.

Sintaxis:

```
variable_arreglo[índice]
```

Por ejemplo para guardar en la posición 3 del arreglo lista3 el nombre Claudia, la instrucción es:

```
lista[3] = "Claudia";
```

Para imprimir el contenido de lista1, las instrucciones son:

```
for( int k = 0; k < 10; k++ )
    System.out.println( lista1[k] );
```

7.2.4 Inicialización de un arreglo

Podemos crear e inicializar un arreglo al mismo tiempo.

Sintaxis

```
tipo variable_arreglo[] = { elemento1, elemento2, . . . } ;
```

Veamos el siguiente ejemplo:

```
String ciudades[] = { "Trujillo", "Chiclayo", "Piura", "Tumbes" } ;

for( String dato: ciudades )
    System.out.println( dato );
```

7.2.5 Averiguar el tamaño de un arreglo

Los arreglos tienen una serie de propiedades y métodos entre los que encontramos la propiedad **length**, la cual nos informa sobre la longitud del arreglo.

Esta propiedad nos puede ayudar a realizar el recorrido del arreglo si utilizamos un bucle, como se ilustra en el siguiente ejemplo:

```
String dbs[] = { "MySQL", "PostgreSQL", "Oracle", "SQL Server" } ;

for( int k = 0; k < dbs.length; k++ )
    System.out.println( dbs[k] );
```

7.3 Aplicación de arreglos unidimensionales

7.3.1 Generación de números aleatorios

En Java tenemos la función **random()** que pertenece a la clase Math. Esta genera números aleatorios mayores o iguales que cero (0) y menores que uno (1).

- **Ejemplo 1**

Aquí se muestra como ordenar un vector de números enteros generados en forma aleatoria en el rango de 1 a 20.

En el arreglo lista, se genera la lista de números aleatorios; el tamaño de la lista pasa como parámetro en la línea de comandos, luego sacamos una copia en lista2. El arreglo lista2 es el que se ordena aplicando el método burbuja, y finalmente se imprimen ambas listas.

prog0701.java

```
public class prog0701
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            int lista[] = new int[n]; // Lista original
            int lista2[];  // Lista ordenada

            // Generar Lista
            for( int k = 0; k < n; k++ )
                    lista[k] = (int)( Math.random() * 20 + 1 );

            // Sacar una copia de lista en lista2
            lista2 = lista.clone();
```

```
// Ordenar Lista - Metodo Burbuja
            for( int i = 0; i < (n-1); i++ ) {
                    for( int j = i + 1; j < n; j++ ) {
                            if( lista2[i] < lista2[j] ) {
                                    int temp = lista2[i];
                                    lista2[i] = lista2[j];
                                    lista2[j] = temp;
                            }
                    }
            }

            // Imprimir lista ordenada
            System.out.println( "Lista Generada\tLista Ordenada" );
            for( int k = 0; k < n; k++ ) {
                    System.out.println( "\t" + lista[k] + "\t\t" + lista2[k]
);
            }

            System.out.println( "\t----- Fin -----" );

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap7>javac prog0701.java          [Enter]

C:\java2n100\cap7>java prog0701 5              [Enter]
Lista Generada  Lista Ordenada
        18              20
        9               18
        4               11
        11              9
        20              4
        ----- Fin -----
```

- **Ejemplo 2**

  Se genera una lista de **n** números aleatorios, luego se ordena para encontrar la mediana.

  Para encontrar la mediana, primero se debe ordenar la lista. Si el número de elementos es impar, la mediana es el número central de la lista, pero si el número de elementos es par, la mediana es el promedio de los dos números centrales.

prog0701.java

```
public class prog0702
{
    public static void main(String[] args)
    {
            int n = Integer.parseInt( args[0] );
            int lista[] = new int[n]; // Lista original
            int lista2[];  // Lista ordenada
```

```
// Generar Lista
            for( int k = 0; k < n; k++ ) {
                    lista[k] = (int)( Math.random() * 100 + 1 );
            }

            // Ordenar Lista - Metodo Burbuja
            for( int i = 0; i < (n-1); i++ ) {
                    for( int j = i + 1; j < n; j++ ) {
                            if( lista[i] < lista[j] ) {
                                    int temp = lista[i];
                                    lista[i] = lista[j];
                                    lista[j] = temp;
                            }
                    }
            }

            // Obtener la Mediana
            int p = n / 2;
            float mediana;
            if( n % 2 == 0 )
                    mediana = (float)( lista[p] + lista[p-1] ) / 2;
            else
                    mediana = lista[p];

            // Imprimir lista ordenada
            System.out.println( "Lista Generada" );
            for( int k = 0; k < n; k++ )
                    System.out.println( "\t" + lista[k] );

            // Imprimir mediana
            System.out.println( "Mediana: " + mediana );
            System.out.println( "----- Fin -----" );

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap7>javac prog0702.java[Enter]

C:\java2n100\cap7>java prog0702 5     [Enter]
Lista Generada
        98
        89
        73
        65
        36
Mediana: 73.0
----- Fin -----

C:\java2n100\cap7>java prog0702 4     [Enter]
Lista Generada
        89
        80
        75
        7
Mediana: 77.5
----- Fin -----
```

- **Ejemplo 3**

  Se tiene una lista de amigos, entonces se trata de ubicar la posición de uno de ellos, para lo cual ingresamos el nombre del amigo a buscar en la línea de comando como parámetro.

prog0703.java

```
public class prog0703
{
    public static void main(String[] args)
    {
            int N = 10;
        String amigos[] = new String[N];
            String nombre = args[0];

            // Lista de amigos
        amigos[0] = "Sergio";
        amigos[1] = "Claudia";
        amigos[2] = "Hugo";
        amigos[3] = "Delia";
        amigos[4] = "Julio";
        amigos[5] = "Karla";
        amigos[6] = "Ricardo";
        amigos[7] = "Mariela";
        amigos[8] = "Laura";
        amigos[9] = "Adriana";

            // Ubicar amigo
        int p = -1;
        for(int j=0; j<N; j++  ){
            if (amigos[j].equals(nombre)){
                p = j;
                            break;
            }
        }

            // Imprimir lista de amigos
        System.out.println("Lista de Amigos\n");
        for(int j = 0; j < amigos.length; j++)
            System.out.println( j + "\t" + amigos[j] );


            // Imprimir resultado
            if (p == -1)
                    System.out.println( "\n" + nombre + " no existe en la
lista" );
            else
                    System.out.println( "\n" + nombre + " esta en la po-
sicion " + p );

            System.out.println( "\n\t----- Fin -----" );

        }
}
```

Compilación y ejecución:

```
C:\java2n100\cap7>javac prog0703.java                    [Enter]

C:\java2n100\cap7>java prog0703 "Karla"                  [Enter]
Lista de amigos

0       Sergio
1       Claudia
2       Hugo
3       Delia
4       Julio
5       Karla
6       Ricardo
7       Mariela
8       Laura
9       Adriana

Karla esta en la posición 5

        ----- Fin -----
```

El nombre del amigo a buscar lo debe ingresar tal como está en la lista, respetando mayúsculas y minúsculas.

- **Ejemplo 4**

Mostramos cómo hacer una búsqueda en un arreglo, se trata de una búsqueda secuencial.

Se tienen dos arreglos; el primero de ellos es **amigos** donde se tiene los nombres de amigos, y el segundo arreglo es **edades** donde se generan sus respectivas edades en forma aleatoria.

El programa se encarga de ubicar al amigo de mayor edad, para eso se asume que es el primero de la lista, con esta suposición se recorre el resto de la lista para comparar con cada uno de ellos y ubicar finalmente el mayor edad.

prog0704.java

```
public class prog0704
{
    public static void main(String[] args)
    {
            int N = 10;
        String amigos[] = new String[N];
            int edades[] = new int[N];
```

```
// Lista de amigos
        amigos[0] = "Sergio";
        amigos[1] = "Claudia";
        amigos[2] = "Hugo";
        amigos[3] = "Delia";
        amigos[4] = "Julio";
        amigos[5] = "Karla";
        amigos[6] = "Ricardo";
        amigos[7] = "Mariela";
        amigos[8] = "Laura";
        amigos[9] = "Adriana";

         // Generar edades
         for(int j = 0; j < edades.length; j++)
        edades[j] = (int)(Math.random()*100);

         // Ubicar al de mayor edad
         // Se asume que el de mayor edad es el primero de la lista
    int mayor = edades[0], p = 0;
         // Luego se hace un recorrido del arreglo
    for(int j =1; j < N; j++  ){
        if (mayor < edades[j]){
            mayor = edades[j];
            p = j;
        }
    }

         // Imprimir lista de amigos
    System.out.println("Lista de Amigos\n");
    for(int j = 0; j < amigos.length; j++){
        System.out.println( j + "\t" + amigos[j] + "\t" + edades[j] );
```

Compilación y ejecución:

```
C:\java2n100\cap7>javac prog0704.java        [Enter]

C:\java2n100\cap7>java prog0704              [Enter]
Lista de amigos

0       Sergio              13
1       Claudia             80
2       Hugo        72
3       Delia               4
4       Julio               76
5       Karla               43
6       Ricardo     76
7       Mariela             65
8       Laura               3
9       Adriana     70


Resultado de la búsqueda

Posicion: 1
Amigo: Claudia
Edad: 80

        ----- Fin -----
```

```
C:\java2n100\cap7>java prog0704      [Enter]
Lista de amigos

0       Sergio              48
1       Claudia             20
2       Hugo        46
3       Delia               52
4       Julio               29
5       Karla               71
6       Ricardo     28
7       Mariela             83
8       Laura               33
9       Adriana     46

Resultado de la búsqueda

Posición: 7
Amigo: Mariela
Edad: 83

        ----- Fin -----
```

7.4 ARREGLOS BIDIMENSIONALES – MATRICES

Podemos considerar la siguiente situación:
«*Si cada elemento de un arreglo unidimensional es otro arreglo, entonces tenemos un arreglo de dos dimensiones. Si los elementos de este segundo arreglo son otros arreglos, entonces tenemos un arreglo de tres dimensiones, y así sucesivamente.*»

Un arreglo bidimensional se representa por un conjunto de filas y columnas (ver Fig. 7.3). Para acceder a un elemento en particular se necesita conocer la fila y columna donde se encuentra.

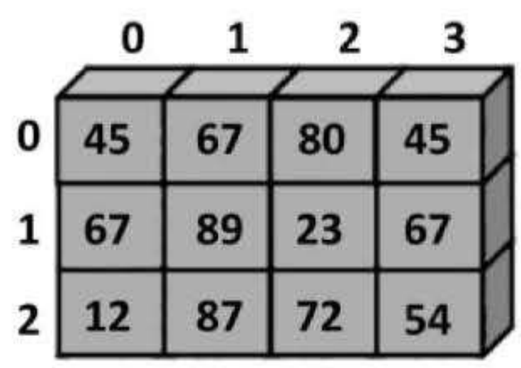

Fig. 7.3 Representación de un arreglo bidimensional

7.4.1 Declaración de arreglos bidimensionales

Sintaxis

```
tipo variable_arreglo[][];
```

o

```
tipo[][] variable_arreglo;
```

Veamos a continuacion algunos ejemplos:

```
int mat1[][];
String mat2[][];
```

Debe tener presente que solo está creando la variable que va a contener el arreglo, pero no el arreglo en sí.

7.4.2 Creación de arreglos

Para la creación del arreglo debemos utilizar el operador **new**.

Sintaxis

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

Es necesario haber declarado la variable **variable_arreglo** previamente.

Veamos algunos ejemplos:

```
mat1 = new int[10][5];
mat2 = new String[5][10];

int mat3[][] = new int[4][];
mat3[0] = new int[5];
mat3[1] = new int[5];
mat3[2] = new int[5];
mat3[3] = new int[5];
```

7.4.3 Acceso a los elementos de una matriz

Para acceder a los elementos de una matriz necesitamos conocer la fila y columna donde está ubicado el elemento.

Sintaxis

```
variable_arreglo[fila][columna]
```

Por ejemplo para guardar en la fila 2, columna 3 de la matriz mat1 el valor 20, la instrucción es:

```
mat1[2][3] = 20;
```

Para imprimir el contenido de **mat1**, las instrucciones son:

```
for( int i = 0; i < mat.length; i++ )
{
    for( int j = 0; j < mat[i].length; j++ )
            System.out.print( mat[i][j] + "\t");
    System.out.println();
}
```

7.4.4 Inicialización de una matriz

Podemos crear e inicializar una matriz al mismo tiempo.

Sintaxis

```
tipo variable_arreglo[][] = { {elemento,elemento,...}, {elemento,elemento,...} };
```

Veamos el siguiente ejemplo:

```
int[][] mat = { {13,15,18}, {67,23,56}, {15,45,23} };

for( int i = 0; i < mat.length; i++ )
{
    for( int j = 0; j < mat[i].length; j++ )
            System.out.print( mat[i][j] + "\t");
    System.out.println();
}
```

7.5 APLICACIÓN DE ARREGLOS BIDIMENSIONALES

- **Ejemplo 5**
  Indicamos la suma de matrices, para lo cual es necesario que ambas matrices tengan las mismas dimensiones. Las matrices se generan de manera aleatoria.

  Para imprimir las matrices se a creado el método **sayMatriz**, que se encarga de imprimir una matriz que se le pasa como parámetro.

prog0704.java

```
public class prog0705
{
    public static void main(String[] args)
    {
            int mat1[][] = new int[4][5];
            int mat2[][] = new int[4][5];
            int suma[][] = new int[4][5];
```

```
// Generar Matriz 1
            for( int i=0; i<mat1.length; i++ )
                    for( int j=0; j<mat1[i].length; j++ )
                            mat1[i][j] = (int)(Math.random()*20 + 10);

            // Generar Matriz 2
            for( int i=0; i<mat2.length; i++ )
                    for( int j=0; j<mat2[i].length; j++ )
                            mat2[i][j] = (int)(Math.random()*20 + 10);

            // Calcular Suma
            for( int i=0; i<mat2.length; i++ )
                    for( int j=0; j<mat2[i].length; j++ )
                            suma[i][j] = mat1[i][j] + mat2[i][j];

            // Imprimir Matrices
            sayMatriz( "\nMatriz 1", mat1 );
            sayMatriz( "\nMatriz 2", mat2 );
            sayMatriz( "\nMatriz Suma", suma );
    }

    private static void sayMatriz( String titulo, int mat[][] ){
            System.out.println( titulo + "\n" );
            for( int i = 0; i < mat.length; i++ )
            {
                    for( int j = 0; j < mat[i].length; j++ )
                            System.out.print( mat[i][j] + "\t");
                    System.out.println("");
            }
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap7>javac prog0705.java[Enter]

C:\java2n100\cap7>java prog0705      [Enter]

Matriz 1

22      29      27      10      27
25      14      24      29      11
12      24      18      26      12
29      20      19      26      10

Matriz 2

27      16      13      14      14
28      20      28      29      11
10      22      25      14      29
13      23      29      24      28

Matriz Suma

49      45      40      24      41
53      34      52      58      22
22      46      43      40      41
42      43      48      50      38
```

- **Ejemplo 6**
  Indicamos el producto de dos matrices, para poder multiplicar dos matrices se debe cumplir:

```
prodij = mat1ik * mat2kj
```

Por lo tanto, para encontrar prod[3][4], se opera con la fila 3 de la mat1 y con la columna 4 de mat2, suponiendo que son 2 columnas de mat1; en consecuencia deben ser 2 filas de mat2, la operacion sería así:

```
prod[3][4] = mat1[3][1] * mat2[1][4] + mat1[3][2] * mat2[2][4];
```

Aplicando este mismo razonamiento se calcula todos los elementos de la matriz producto, lo más recomendable es hacerlo en un bucle; por ejemplo un for, tal como se ilustra en el programa.

prog0706.java

```
public class prog0706
{
    public static void main(String[] args)
    {
        int mat1[][] = new int[4][2];
        int mat2[][] = new int[2][3];
        int prod[][] = new int[4][3];

        // Generar Matriz 1
        for( int i=0; i<mat1.length; i++ )
            for( int j=0; j<mat1[i].length; j++ )
                mat1[i][j] = (int)(Math.random()*10);

        // Generar Matriz 2
        for( int i=0; i<mat2.length; i++ )
            for( int j=0; j<mat2[i].length; j++ )
                mat2[i][j] = (int)(Math.random()*10);

        // Calcular Producto
        for( int i=0; i<prod.length; i++ )
            for( int j=0; j<prod[i].length; j++ )
            {
                int suma = 0;
                for( int k = 0; k<2; k++ )
                    suma += mat1[i][k] * mat2[k][j];
                prod[i][j] = suma;
            }

        // Imprimir Matrices
        sayMatriz( "\nMatriz 1", mat1 );
        sayMatriz( "\nMatriz 2", mat2 );
        sayMatriz( "\nMatriz Producto", prod );
```

```
        System.out.println( "\n\t----- Fin -----" );

        }

        private static void sayMatriz( String titulo, int mat[][] ){
                System.out.println( titulo + "\n" );
                for( int i = 0; i < mat.length; i++ )
                {
                        for( int j = 0; j < mat[i].length; j++ )
                                System.out.print( mat[i][j] + "\t");
                        System.out.println("");
                }
        }
    }
```

Compilación y ejecución:

```
        C:\java2n100\cap7>javac prog0706.java     [Enter]

    C:\java2n100\cap7>java prog0706               [Enter]

    Matriz 1

    6       3
    3       9
    8       6
    2       6

    Matriz 2

    2       1       9
    3       2       2

    Matriz Producto

    21      12      60
    33      21      45
    34      20      84
    22      14      30

            ----- Fin -----
```

Clases y objetos

CAPÍTULO 8

Las clases son el núcleo de Java. Son la construcción lógica sobre la que se basa el lenguaje Java, ya que definen la forma y naturaleza de un objeto, y constituyen los fundamentos de la programación orientada a objetos. Cualquier concepto que quiera implementarse en un programa Java debe ser encapsulado dentro de una clase.

Este capítulo esta dedicado a las clases y la forma como se deben utilizar, en capítulos posteriores veremos aspectos más avanzados de este importante tema.

8.1 CLASES VERSUS OBJETOS

Una clase es una representación de un tipo de objeto en particular, y define un nuevo tipo de dato. Las clases se utilizan para crear objetos de ese tipo de clase en particular, esto es, crear objetos que incorporen todos los componentes especificados dentro de la clase.

Una clase está conformada básicamente por atributos y operaciones; los atributos representan los datos de la clase y se implementan mediante variables; por otro lado, las operaciones se implementan a través de los métodos. Los atributos se deben manipular solamente a través de los métodos.

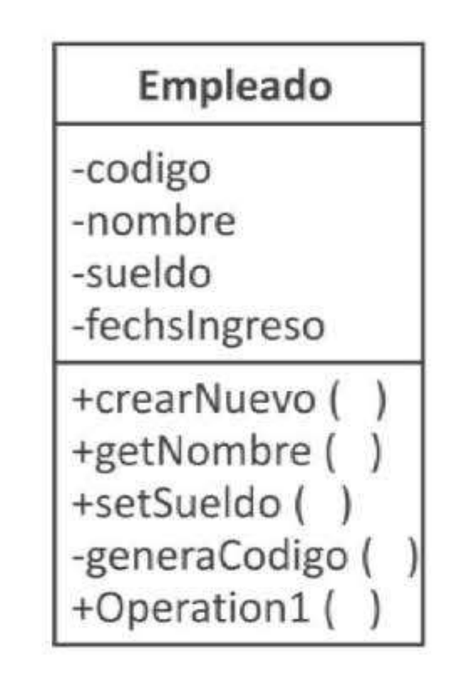

Fig. 8.1 Ejemplo de una clase utilizando notación UML

Las clases son la base de la **Programación Orientada a Objetos (POO)** y sus características más importantes son:

8.1.1 Encapsulación

Las clases pueden ser declaradas como públicas (*public*) y como *package* (accesibles solo para otras clases del *package*). Los atributos miembros y los métodos pueden ser *public, private, protected* y *package*. De esta forma, se puede controlar el acceso y evitar un uso inadecuado.

8.1.2 Herencia

Una clase puede derivar de otra (*extends*) y, en ese caso, hereda todos sus atributos y operaciones. Una clase derivada puede añadir nuevos atributos y operaciones y/o redefinir los atributos y operaciones heredados.

8.1.3 Polimorfismo

Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo. Esto facilita la programación y el mantenimiento del código.

Para utilizar una clase debe instanciarse, esto se realiza con el operador **new**, por ejemplo, supongamos que tenemos la clase **Ventana**:

```
public class Ventana{

    public int x;
    public int y;

    . . .
    . . .

    public void pintar(){

            . . .
            . . .

    }

}
```

Para crear un objeto o instancia de esta clase tenemos dos alternativas, la primera en la misma línea de declaración, tal como se ilustra a continuación:

```
Ventana vent1 = new Ventana();
```

Y la otra, en forma separada:

```
Ventana vent2;
vent2 = new Ventana();
```

Para acceder a los componentes (variable o campos, y métodos) de un objeto, debemos utilizar el operador punto (.); por ejemplo para modificar la posición de la ventana:

```
vent1.x = 30;
vent1.y = 80;
```

Para ejecutar el método **pintar** la instrucción es la siguiente:

```
vent1.pintar();
```

Para resumir, podemos afirmar que una clase es la definición de un tipo de objeto y está compuesta por atributos y operaciones; mientras que el objeto es la instanciación de una clase, y está conformado por variables o campos, y métodos.

8.2 DEFINICIÓN DE UNA CLASE

Sintaxis

```
class Nombre_de_Clase{
    // Cuerpo de la clase
}
```

Por ejemplo si queremos definir la clase **Empleado**, la instrucción es:

```
class Empleado{
    // Cuerpo de la clase
}
```

Es importante recordar que el nombre del archivo debe ser igual al nombre de la clase seguido de **.java**; en este caso debe ser **Empleado.java**.

8.3 DECLARACIÓN DE OBJETOS

Como ya se mencionó, para crear los objetos se debe utilizar el operador **new**.

8.3.1 Operador new

El operador **new** crea el objeto, esto quiere decir que asigna memoria al objeto. Para utilizar el operador **new** tenemos dos sintaxis.

Sintaxis 1

```
Nombre_Clase Nombre_Variable;
Nombre_Variable = new Nombre_Clase();
```

En la primera instrucción se define la variable que apuntará al objeto que se crea en la segunda instrucción.

Sintaxis 2

```
Nombre_Clase Nombre_Variable = new Nombre_Clase();
```

En este caso tanto la creación de la variable y del objeto se realiza en la misma instrucción.

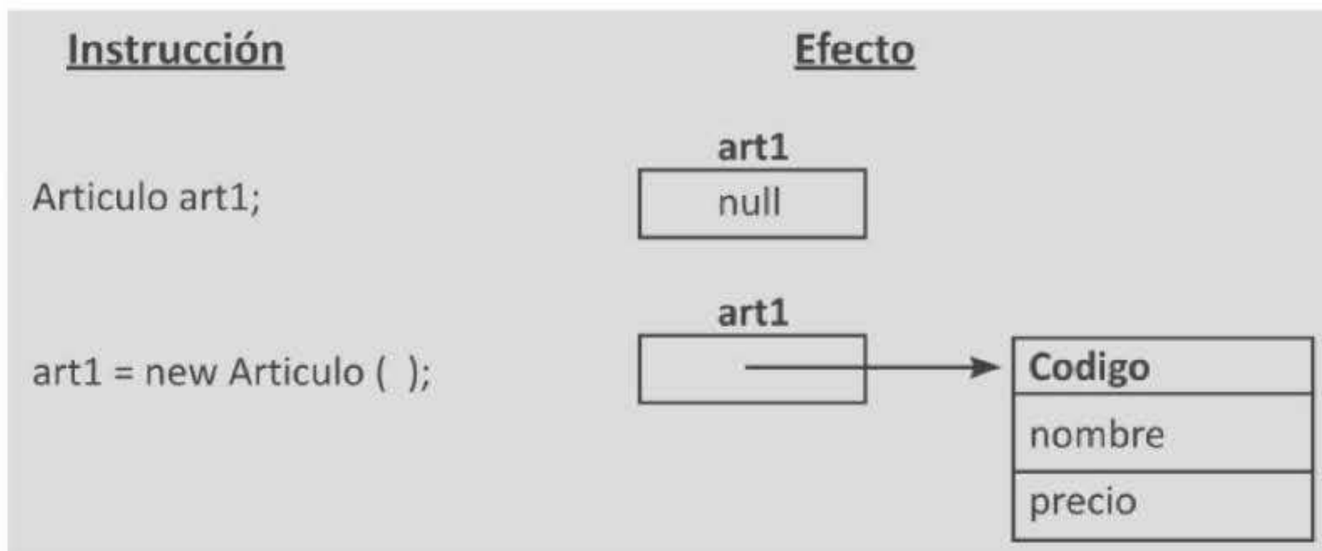

Fig. 8.2 Instanciación de una clase (creación de un objeto)

8.3.2 Asignación de objetos

Cuando creamos un objeto internamente existe un puntero al que no tenemos acceso; pero cuando asignamos objetos, lo que realmente estamos asignando son las direcciones de memoria donde están definidos los objetos (ver Fig. 8.3).

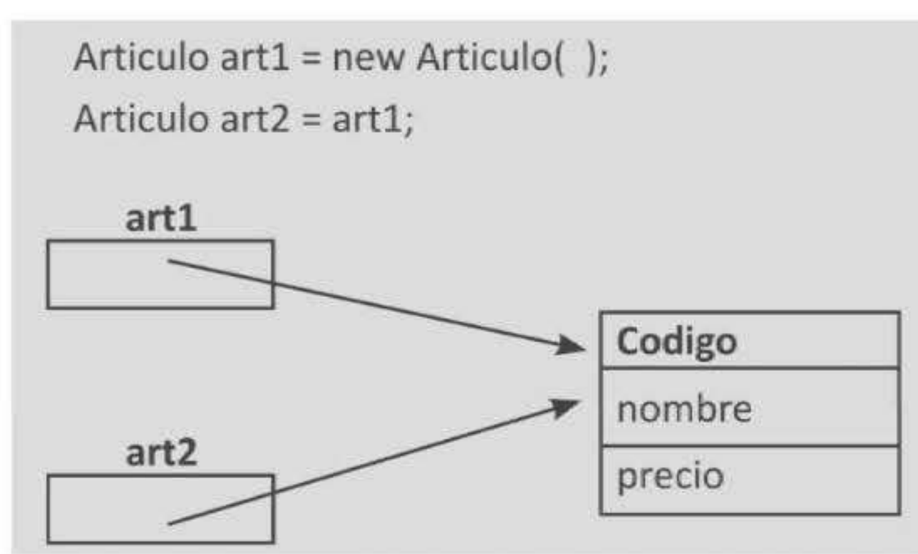

Fig. 8.3 Asignación de objetos

8.4 DEFINICIÓN DE VARIABLES O CAMPOS

Sintaxis

```
[Tipo_Acceso] Tipo nombre;
```

El **Tipo_Acceso** puede ser:

public	Indica que se puede acceder al campo desde cualquier programa. También es accesible por clases derivadas (subclases).
protected	Indica que solo se puede acceder al campo desde métodos de la misma clase y clases derivadas (subclases).
private	Indica que solo se puede acceder al campo desde métodos de la misma clase.

Cuando no se especifica el **Tipo_Acceso**, el acceso al campo es permitido por métodos de la misma clase y desde clases que se encuentran en el mismo paquete.

- **Ejemplo 1**
  Para ilustrar tenemos la clase **Artículo** que tiene definido 3 atributos.

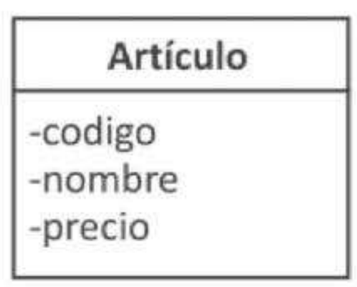

Fig. 8.4 Clase Artículo

La implementación de esta clase se ilustra a continuación:

Articulo.java

```
public class Articulo{
    public int codigo;
    public String nombre;
    public double precio;

}
```

Esta clase define un nuevo tipo de dato constituido por tres campos. El siguiente programa indica como se debe usar la clase **Artículo**.

prog0801.java

```
public class prog0801{
    public static void main(String[] args)   {
            // Creación de dos objetos
            Articulo art1 = new Articulo();
            Articulo art2 = new Articulo();
            Articulo art3;
            // Asignación de datos
            art1.codigo = 100;
            art1.nombre = "Televisor";
            art1.precio = 500;
            art2.codigo = 200;
            art2.nombre = "Plancha";
            art2.precio = 80;
            art3 = art1;
            // Imprimir articulos
            System.out.println(art1.codigo + "-" + art1.nombre + "-" + art1.
precio);
            System.out.println(art2.codigo + "-" + art2.nombre + "-" + art2.
precio);
            System.out.println(art3.codigo + "-" + art3.nombre + "-" + art3.
precio);
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap8>javac prog0801.java          [Enter]

C:\java2n100\cap8>java prog0801                [Enter]
100-Televisor-500.0
200-Plancha-80.0
100-Televisor-500.0
```

Se deduce de este resultado que tanto **art1** y **art3** apuntan al mismo objeto.

8.5 DEFINICIÓN DE MÉTODOS

Sintaxis

```
[Tipo_Acceso] Tipo_Dato nombre_método( lista_de_parámetros ) {

    // Cuerpo del método;

};
```

El **Tipo_Acceso** puede ser:

public	Indica que se puede acceder al método desde cualquier programa. También es accesible por clases derivadas (subclases).
protected	Indica que solo se puede acceder al método desde de la misma clase y clases derivadas (subclases).
private	Indica que solo se puede acceder al método desde de la misma clase.

Cuando no se especifica el **Tipo_Acceso**, el acceso al método es permitido desde la misma clase y desde clases que se encuentran en el mismo paquete.

El **Tipo_Dato** determina el tipo de dato que retorna el método. Puede ser cualquier tipo válido, incluso los tipos de clases creados por usted; si el método no retorna ningún valor se debe especificar **void**.

Los parámetros se especifican de la siguiente manera:

```
tipo1 parametro1, tipo2 parametro2, . . .
```

Los parámetros son variables que reciben los valores de los argumentos que se pasan al método cuando este es invocado. Si el método no tiene parámetros, la lista de parámetros estará vacía.

Los métodos que retornan un tipo diferente a **void**, deben retornar un valor a la instrucción que realiza la llamada mediante la instrucción **return** utilizando la siguiente sintaxis:

```
return expresión;
```

Expresión representa el valor que retorna el método.

- **Ejemplo 2**
  Ilustramos el uso de métodos, para eso se define una clase de nombre **Calculadora** que tiene definido dos atributos públicos **n1** y **n2**, y dos operaciones suma() y producto().

Calculadora
+n1 +n2
+suma ( ) +producto ( )

Fig. 8.5 Clase Operación

A continuación, la implementación de la clase:

Calculadora.java

```
public class Calculadora{
    public int n1;
    public int n2;

    public int suma() {
            return (n1 + n2);
    }

    public int producto() {
            return (n1 * n2);
    }
}
```

El programa para ilustrar el uso de la clase **Calculadora** es el siguiente:

prog0802.java

```
public class prog0802
{
    public static void main(String[] args)
    {
            Calculadora obj = new Calculadora();
            obj.n1 = 10;
            obj.n2 = 8;
            System.out.println("n1 = " +obj.n1);
            System.out.println("n2 = " +obj.n2);
            System.out.println("Suma = " +obj.suma());
            System.out.println("Producto = " +obj.producto());
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap8>javac prog0802.java          [Enter]

C:\java2n100\cap8>java prog0802                [Enter]
n1 = 10
n2 = 8
Suma = 18
Producto = 80
```

Como se observa en el ejemplo, después de asignar los datos e invocando a los métodos tenemos acceso al resultado esperado.

- **Ejemplo 3**
  Tenemos una segunda versión de la clase **Calculadora**, donde las operaciones tienen parámetro (ver Fig. 8.6).

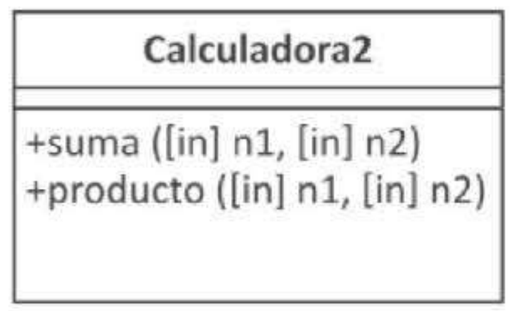

Fig. 8.6 Clase Calculadora versión 2

A continuación la implementación de la clase:

Calculadora2.java

```
public class Calculadora2{

        public int suma(int n1, int n2) {
                return (n1 + n2);
        }

        public int producto(int n1, int n2) {
                return (n1 * n2);
        }

}
```

El programa que ilustra el uso de la clase **Calculadora2** es el siguiente:

Calculadora3.java

```
public class prog0803
{
    public static void main(String[] args)
    {
            int a = 15;
            int b = 8;
            Calculadora2 obj = new Calculadora2();
            System.out.println("n1 = " +a);
            System.out.println("n2 = " +b);
            System.out.println("Suma = " +obj.suma(a,b));
            System.out.println("Producto = " +obj.producto(a,b));
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap8>javac prog0803.java          [Enter]

C:\java2n100\cap8>java prog0803                [Enter]
n1 = 15
n2 = 8
Suma = 23
Producto = 120
```

En este caso los valores con los que debe operar el método se pasan como parámetros.

8.6 OCULTANDO LOS DATOS

Uno de los fundamentos de la Programación Orientada a Objetos (POO) es que el usuario debe tener acceso a los datos que son de su interés, y del modo que le corresponde. Por ejemplo solo lectura, solo escritura, o ambos.

Para conseguir esto se debe implementar los atributos como privados, y los métodos para acceder a ellos. Existe un estándar para definir este método; por ejemplo si el atributo es **nombre**, los métodos son:

setNombre	Este método permite asignar una valor al atributo.
getNombre	Este método permite leer el valor del atributo.

Como puede apreciar existen dos prefijos, el prefijo **set** que se utiliza para asignar un valor al atributo y el prefijo **get** para leer el valor del atributo. De esta manera, podemos seleccionar si se implementa el método **set**, **get** o ambos, y restringir el nivel de acceso a los datos.

Otra posibilidad que nos da la implementación de estos métodos, es poder agregar funcionalidad que puede servir para verificar, por ejemplo si el dato que se está asignando es correcto o no.

- **Ejemplo 4**
  Ilustramos el uso de métodos para acceder a los campos de un objeto. La definición de clase se observa en la Fig. 8.7.

Fig. 8.7 Clase Artículo2

A continuación, observe la implementación de la clase **Articulo2**:

Articulo2.java

```
public class Articulo2
{
    private int código;
    private String nombre;
    private double preCosto;
    private double preVenta;

    public void setCódigo(int nCódigo){
            código = nCódigo;
    }

    public void setNombre(String sNombre){
            nombre = sNombre;
    }


    public void setPreCosto(double nPreCosto ){
            preCosto = nPreCosto;
            preVenta = preCosto * 1.4;
    }

    public int getCódigo(){
            return código;
    }

    public String getNombre(){
            return nombre;
    }
```

```
        public double getPreCosto(){
                return preCosto;
        }

        public double getPreVenta(){
                return preVenta;
        }

    }
```

El programa que ilustra el uso de esta clase es el siguiente:

prog0804.java

```
    public class prog0804
    {
        public static void main(String[] args)
        {
                Artículo2 art1 = new Articulo2();
                art1.setCódigo( 1000 );
                art1.setNombre( "Auto" );
                art1.setPreCosto( 15000 );
                System.out.println("Código = " + art1.getCódigo() );
                System.out.println("Nombre = " + art1.getNombre() );
                System.out.println("Precio Costo = " + art1.getPreCosto() );
                System.out.println("Precio Venta = " + art1.getPreVenta() );
        }
    }
```

Compilación y ejecución:

```
    C:\java2n100\cap8>javac prog0804.java          [Enter]

    C:\java2n100\cap8>java prog0804                [Enter]
    Codigo = 1000
    Nombre = Auto
    Precio Costo = 15000.0
    Precio Venta = 21000.0
```

Como se puede apreciar, el precio de venta se calcula de manera automática cuando establecemos el precio de costo; esto debido a que utilizamos un método para asignar el precio de costo. Por otro lado, el precio de venta solo puede ser leído (solo lectura), pues no existe manera alguna de cambiar su valor, a menos que cambiemos el precio de costo.

8.7 SOBRECARGA

La sobrecarga permite definir dos o más métodos que comparten el mismo nombre, pero la declaración de sus parámetros debe ser diferente. Esta es una de las características más importante de la POO, y por lo tanto, también de Java.

Cuando se invoca un método sobrecargado, Java utiliza el tipo y/o número de parámetros como guía para determinar a qué versión del método debe llamar. Por lo tanto, los métodos sobrecargados deben ser diferentes en el tipo y/o número de parámetros.

- **Ejemplo 5**
  Observe el uso de la sobrecarga, para lo cual se implementa una clase que nos permite convertir a *String* un número; dependiendo el tipo de número se ejecutará el método que le corresponde.

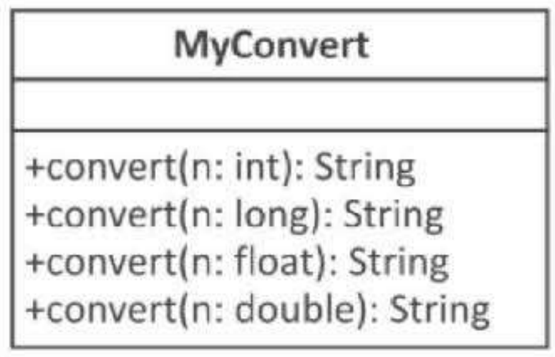

Fig. 8.8 Clase **MyConvert** con una función sobrecargada

La clase que se utilizará para esta demostración es **MyConvert** (ver Fig. 8.8) véase que el método **convert** está sobrecargado.

A continuación, se muestra la implementación de la clase **MyConvert**:

MyConvert.java

```
public class MyConvert{

    public String convert(int n) {
            return (Integer.toString(n));
    }

    public String conver(long n) {
            return (Long.toString(n));
    }

    public String convert(float n) {
            return (Float.toString(n));
    }

    public String convert(double n) {
            return (Double.toString(n));
    }

}
```

El programa que ilustra el uso de esta clase es el siguiente:

prog0805.java

```
public class prog0805
{
    public static void main(String[] args)
    {
            MyConvert obj = new MyConvert();
            int    n1 = 500;
            long   n2 = 100000;
            float  n3 = 456.459834f;
            double n4 = 8934.42573485720;

            System.out.println("n1 = " + obj.convert(n1));
            System.out.println("n2 = " + obj.convert(n2));
            System.out.println("n3 = " + obj.convert(n3));
            System.out.println("n4 = " + obj.convert(n4));
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap8>javac prog0805.java[Enter]

C:\java2n100\cap8>java prog0805      [Enter]
n1 = 500
n2 = 100000.0
n3 = 456.45984
n4 = 8934.4257348572
```

En este caso al invocar al método **Convert** con datos de diferentes tipos, internamente se ejecutará el método que corresponde según el tipo de dato del parámetro.

8.8 CONSTRUCTORES

Permiten inicializar un objeto inmediatamente después de su creación. Tienen el mismo nombre de la clase donde se encuentra, y sintácticamente son similares de los métodos.

Sintaxis

```
public nombre_clase( lista_de_parámetros ){

    // instrucciones

}
```

El constructor se invoca cuando creamos el objeto junto con el operador **new**.

Sintaxis

```
nombre_clase nombre_variable;

nombre_variable = new nombre_clase( lista_de_argumentos );
```

Los constructores tienen el mismo nombre de la clase, y no tienen tipo; esto debido a que retornan el mismo tipo de la clase. El objetivo principal de un constructor es inicializar el estado interno del objeto.

- **Ejemplo 6**
  Crearemos una clase de nombre **Empleado** que tiene un constructor que inicializa sus atributos.

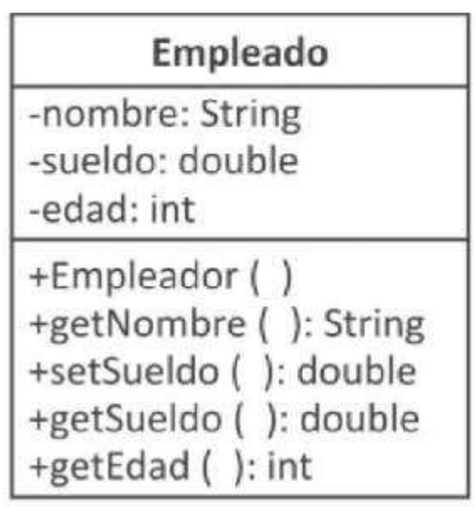

Fig. 8.9 Clase **Empleado** con un constructor

A continuación, tenemos la implementación de la clase **Empleado**. Aquí puede apreciar que el constructor tiene el mismo nombre que la clase.

Empleado.java

```
public class Empleado {

    // Atributos:

    private String nombre;
    private double sueldo;
    private int edad;

    // Constructor:

    public Empleado() {
            nombre = "Claudia";
            sueldo = 5000.00;
            edad = 22;
    }

    // Operaciones:

    public String getNombre() {
            return this.nombre;
    }
```

```
        public double getSueldo() {
                return this.sueldo;
        }

        public int getEdad() {
                return this.edad;
        }

    }
```

El programa que ilustra el uso de esta clase es el siguiente, puede notar que el constructor se invoca junto con el operador **new**.

prog0806.java

```
public class prog0806
{
        public static void main(String[] args)
        {
                Empleado obj = new Empleado( );

                System.out.println( "Nombre = " + obj.getNombre() );
                System.out.println( "Sueldo = " + obj.getSueldo() );
                System.out.println( "Edad   = " + obj.getEdad() );
        }
}
```

Compilación y ejecución:

```
C:\java2n100\cap8>javac prog0806.java[Enter]

C:\java2n100\cap8>java prog0806      [Enter]
Nombre = Claudia
Sueldo = 5000.0
Edad   = 22
```

8.8.1 Constructores con parámetros

Los constructores pueden resultar más útiles si tienen parámetros, en nuestro ejemplo de la clase **Empleado**, resulta que no todos los empleados se llaman Claudia. El Ejemplo 7 ilustra el uso de constructores con parámetros.

- **Ejemplo 7**
  Véase el uso de constructores con parámetros, la clase utilizada es **Empleado1**.

Empleado1
-nombre: String -sueldo: double -edad: int
+Empleador ( ) +getNombre ( ): String, sueldo: double, edad: int) +setSueldo ( ): String +getSueldo ( ): double +getEdad ( ): int

Fig. 8.10 Clase Empleado1 con un constructor con parámetros

A continuación, tenemos la implementación de la clase **Empleado1**.

Empleado1.java

```
public class Empleado1 {

    // Atributos:

    private String nombre;
    private double sueldo;
    private int edad;

    // Constructor:

    public Empleado1(String nombre, double sueldo, int edad) {
            this.nombre = nombre;
            this.sueldo = sueldo;
            this.edad = edad;
    }

    // Operaciones:

    public String getNombre() {
            return this.nombre;
    }

    public double getSueldo() {
            return this.sueldo;
    }

    public int getEdad() {
            return this.edad;
    }

}
```

El constructor de esta clase permite inicializar los objetos con diferentes datos, así podemos tener un objeto para cada empleado.

El programa que ilustra el uso de esta clase es el siguiente:

prog0807.java

```
public class prog0807
{
    public static void main(String[] args)
    {
            Empleado1 obj = new Empleado1( "Angélica", 4000, 23 );

            System.out.println( "Nombre = " + obj.getNombre() );
            System.out.println( "Sueldo = " + obj.getSueldo() );
            System.out.println( "Edad   = " + obj.getEdad() );
    }
}
```

Puede notar que el constructor se invoca junto con el operador **new** y es en esa instrucción que se le pasa los datos para que inicialice el objeto.

Compilación y ejecución:

```
C:\java2n100\cap8>javac prog0807.java[Enter]

C:\java2n100\cap8>java prog0807      [Enter]
Nombre = Angélica
Sueldo = 4000.0
Edad   = 23
```

Podemos concluir que cada vez que se crea un objeto, se puede inicializar con datos diferentes que correspondan a distintos empleados.

8.8.2 Sobrecarga de constructores

Al igual que los métodos, los constructores también pueden ser sobrecargados; esto decir que podemos tener diferentes constructores dependiendo de los datos de los que disponemos.

- **Ejemplo 8**
  Observe como construir clases con constructores sobrecargados, la clase que se utiliza es **Empleado2**, en esta clase tenemos cuatro versiones del constructor (ver Fig. 8.11).

Empleado2
-nombre: String -sueldo: double -edad: int -SUELDO BASE: double = 2000.0
+Empleado2 (nombre: String, sueldo: double, edad: int) +Empleado2 (nombre: String, sueldo: double) +Empleado2 (nombre: String, edad: int) +getNombre ( ): String +getSueldo( ): double +getEdad( ): int

Fig. 8.11 Clase Empleado2 con constructor sobrecargado

Los datos de un empleado son: nombre, sueldo y edad. Cuando falta el sueldo se asume un sueldo base que está definido en **SUELDO_BASE**, y cuando falta la edad se asume cero.

A continuación tenemos la implementación de la clase **Empleado2**:

empleado2.java

```
public class Empleado2 {

// Atributos:

    private String nombre;
    private double sueldo;
    private int edad;
    private static final double SUELDO_BASE = 5000.00;

// Constructores:

    public Empleado2(String nombre, double sueldo, int edad) {
            this.nombre = nombre;
            this.sueldo = sueldo;
            this.edad   = edad;
    }

    public Empleado2(String nombre, double sueldo) {
            this( nombre, sueldo, 0 );
    }

    public Empleado2(String nombre, int edad) {
            this( nombre, SUELDO_BASE, edad );
    }

    public Empleado2(String nombre) {
            this( nombre, SUELDO_BASE );
    }

// Operaciones:

    public String getNombre() {
            return nombre;
    }
```

```
    public double getSueldo() {
            return sueldo;
     }

     public int getEdad() {
            return edad;
     }

  }
```

El programa ilustrativo para el uso de esta clase es:

prog0808.java

```
  public class prog0808
  {
      public static void main(String[] args)
      {
              Empleado2 emp1 = new Empleado2( "Gustavo", 15000.0, 30 );
              Empleado2 emp2 = new Empleado2( "Ricardo", 27 );
              Empleado2 emp3 = new Empleado2( "Sergio" );

              System.out.println( "Empleado 1" );
              System.out.println( "Nombre = " + emp1.getNombre() );
              System.out.println( "Sueldo = " + emp1.getSueldo() );
              System.out.println( "Edad   = " + emp1.getEdad() );
              System.out.println();

              System.out.println( "Empleado 2" );
              System.out.println( "Nombre = " + emp2.getNombre() );
              System.out.println( "Sueldo = " + emp2.getSueldo() );
              System.out.println( "Edad   = " + emp2.getEdad() );
              System.out.println();

              System.out.println( "Empleado 3" );
              System.out.println( "Nombre = " + emp3.getNombre() );
              System.out.println( "Sueldo = " + emp3.getSueldo() );
              System.out.println( "Edad   = " + emp3.getEdad() );
              System.out.println();
      }
  }
```

Compilación y ejecución:

```
C:\java2n100\cap8>javac prog0808.java[Enter]

C:\java2n100\cap8>java prog0808      [Enter]
Empleado 1
Nombre = Gustavo
Sueldo = 15000.0
Edad   = 30
Empleado 2
Nombre = Ricardo
Sueldo = 5000.0
Edad   = 27
```

```
Empleado 3
Nombre = Sergio
Sueldo = 5000.0
Edad   = 0
```

Del resultado obtenido podemos deducir que a Ricardo se le asignó el sueldo base, debido que no se le especificó su sueldo, al momento de crear el objeto. En el caso de Sergio solo se especificó su nombre; por lo tanto se le asignó el sueldo base y edad cero.

Herencia

CAPÍTULO 9

La herencia es la característica más sobresaliente de la Programación Orientada a Objetos, porque permite definir una jerarquía de clases, y de esta manera, reutilizar sus definiciones.

Mediante la herencia, el programador puede crear una clase genérica que defina los rasgos comunes de un conjunto de objetos; luego esta clase puede ser heredada por otras clases más específicas, cada una de las cuales añadirá aquellos rasgos que la hacen única.

9.1 FUNDAMENTOS

Por sí misma, la idea de objeto es una herramienta más que buena. Permite empaquetar datos y funcionalidad; de manera que es posible representar cualquier idea del universo del problema mediante clases.

Es posible utilizar clases ya definidas, para crear a partir de estas, nuevas clases (clase heredada), y luego agregarle funcionalidad adicional según las necesidades de la nueva clase.

En terminología de la POO, una clase heredada se llama **superclase** y la clase que hereda se denomina **subclase**. Por lo tanto, una subclase es una versión especializada de la superclase, que hereda todas las definiciones de la superclase (atributos y métodos) y añade su propia funcionalidad.

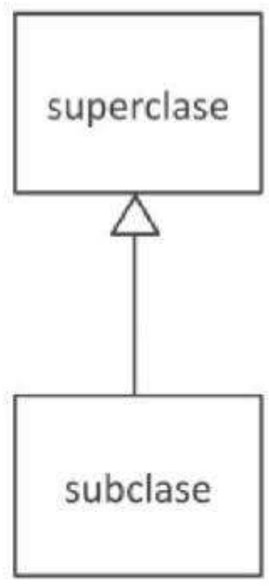

Fig. 9.1 Notación UML para representar la herencia

La Fig. 9.1 nos representa la herencia utilizando notación UML: la flecha apunta a la superclase y es posible que existan muchas subclases que se hereden de una superclase.

A la **superclase** también se le conoce como clase base, clase super o clase padre; y la subclase también es conocida como clase derivada, clase heredada, o clase hija.

- **Ejemplo 1**
  Veremos un caso básico de aplicación de herencia. Tenemos una clase base de nombre **CalculadoraBase**, y una subclase de nombre **Calculadora1** que hereda de **CalculadoraBase**.

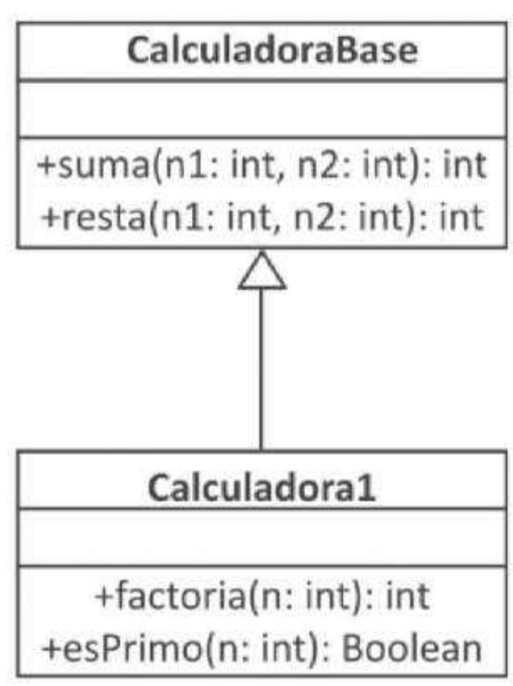

Fig. 9.2 Ejemplo básico de aplicación de herencia

En la Fig. 9.2 se puede apreciar que la subclase hereda los métodos suma y resta, e implementa los métodos factorial y **esPrimo**.

Una instancia de la clase **CalculadoraBase** solo tendrá los métodos suma y resta; mientras que una instancia de la clase **Calculadora1** tendrá los métodos heredados suma y resta, además de sus métodos propios factorial y **esPrimo**.

A continuación tenemos la implementación de la clase **CalculadoraBase**, donde se aprecia claramente la implementación de los métodos suma y resta.

CalculadoraBase.java

```
public class CalculadoraBase {

    public int suma(int n1, int n2) {
            return (n1+n2);
    }

    public int resta(int n1, int n2) {
            return (n1 - n2);
    }
}
```

Observe también la implementación de la clase **Calculadora1**; la palabra **extends** en la definición de la clase indica que está heredando de la clase **CalculadoraBase**, y que solo se están implementando los nuevos métodos.

Calculadora1.java

```
public class Calculadora1 extends CalculadoraBase {

    public int factorial(int n) {
            int f = 1;
            while(n>1){
                    f = f * n--;
            }
            return f;
    }

    public Boolean esPrimo(int n) {
            Boolean primo = true;
            int k = 1;
            while(++k<n){
                    if( n % k == 0 ){
                            primo = false;
                            break;
                    }
            }
            return primo;
    }
}
```

Véase además el uso de la clase **Calculadora1**, solo se está creando una instancia de esta clase (*cal*), y debido a la herencia, tenemos acceso a los métodos suma y resta, que son los métodos heredados, y los métodos propios factorial y **esPrimo**.

prog0901.java

```
public class prog0901
{
    public static void main(String[] args)
    {

            int n1 = 17, n2 = 6;
            Calculadora1 cal = new Calculadora1();

            System.out.println( "n1 = " + n1 );
            System.out.println( "n2 = " + n2 );
            System.out.println( "Suma: " + cal.suma(n1,n2) );
            System.out.println( "Resta: " + cal.resta(n1,n2) );
            System.out.println( "Factorial de " + n2 + " es " + cal.
factorial(n2) );
            System.out.println( n1 + " es primo: " + cal.esPrimo(n1) );

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap9>javac prog0901.java[Enter]

C:\java2n100\cap9>java prog0901      [Enter]
n1 = 17
n2 = 6
Suma: 23
Resta: 11
Factorial de 6 es 720
17 es primo: true
```

9.2 IMPLEMENTANDO LA HERENCIA

9.2.1 PALABRA CLAVE EXTENDS

La palabra clave **extends** permite implementar la herencia de clases. En Java solo se puede heredar desde una clase, en otras palabras no existe la herencia múltiple.

Sintaxis

```
class nombre_subclase extends nombre_superclase {

    // implementación

}
```

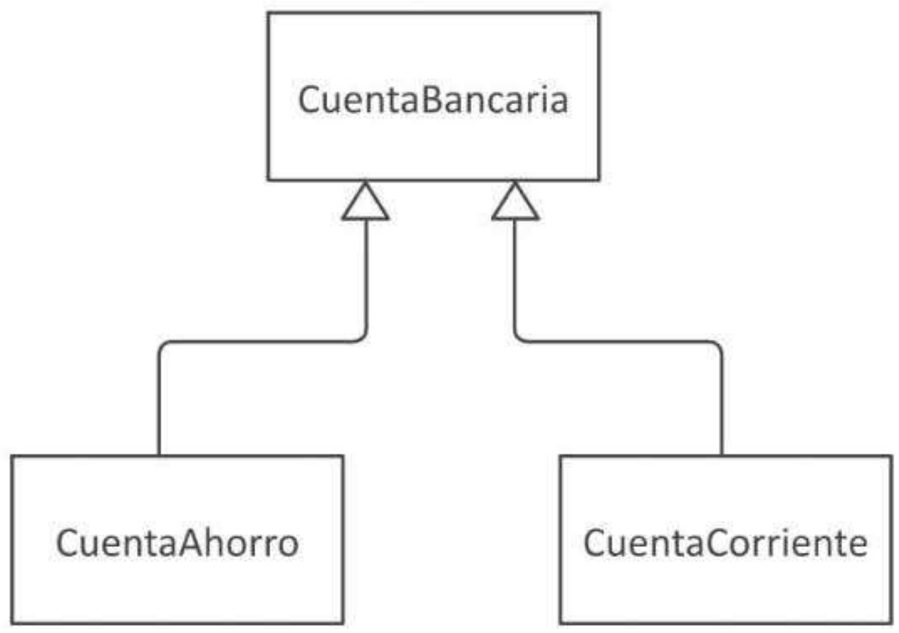

Fig. 9.3 Ejemplo de herencia

Por ejemplo, en la Fig. 9.3 tenemos una clase base **CuentaBancaria** que define las características comunes a todas las cuentas, y a partir de esta, se obtienen dos subclases **CuentaAhorro** y **CuentaCorriente**.

A continuación, tenemos el código básico para la implementación de estas clases:

```
public class CuentaBancaria {

    // Implementación

}

public class CuentaAhorro extends CuentaBancaria {

    // Implementación

}

public class CuentaCorriente extends CuentaBancaria {

    // Implementación

}
```

En este *script* básico se ilustra la implementación de la herencia.

9.2.2 Herencia de variables o campos

Sintaxis

```
[ public | protected | private] tipo nombre_variable [ = dato ] ;
```

Donde:

public	Un campo público (*public*) puede ser accedido desde la misma clase, cualquier subclase e instancia.
protected	Un campo protegido (*protected*) puede ser accedido desde la misma clase y las clases heredadas.
private	Un campo privado (*private*) solo puede ser accedido desde la misma clase.

Para que los campos puedan ser heredados deben ser públicos (*public*) o protegidos (*protected*). Los campos protegidos (*protected*) se comportan como públicos dentro de su propio paquete.

Cuando no se utiliza ninguna especificación de acceso, el campo es por defecto público (*public*) dentro de su propio paquete, pero su acceso no es posible desde otro paquete; es decir que es heredado por subclases dentro de su mismo paquete.

9.2.3 Herencia de métodos

Sintaxis

```
[ public | protected | private] tipo nombre_método ( parámetros ){
    // Implementación
}
```

Donde:

public	Un método público (*public*) puede ser accedido desde la misma clase, cualquier subclase e instancia.
protected	Un método protegido (*protected*) puede ser accedido desde la misma clase y las clases heredadas.
private	Un método privado (*private*) solo puede ser accedido desde la misma clase.

Para que los métodos puedan ser heredados deben ser públicos (*public*) o protegidos (*protected*). Los métodos protegidos se comportan como públicos dentro de su propio paquete.

Cuando no se utiliza ninguna especificación de acceso, el método es por defecto público (*public*) dentro de su propio paquete, pero su acceso no es posible desde otro paquete; esto quiere decir que es heredado por subclases dentro de su mismo paquete.

- **Ejemplo 2**
  Véase los diferentes tipos de acceso a los miembros de una clase.

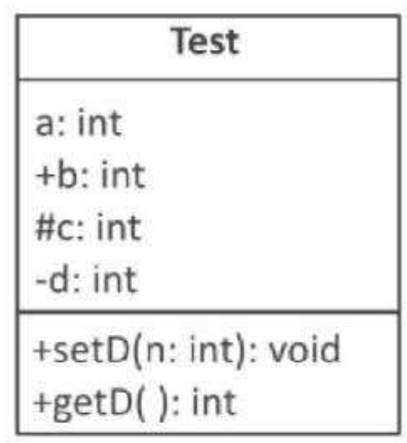

Fig. 9.4 Clase con diferentes tipos de acceso

Implementación de la clase **Test**.

Test.java

```
public class Test {

    int a;                          // Acceso por defecto
    public int b;           // Acceso público
    protected int c;        // Acceso protegido
    private int d;          // Acceso privado

    // Método para establecer el valor de c
    public void setD( int n ) {
            d = n;
    }

    // Método para leer el valor de c
    public int getD() {
            return d;
    }

}
```

Dentro de la clase **Test** tenemos definidas las siguientes variables:

int a;	Acceso por defecto (que en este caso es lo mismo que *public)*, porque se encuentran en la clase por defecto.
public int b;	Explícitamente está especificado como *public*.
protected int c;	Acceso protegido, es decir que en el mismo paquete actúa como público.
private int d;	Acceso privado, es decir que solo puede ser accedido desde métodos que se encuentren definidos dentro de la clase; por tal motivo se han creado los métodos **setD** y **getD**.

El programa ilustrativo para el uso de esta es:

prog0902.java

```
public class prog0902 {

    public static void main(String[] args) {
            Test obj = new Test();

            // a es público en el mismo paquete
            obj.a = 50;

            // b es público
            obj.b = 70;

            // c tiene comportamiento público en el mismo paquete
            obj.c = 80;

            // d es privado, su acceso es mediante métodos públicos
            obj.setD( 100 );

            // Salida
            System.out.println( "a = " + obj.a );
            System.out.println( "b = " + obj.b );
            System.out.println( "c = " + obj.c );
            System.out.println( "d = " + obj.getD() );
    }

}
```

Compilación y Ejecución:

```
C:\java2n100\cap9>javac prog0902.java[Enter]

C:\java2n100\cap9>java prog0902      [Enter]
a = 50
b = 70
c = 80
d = 100
```

9.3 PALABRAS CLAVES: THIS Y SUPER

9.3.1 Palabra clave this

Hace referencia al objeto actual, y puede utilizarse dentro de cualquier método. Por ejemplo, si tenemos la clase **Demo**:

```
class Demo{

        private int n;

        public void setN( int n ){
                this.n = n;
        }

  }
```

En esta clase tenemos una variable privada **n**, y un método **setN** para asignar un valor a la variable, pero resulta que el parámetro del método tiene el mismo nombre que la variable de la clase. Por lo tanto, la única manera de hacer referencia a la variable de la clase es mediante la palabra clave **this**, tal como se ilustra en el *script*.

La palabra clave **this** por si sola hace referencia al constructor de la misma clase.

- **Ejemplo 3**

Observe la aplicación de la palabra clave **this**, la clase que utilizaremos es **Empleado**.

Empleado
-codigo: int -nombre: String -sueldo: Double
+Empleado(codigo: int, nombre: String, sueldo: Double) +Empleado(nombre: String) +toString( ): String

Fig. 9.5 Clase Empleado

Implementación de la clase **Empleado**:

Empleado.java

```
public class Empleado {

// Atributos:

     private int código;
     private String nombre;
     private Double sueldo;

// Constructores:

     public Empleado(int código, String nombre, Double sueldo) {
```

```
    this.código = código;
                this.nombre = nombre;
                this.sueldo = sueldo;
        }

        public Empleado(String nombre) {
                this( 1000, nombre, 3500.00 );
        }

    // Operaciones

        public String toString() {
                return codigo + " " + nombre + " " + sueldo;
        }

    }
```

Esta clase tiene tres campos privados: código, nombre y sueldo; además dos constructores, el primero recibe tres datos (código, nombre y sueldo). Mientras que el segundo recibe solo el nombre y asume los otros dos datos, tal como se puede apreciar en el *script*, pero invoca al primer constructor mediante la palabra clave **this**. El otro método (**toString**) retorna una cadena con los tres datos del empleado.

Programa ilustrativo para el uso de la clase **Empleado**:

Prog0903.java

```
    public class prog0903 {

        public static void main(String[] args) {

                Empleado emp1 = new Empleado ( 500, "Claudia", 4000.00 );
                Empleado emp2 = new Empleado ( "Laura" );

                System.out.println( "Empleado 1: " + emp1.toString() );
                System.out.println( "Empleado 2: " + emp2.toString() );
        }

    }
```

Compilación y ejecución:

```
    C:\java2n100\cap9>javac prog0903.java[Enter]

    C:\java2n100\cap9>java prog0903      [Enter]
    Empleado 1: 500 Claudia 4000.0
    Empleado 2: 1000 Laura 3500.0
```

En el programa anterior, se observa que la segunda instancia (emp2) se crea solo con el nombre del empleado (Laura), pero internamente se le asigna código 1000 y sueldo 3500 00. Si revisamos la clase **Empleado**, esta se realiza a través del segundo constructor que invoca al primer constructor utilizando la palabra clase **this**.

9.3.2 Palabra clave super

Permite a una subclase hacer referencia a la superclase inmediata superior. Esta palabra clave tiene dos formas generales: la primera es para llamar al constructor de la superclase y la segunda es para acceder a un miembro de la superclase que se ha ocultado por un miembro de la subclase.

a. Invocar constructores de la superclase

Para invocar constructores de la superclase, se deben realizar desde el constructor de la subclase.

Sintaxis

```
super( lista_de_parámetros );
```

En este caso, **lista_de_parámetros** especifica cualquier parámetro que necesite el constructor en la superclase **.super** (**lista_de_parametros**) debe ser la primera sentencia que se ejecute dentro del constructor de la subclase.

b. Acceder a miembros de la superclase

La palabra clave **super** actúa de manera similar a **this**, pero en este caso hace referencia a la superclase.

Sintaxis

```
super.miembro
```

En este caso el miembro puede ser un método o una variable de instancia. Esta forma de usar la palabra clave super tiene mayor aplicación, sobre todo cuando miembros de la subclase ocultan miembros de igual nombre en la superclase.

- **Ejemplo 4**
  Véase la aplicación de la palabra clase **super** para invocar al constructor de la superclase.

  Se trata de una superclase **Persona** y dos subclases **Trabajador** y **Docente**; desde estas subclases se invocará al constructor de la superclase.

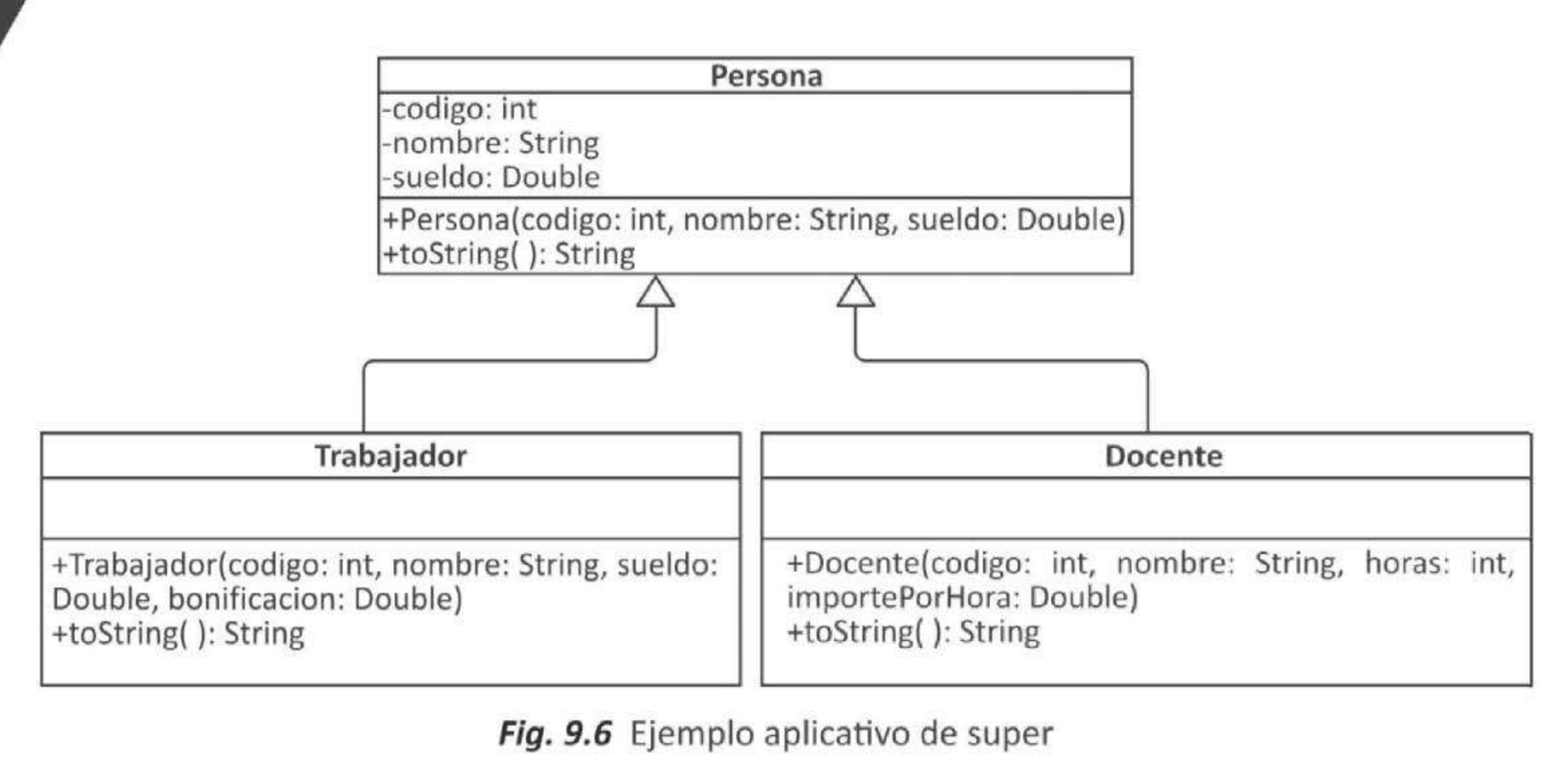

Fig. 9.6 Ejemplo aplicativo de super

Implementación de la clase **Persona**:

Persona.java

```
public class Persona {

// Atributos:

    private int código;
    private String nombre;
    private Double sueldo;

// Constructores:

    public Persona(int código, String nombre, Double sueldo) {
            this.código = código;
            this.nombre = nombre;
            this.sueldo = sueldo;
    }

// Operaciones:

    public String toString() {
            return ( código + " - " + nombre + " - " + sueldo );
    }
}
```

Esta clase tiene un constructor, a través del cual se asignan los datos, y un método (*toString*) que retorna los datos de la persona concatenados y separados por un guion.

A continuación, tenemos la implementación de la clase **Trabajador**:

Trabajador.java

```
public class Trabajador extends Persona {

// Operaciones:

     public Trabajador(int código, String nombre, Double sueldo, Double
bonificacion) {
             super( código, nombre, sueldo + bonificacion );
     }

     public String toString(){
             return ( "Trabajador: " + super.toString() );
     }
}
```

El constructor de esta clase invoca al constructor de la clase **Persona** mediante la palabra clave **super**. Se está pasando la suma del sueldo y la bonificación como el sueldo del trabajador. También se está redefiniendo el método **toString**, pero se está invocando al método **toString** de la clase **Persona** a través de la palabra clave **super**, y a su resultado se le está concatenando la palabra **Trabajador**.

A continuación tenemos la implementación de la clase **Docente**:

Docente.java

```
public class Docente extends Persona {

// Operaciones:

     public Docente(int código, String nombre, int horas, Double importe-
PorHora) {
             super( código, nombre, horas * importePorHora );
     }

     public String toString(){
             return ( "Docente: " + super.toString() );
     }
}
```

En el constructor de esta clase, también se está invocando al constructor de la clase **Persona**. El sueldo se está calculado como el producto de la cantidad de horas (horas) por el importe por hora (**importePorHora**).

De igual forma se está redefiniendo el método **toString** y se está accediendo al método **toString** de la clase **Persona**, a través de la palabra **super** y a su resultado se le está concatenando la palabra **Docente**.

A continuación tenemos el programa ilustrativo:

prog0904.java

```
public class prog0904 {

    public static void main(String[] args) {

            Trabajador tra = new Trabajador ( 500, "Julio Flores", 2000.00,
500.00 );
            Docente     doc = new Docente ( 501, "Gustavo Coronel", 150,
50.00 );

            System.out.println();
            System.out.println( tra.toString() + "\n" );
            System.out.println( doc.toString() );
    }

}
```

En este programa estamos creando dos instancias: **Trabajador** y **Empleado**; luego estamos imprimiendo la cadena que retorna el método **toString** de cada una de ellas respectivamente.

Compilación y ejecución:

```
C:\java2n100\cap9>javac prog0904.java [Enter]

C:\java2n100\cap9>java prog0904      [Enter]

Trabajador: 500 - Julio Flores - 2500.0

Docente: 501 - Gustavo Coronel - 7500.0
```

Según el resultado obtenido podemos concluir que se cumple con lo que se había previsto.

En el ejemplo se ha comprobado la herencia y la reutilización del código mediante la palabra clave **super**.

9.4 PALABRAS CLAVES: ABSTRACT, FINAL Y STATIC

9.4.1 Palabra clave abstract

La palabra clave **abstract** se utiliza para definir clases y métodos abstractos.

Una clase abstracta se utiliza para definir la interfaz de sus clases derivadas, es decir, es una clase base que no se podrá instanciar solamente se utiliza para ser heredada. Una clase abstracta debe por lo menos definir un método abstracto.

Los métodos abstractos no son implementados, estos deben ser necesariamente implementados en las clases derivadas (subclases).

a. Sintaxis: Clases abstractas

```
abstract class Nombre_Clase{

        // Implementación

}
```

Tal como se ilustra, la palabra clave **abstract** debe preceder a la palabra clave **class**.

b. Sintaxis: Métodos abstractos

```
abstract tipo Nombre_Método (lista_de_parámetros );
```

El método abstracto no se implementa, solo se define su interfaz, estos métodos se implementan en las clases derivadas. Observe el diagrama ilustrativo de una clase abstracta (ver Fig. 9.7).

Vehiculo es una clase abstracta, en ella se definen dos métodos **acelerar** y **frenar** que deben ser implementadas en las clases derivadas; esta clase no podrá ser instanciada.

La clase **Auto** es una clase derivada de la clase **Vehiculo**, aquí se debe implementar los métodos acelerar y frenar, pero además define e implementa las características propias de este tipo de clase; sí podrá ser instanciada.

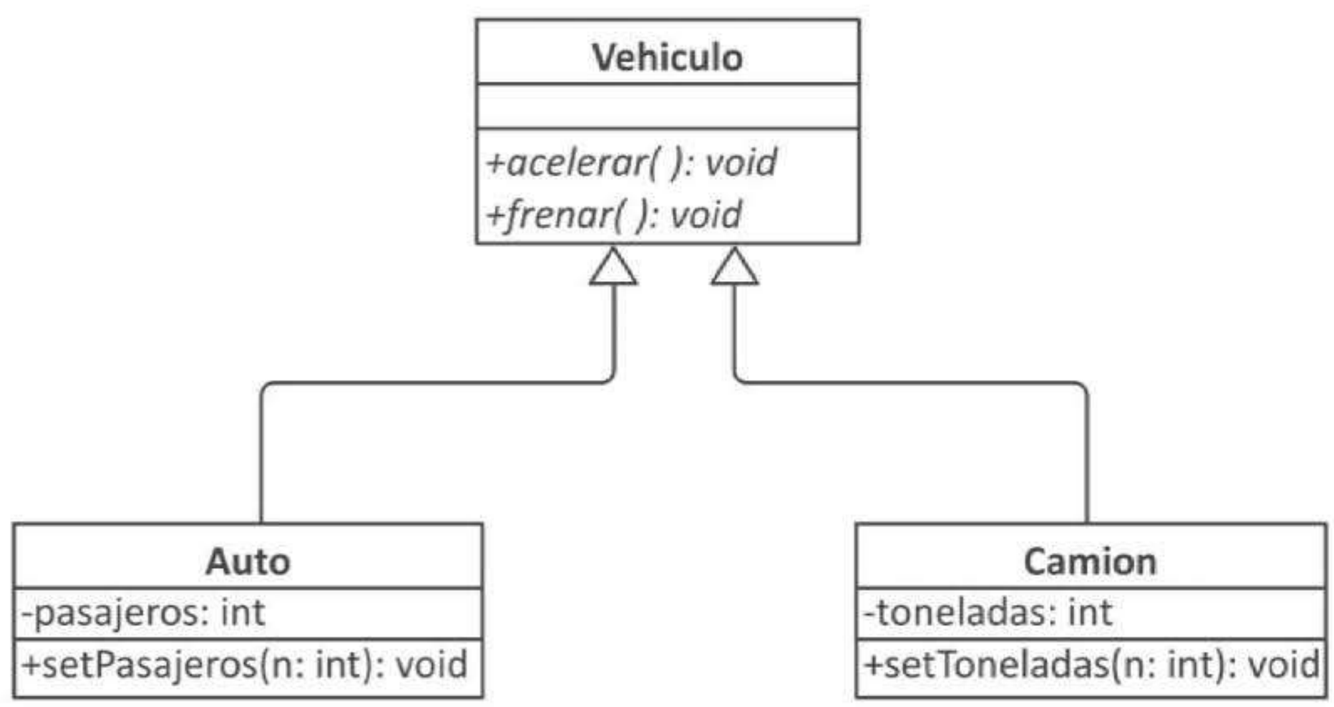

Fig. 9.7 Clase Abstracta

La clase **Camión** es una clase derivada de la clase **Vehiculo**, en esta clase se deben implementar los métodos **acelerar** y **frenar**, pero además define e implementa las características propias de este tipo de clase; sí podrá ser instanciada.

A continuación tenemos la implementación de la clase **Vehiculo**:

```
public abstract class Vehiculo {

// Operaciones:

    public abstract void acelerar();
    public abstract void frenar();
}
```

El prototipo de la clase **Auto** es el siguiente:

```
public class Auto extends Vehiculo {

// Atributos:

    private int pasajeros;

// Operaciones:

    public void acelerar() {

            // Implementación

    }

    public void frenar() {

            // Implementación

    }

    public void setPasajeros(int n) {

            // Implementación

    }
}
```

El prototipo de la clase **Camión** es el siguiente:

```
public class Camión extends Vehiculo {

// Atributos:

    private int toneladas;
```

```
// Operaciones:
    public void acelerar() {
            // Implementación
    }
    public void frenar() {
            // Implementación
    }
    public void setToneladas(int n) {
            // Implementación
    }
}
```

Como se puede apreciar en los prototipos, la implementación de los métodos abstractos se da en las clases derivadas o subclases; esto quiere decir que la siguiente instrucción en algún programa daría error:

```
Vehiculo obj = new Vehiculo();  // Esta instrucción es un error
```

Las clases que sí pueden ser instanciadas son: **Auto** y **Camión**.

9.4.2 Palabra clave final

La palabra clave **final** tiene varios usos: el primero es la definición de variables finales o constantes; la segunda la definición de métodos finales, y la última es la definición de clases finales.

a. Constantes

Sintaxis

```
final tipo nombre_variable = valor;
```

La definición de variables finales en realidad son constantes, es decir que deben ser inicializadas y no es posible modificar su valor. A continuación tenemos algunos ejemplos:

```
final int N1 = 100;
static final int N2 = 200;
public static final int N3 = 300;
protected final int N4 = 400;
private final int N5 = 500;
```

Una convención en programación es utilizar letras mayúsculas para definir variables finales o constantes.

b. Métodos finales

Normalmente es posible sobreescribir un método en una clase derivada o subclase. La definición de métodos finales proviene de la sobreescritura de los métodos en las clases derivadas o subclases; esto puede ser útil en muchos casos para evitar la sobreescritura involuntaria de métodos.

Sintaxis

```
final tipo Nombre_Método ( lista_parámetros ) {

    // Implementación

}
```

A continuación, tenemos un prototipo de un método final:

```
public final void reporte() {

    // Implementación

}
```

Los métodos declarados como finales, a veces, pueden proporcionar una mejora del rendimiento, porque el compilador ya sabe que no será sobreescrito por una subclase.

c. Clases finales

Normalmente una clase puede ser heredada, pero pueden haber casos donde no se produzca esta posibilidad; en estos casos las clases deben ser definidas como finales.

Sintaxis

```
final class Nombre_Clase {

    // Implementación

}
```

A continuación, tenemos un ejemplo ilustrativo:

```
public final class CuentaCorriente extends CuentaAhorro {

    // Implementación

}
```

Al definir una clase como final, automáticamente todos sus métodos son declarados como finales; por otro lado, una clase no puede definirse como abstracta y final al mismo tiempo.

- **Ejemplo 5**
  Véase la aplicación de clases **abstractas** y **finales**.

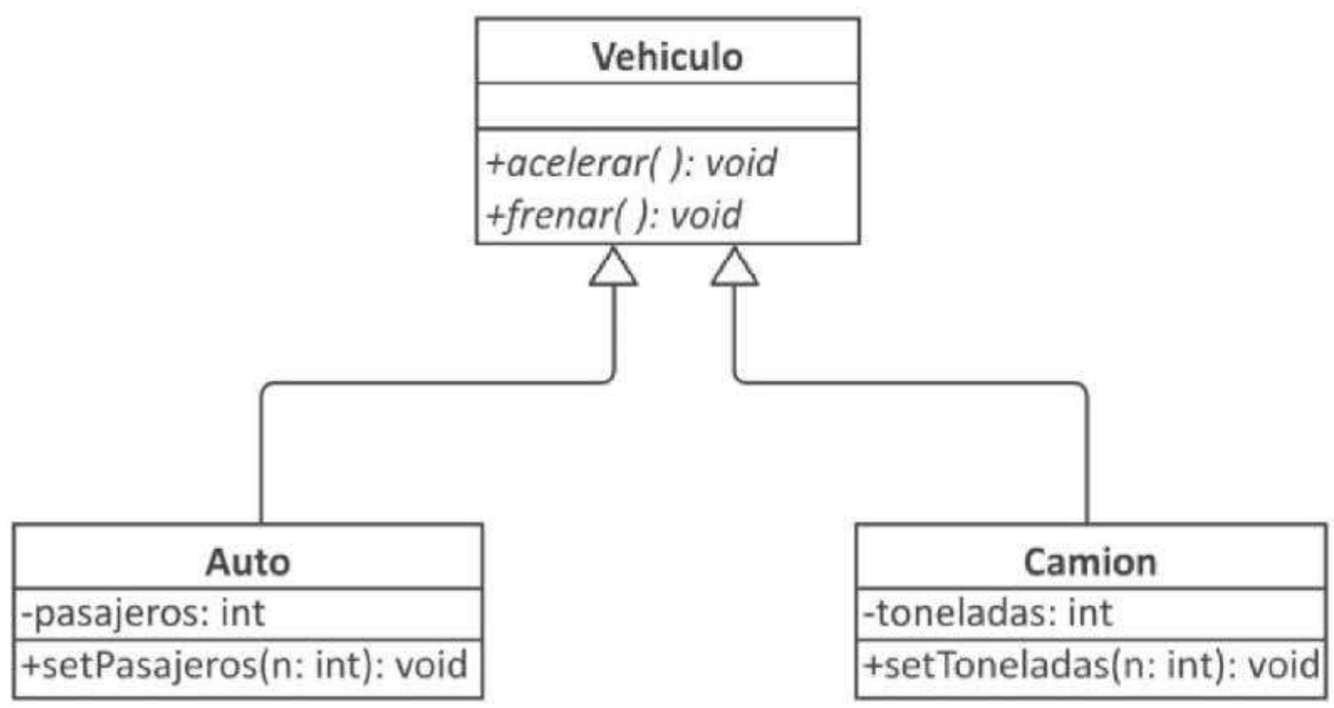

Fig. 9.8 Ejemplo de clase abstracta y final

Se han definido las siguientes clases:

Figura	Se trata de una clase abstracta que representa en forma genérica cualquier figura; en esta clase se define un método abstracto de nombre área, este método calcula el área de la figura y lo retorna con un dato double. Además debe ser implementado en cada clase derivada (subclase).
Cuadrado	Se trata de una clase Final que deriva de la clase Figura, es decir que esta clase no podrá ser heredada. Se debe implementar el método área.
Círculo	Se trata de una clase Final que deriva de la clase Figura, lo cual significa que no podrá ser heredada. En esta clase necesariamente se debe implementar el método área.

A continuación tenemos la implementación de la clase **Figura**:

Figura.java

```
public abstract class Figura {

// Atributos:

    protected int x;
    protected int y;

// Constructores:

    public Figura(int x, int y) {
            this.x = x;
            this.y = y;
    }


// Operaciones:

    public abstract double área();
}
```

Tal como se definió, se trata de una clase abstracta, con un método abstracto.

A continuación, observe la implementación de la clase **Cuadrado**:

Cuadrado.java

```
public final class Cuadrado extends Figura {

// Atributos:

    public int lado;

// Constructores:

    public Cuadrado(int x, int y, int lado) {
            super( x, y );
            this.lado = lado;
    }

// Operaciones:

    public double área(){
            return (lado * lado );
    }

}
```

Tal como se mencionó, la clase **Figura** es una clase **final**, y además implementa el método abstracto.

Véase la implementación de la clase **Círculo**:

Círculo.java

```
public final class Círculo extends Figura {

// Atributos:

    public int radio;

// Constructores:

    public Círculo(int x, int y, int radio) {
            super( x, y );
            this.radio = radio;
    }

// Operaciones:

    public double área() {
            return ( Math.PI * Math.pow(radio,2) );
    }

}
```

Como ya se mencionó, deriva de la clase **Figura**, se trata de una clase **final**, y además implementa el método abstracto.

El siguiente es un programa ilustrativo del uso de estas clases:

prog0905.java

```
public class prog0905 {

    public static void main(String[] args) {

            // Datos
            int lado = Integer.parseInt( args[0] );
            int radio = Integer.parseInt( args[1] );

            // Cuadrado
            Cuadrado obj1 = new Cuadrado( 1, 1, lado );
            System.out.println();
            System.out.println( "Cuadrado");
            System.out.println( "Lado: " + lado );
            System.out.println( "Área: " + obj1.área() );

            // Círculo
            Círculo obj2 = new Círculo( 1, 1, radio );
            System.out.println();
            System.out.println( "Círculo");
            System.out.println( "Radio: " + radio );
            System.out.println( "Área: " + obj2.área() );
    }

}
```

Este programa recibe dos parámetros; el primero es el lado del cuadrado, y el segundo el radio del círculo, luego se crean las respectivas instancias de cada clase, y se imprime su área.

```
        System.out.println( "Círculo");
                System.out.println( "Radio: " + radio );
                System.out.println( "Área: " + obj2.área() );
        }

    }
```

Compilación y ejecución:

```
C:\java2n100\cap9>javac prog0905.java[Enter]

C:\java2n100\cap9>java prog0905 5 10 [Enter]

Cuadrado
Lado: 5
Área: 25.0

Círculo
Radio: 10
Área: 314.1592653589793
```

A continuación tenemos algunos posibles errores que se podrían cometer en la utilización de estas clases.

Error 1: No se puede instanciar una clase abstracta.

```
class error1{

    Figura obj = new Figura( 1, 1 );   // Error

    . . .

}
```

Error 2: No se puede heredar una clase final.

```
class error2 extends Cuadrado {  // Error

    . . .

}
```

9.4.3 Palabra clave static

La palabra clave **static** se utiliza para definir miembros estáticos, que pueden ser variables o campos y métodos; también se le llama miembros de clase. La particularidad de estos miembros estáticos es que se pueden acceder a ellos sin necesidad de instanciar la clase, por ejemplo tenemos el método **pow** de la clase **Math** que utilizamos en la clase **Círculo** para calcular el área del círculo (ver Ejemplo 5). La instrucción es la siguiente:

```
return ( Math.PI * Math.pow(radio,2) );
```

Como se observa se está accediendo al método **pow** y no es necesario crear una instancia de la clase **Math**.

a. Variables estáticas

Sintaxis:

```
static tipo nombre_variable [ = valor ];
```

Se recomienda inicializar las variables estáticas.

Las variables estáticas son las formas controladas que implementa Java para tener variables globales. Cuando crean instancias de clase no se crean copias de las variables estáticas, en lugar de ello, todas las instancias comparten las mismas variables estáticas.

b. Métodos estáticos

Sintaxis:

```
static tipo nombre_método( lista_parámetros ) {

    // Implementación

}
```

Los métodos estáticos tienen las siguientes restricciones:

- Solo pueden invocar a otros métodos estáticos.
- Solo pueden acceder a variables declaradas como estáticas.
- No pueden hacer referencia a las palabras claves **this** y **super**.

Un caso especial es el método **main**, que es un método estático porque debe ser invocado antes de que exista el objeto. También es posible tener un bloque **static** que se ejecuta en una sola ocasión cuando se carga la clase por vez primera; este bloque puede servir para inicializar las variables estáticas de la clase.

- **Ejemplo 6**
  Se ilustra el uso de la palabra clave **static** para declarar variables estáticas, un bloque **static**, y métodos estáticos.

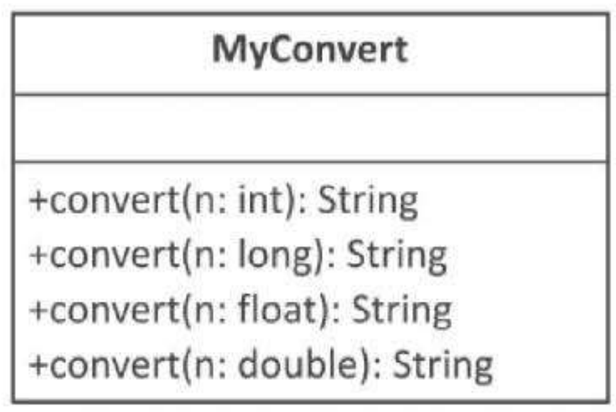

Fig. 9.9 Miembros estáticos de una clase

A continuación, observe la implementación del programa:

prog0906.java

```
public class prog0906 {

// Atributos:

    public static int a = 15;
    public static int b;

// Operaciones:

    static{
            System.out.println( "Mensaje desde el bloque static" );
            b = a * a;
    }

    public static void main( String args[] ) {
            System.out.println( "Mensaje de main" );
            print();
    }

    public static void print() {
            System.out.println( "Mensaje desde print" );
            System.out.println( "a = " + a );
            System.out.println( "b = " + b );
    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap9>javac prog0906.java[Enter]

C:\java2n100\cap9>java prog0906      [Enter]
Mensaje desde el bloque static
Mensaje de main
Mensaje desde print
a = 15
b = 225
```

Todas las instrucciones estáticas se cargan en cuanto se carga la clase. Se puede deducir del resultado que se obtiene que la ejecución se da en el siguiente orden:

1. Se definen las variables, y se inicializa la variable en 15.
2. Se ejecuta el bloque **static**.
3. Se ejecuta el método **main**.
4. Se ejecuta el método **print**, porque es invocado desde el método **main**.

Paquetes y clases

CAPÍTULO 10

Los paquetes son contenedores de clases que se utilizan para organizar las clases de una aplicación y así prevenir la colisión de clases del mismo nombre.

10.1 PAQUETES

En los ejemplos del capítulo anterior, el nombre de cada clase se toma del mismo espacio del nombre. Esto quiere decir que para evitar colisiones cada clase debe tener su propio nombre diferente de las demás clases.

Por ejemplo para evitar la colisión de la clase **Calculadora** creamos la clase **Calculadora2**, lo mismo sucedió con la clase **Artículo** y **Empleado**.

Los paquetes son los mecanismos que proporciona Java para agrupar clases relacionadas en espacios de nombres más manejables y controlar también su visibilidad.

10.1.1 Definición de paquete

Para crear un paquete debemos incluir la cláusula *package* como primera sentencia del archivo fuente de Java; cualquier clase que se declare dentro del archivo pertenecerá al paquete.

Sintaxis

```
package nombre_paquete;
```

La sentencia **package** define el espacio de nombre donde se almacenan la clases. Si omitimos esta sentencia, las clases se almacenan en un paquete que no tiene nombre por defecto (por ese motivo en los capítulos anteriores no nos hemos preocupado por los paquetes). El paquete por defecto es adecuado para pequeños ejemplos, pero no lo es en aplicaciones reales donde muchas veces participan varios programadores.

En el siguiente ejemplo:

```
package librería;

public class Test{

    . . .
    . . .

}
```

La clase **Test** pertenece al paquete **librería**.

Java utiliza directorios del sistema para almacenar los paquetes. Así tenemos que la clase que se declare como parte del paquete **librería** debe almacenarse en un directorio de nombre **librería**.

Recuerde que Java distingue entre mayúsculas y minúsculas, y que el nombre del directorio debe coincidir exactamente con el nombre del paquete.

Se puede crear una jerarquía de paquetes, para ello, debe separarse con un punto el nombre de cada paquete del inmediato superior.

Sintaxis

```
package paquete1[.paquete2[.paquete3[...]]];
```

La jerarquía de paquetes se refleja en la jerarquía de directorios del sistema. Por ejemplo, si declaramos el siguiente paquete:

```
package librería.almacén.venta;
```

Se debe almacenar en **librería/almacén/venta**, **librería\almacén\venta**, o **librería: almacén: venta**, según se trate de UNIX, Windows o Macintosh, respectivamente.

Los nombres de los paquetes se deben elegir cuidadosamente, porque si queremos cambiar después el nombre, se deberá cambiar también el directorio y la sentencia **import** en las clases que hacen referencia a dichos paquetes.

- **Ejemplo 1**
  Observe el uso de un paquete, en este caso crearemos la clase **MyMath** dentro de paquete **útil1**. La definición de la clase se ilustra en la Fig. 10.1.

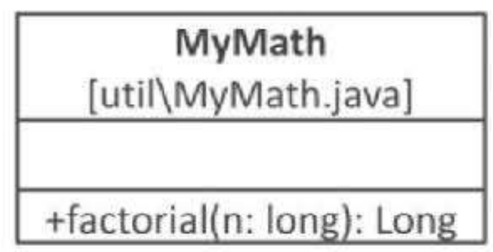

Fig. 10.1 Clase MyMath dentro del paquete útil1

Indicamos a continuación la implementación de la clase **MyMath**:

útil1\MyMath.java

```
package útil1;

public class MyMath {

    public Long factorial(long n) {
            long f = 1;
            while(n>1){
                    f = f * n--;
            }
            return f;
    }
}
```

Es importante recordar que esta clase se debe grabar en el directorio **útil1**; nuestro directorio raíz en este caso es **C:\java2n100\cap10**.

El programa que ilustra el uso de esta clase es el siguiente:

prog1001.java

```
import útil1.MyMath;

public class  prog1001{

    public static void main(String[] args){
            int n = Integer.parseInt( args[0] );
            MyMath obj = new MyMath();
            System.out.println( "N = " + n );
            System.out.println( "Factorial = " + obj.factorial(n) );
    }

}
```

Este programa se encuentra en el paquete por defecto, por tal motivo no necesita de la cláusula **package**, pero sí debe importar la clase **MyMath** para instanciarla; por lo tanto debe utilizarse la instrucción **import**.

Compilación y ejecución:

```
package librería;

public class Test{

    . . .
    . . .

}
```

10.2 PAQUETES Y CLASSPATH

Los paquetes se reflejan en los directorios dando lugar a una importante cuestión:

¿Cómo sabe el intérprete de Java dónde buscar los paquetes?

La respuesta tiene dos partes:

1. En primer lugar, el intérprete utiliza por defecto el directorio de trabajo como su punto de salida, por tanto lo encontrará si el paquete está en el directorio o subdirectorio de este.
2. En segundo lugar, se puede especificar una o más rutas en la variable de entorno **CLASSPATH**.

Por ejemplo, veamos el siguiente paquete:

```
package tool;

public class demo{

    // Implementación

}
```

Para que los programas encuentren el paquete **tool**, debe cumplirse una de estas dos condiciones:

- Se ejecuta desde un directorio inmediatamente superior de **tool** o;
- Se incluye en la variable de entorno **CLASSPATH** la ruta donde se encuentra el directorio **tool**.

La primera opción es la más sencilla, porque no necesita un cambio en **CLASSPATH**; sin embargo la segunda permite que el programa encuentre **tool** independientemente de cual sea el directorio donde se encuentre el programa almacenado. La elección depende de usted.

10.3 IMPORTACIÓN DE PAQUETES

Puesto que los paquetes constituyen un buen mecanismo para la división de unas clases procedentes de otras, todas las clases incorporadas en Java están almacenadas en paquetes. No hay clases pertenecientes al núcleo de Java en el paquete sin nombre utilizado por defecto, esto significa que todas las clases estándares están almacenadas en paquetes con nombres.

Dado que las clases contenidas en un paquete se deben calificar en el nombre del paquete, puede resultar una tarea pesada tener que escribir el nombre para cada clase que se quiera utilizar. Por ese motivo, Java incluye la instrucción **import**, la cual permite acceder a ciertas clases o paquetes enteros. Una vez que se importa una clase, se puede hacer referencia a ella usando solo su nombre.

En un archivo fuente de Java, las instrucciones **import** deben darse inmediatamente después de la instrucción **package**, en caso exista, y antes de la definición de cualquier clase.

Sintaxis

```
import paquete1[.paquete2[...]].{nombre_clase|*};
```

Donde **paquete1** es el nombre del paquete de nivel superior, y **paquete2** es el nombre de un paquete inferior dentro del paquete exterior y separado por un punto. En la práctica, no existe un límite para la profundidad de un jerarquía de paquetes, excepto el que imponga el Sistema Operativo. Finalmente se puede especificar una determinada clase a través de su nombre o un asterisco (*), lo cual indicará que el compilador de Java debe importar el paquete completo.

A continuación véase algunos ejemplos del uso de la instrucción **import**:

```
import librería.almacén.Artículo;
import librería.contabilidad.*;
```

El uso del asterisco puede incrementar el tiempo de compilación, sobre todo si se importan varios paquetes grandes; por ello, es una buena práctica especificar la clase que se desea importar, en lugar de importar un paquete completo. Sin embargo la opción asterisco no tiene ningún efecto sobre el tiempo de ejecución o tamaño de las clases.

Todas las clases estándares de Java están almacenadas en un paquete llamado **java**. Las funciones que constituyen el lenguaje básico se almacenan en un paquete dentro del paquete **java** llamado **java.lang**. Normalmente, se debe importar cada paquete o clase que se quiere utilizar, pero el paquete **java.lang** se importa implícitamente mediante el compilador para todos los programas. Por ese motivo no es necesario realizar la importación de forma explícita, lo que equivale a que la siguiente instrucción esté al inicio de todos los programas:

```
import java.lang.*;
```

Si una clase existe con el mismo nombre en dos paquetes distintos que se importan utilizando la opción asterisco, el compilador no dará ningún error a menos que se intente utilizar una de las clases. En este caso, se obtendrá un error en tiempo de compilación y será necesario poner explícitamente el nombre de la clase especificando su paquete.

En cualquier ubicación en que se utilice el nombre de una clase, se puede poner el nombre completo, que incluye su jerarquía (de paquetes) completa.

El siguiente ejemplo utiliza la instrucción **import**:

```
import java.útil.*;

public class MyDate extends Date{

    // Implementación

}
```

El mismo ejemplo puede ser reescrito de la siguiente manera, sin la instrucción de **import**:

```
public class MyDate extends java.util.Date{

    // Implementación

}
```

Cuando se importa un paquete, solo aquellos elementos declarados como **public** estarán disponibles para las clases que no sean subclases del código importado.

- **Ejemplo 2**
  Ilustraremos el uso de paquetes y comprobaremos el acceso a los métodos de una clase.

  Tenemos un paquete de nombre **útil2**, en él tenemos dos clases (ver Fig. 10.2).

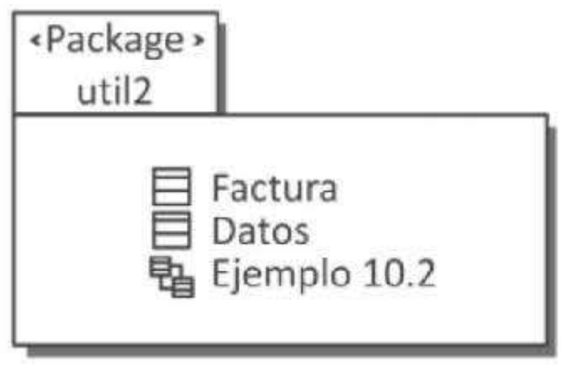

Fig. 10.2 Diagrama del paquete útil2

La clase **Datos** se utiliza como un nuevo tipo de dato, el método **procesa** de la clase **Factura** y retorna la referencia a una instancia de la clase **Datos**.

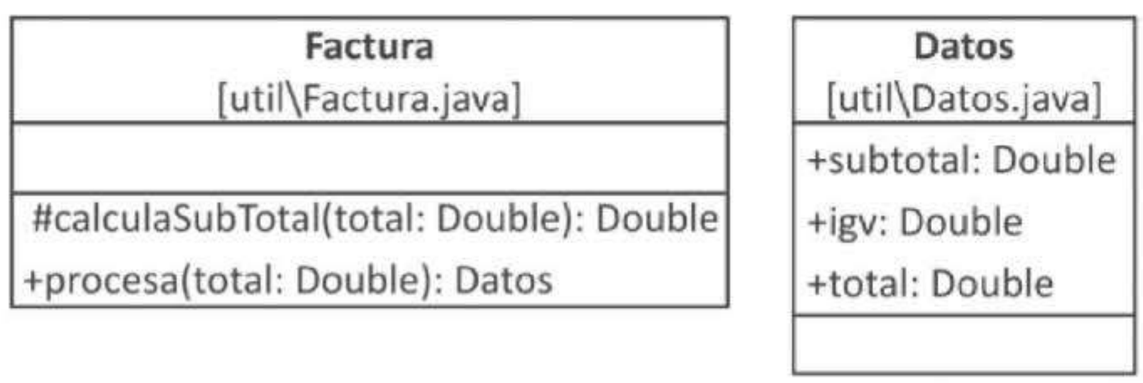

Fig. 10.3 Clases dentro del paquete útil2

El método **procesa** recibe como dato el total de una factura, calcula el subtotal, el IGV y estos tres datos retornan a una instancia de la clase **Datos**.

A continuación tenemos la implementación de la clase **Datos**:

util2\Datos.java

```
package útil2;

public class Datos {

// Atributos:

      public Double subtotal;
      public Double IGV;
      public Double total;

  }
```

Esta clase solo tiene variables públicas, en este caso es similar a un registro o estructura en otros lenguajes.

A continuación, indicamos la implementación de la clase **Factura**:

útil2\Factura.java

```
package útil2;

public class Factura {

     // Operations:

     protected Double calculaSubTotal(Double total) {
             return ( total / 1.19 );
     }

     public Datos procesa(Double total) {
             Datos obj = new Datos();
             obj.total = total;
             obj.subtotal = calculaSubTotal( total );
             obj.IGV = total - obj.subtotal;
             return obj;
     }

}
```

El método **calculaSubTotal** es protegido, por lo tanto no es visible desde programas que se encuentren fuera de este paquete.

Observe el siguiente programa ilustrativo:

prog1002.java

```
import útil2.*;

public class prog1002{

     public static void main(String[] args) {

             Double total = Double.parseDouble( args[0] );
             Datos obj;
             Factura fact = new Factura();

             obj = fact.procesa( total );

             System.out.println( "Subtotal: " + obj.subtotal );
             System.out.println( "IGV: " + obj.igv );
             System.out.println( "Total: " + obj.total );

     }

}
```

Compilación y ejecución:

```
C:\java2n100\cap10>javac prog1002.java        [Enter]

C:\java2n100\cap10>java prog1002 119          [Enter]
Subtotal: 100.0
IGV: 19.0
Total: 119.0
```

Si intentáramos acceder, en este programa, al método **calculaSubTotal** tendríamos un error de compilación.

10.4 PROTECCIÓN DE ACCESO

El uso de paquetes añade otra dimensión de control de acceso a las clases y sus miembros. Como señalamos, Java proporciona muchos niveles de protección para permitir un control adecuado de la visibilidad de las variables y métodos dentro de las clases, subclases y paquetes.

Las clases y los paquetes son medios de encapsulado y contienen el espacio de nombres y campos de acción de las variables y los métodos. Los paquetes actúan como contenedores de clases y otros paquetes subordinados. Las clases actúan como contenedoras de datos y códigos. La clase es la mínima unidad de abstracción en Java. Dada la interacción entre clases y paquetes, Java establece cuatro categorías de visibilidad para los miembros de clase:

- Subclases del mismo paquete.
- No subclases del mismo paquete.
- Subclases de diferentes paquetes.
- Clases que no están en el mismo paquete ni en las subclases.

Los especificadores de acceso: *private*, *protected* y *public*, proporcionan una gran variedad de formas de producir los niveles de acceso; a estos tenemos que agregar un cuarto, *package* (cuando no se especifica ninguno), tal como se resume en la Tabla 10.1.

Aunque el mecanismo de control de acceso de Java parece complicado, podemos simplificarlo a continuación:

1. Se puede acceder a cualquier elemento declarado como *public* desde cualquier parte del programa.
2. No se puede acceder a un elemento declarado como *private* desde fuera de la clase.
3. Cuando un miembro no tiene explícitamente un especificador de acceso, es visible para las subclases y para otras clases del mismo paquete. En esto consiste el acceso por defecto.

4. Si se requiere que un elemento sea visible desde fuera de su paquete actual, pero solamente por clases derivadas (subclases); entonces debe ser declarado como *protected*.

Tabla 10.1 *Acceso a los miembros de una clase*

Ámbito ==>	Sin modificador (package)	private	protected	public
Misma clase	Sí	Sí	Sí	Sí
Subclase del mismo paquete	Sí	No	Sí	Sí
No subclase del mismo paquete	Sí	No	Sí	Sí
Subclases de paquetes diferentes	No	No	Sí	Sí
No subclases de paquetes diferentes	No	No	No	Sí

La Tabla 10.1 se aplica, únicamente, a los miembros de las clases. Una clase tiene solamente dos niveles de acceso: por defecto y público. Cuando se declara una clase como *public*, resulta accesible desde cualquier parte; pero si tiene acceso por defecto solo es accesible desde el mismo paquete donde se encuentra.

- Ejemplo 3
  Observe el paquete de nombre **útil3**, en el tenemos dos clases: **TipoCambio** y **Aplica**.

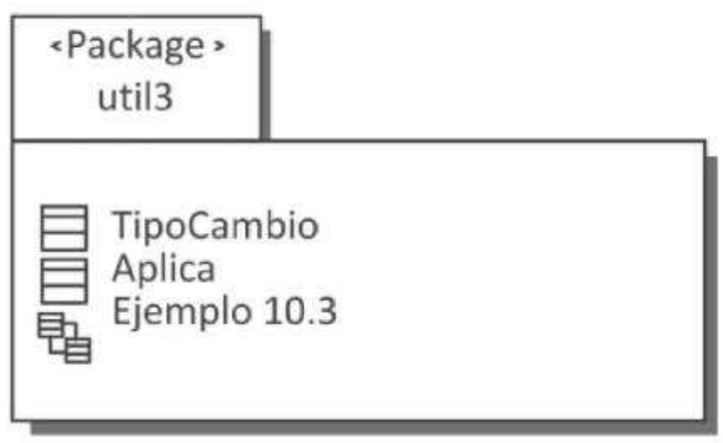

Fig. 10.4 Paquete útil3

– **TipoCambio**. Es una clase que permite pasar Nuevos Soles a Dólares, a través del método **solesToDolares**; y Dólares a Nuevos Soles mediante el método **dolaresToSoles**.
– **Aplica.** Es una clase donde se ilustra la aplicación de la clase **TipoCambio**.

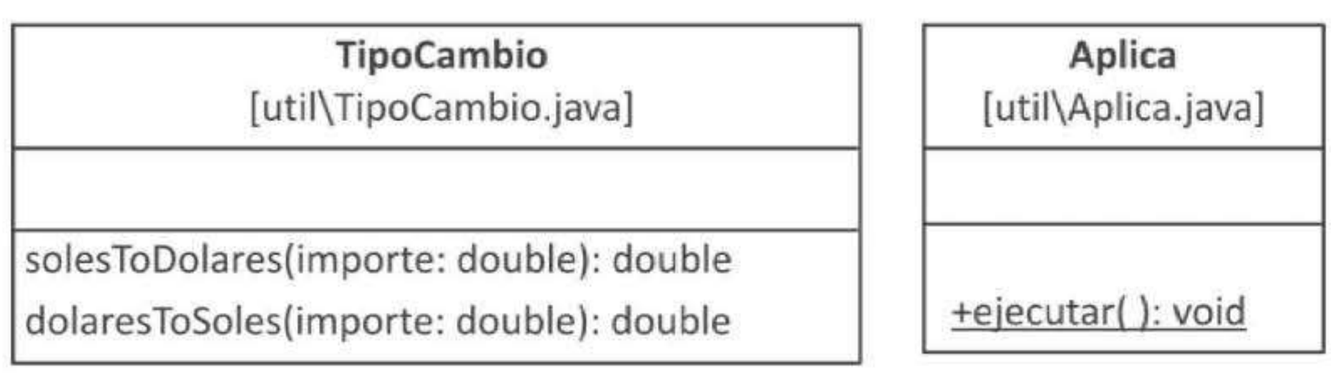

Fig. 10.5 Clases en el paquete útil3

Véase la implementación de la clase **TipoCambio**:

```
útil3\TipoCambio.java

package útil3;

class TipoCambio {

    double solesToDólares(double importe) {
            return ( importe / 3.50 );
    }

    double dólaresToSoles(double importe) {
            return ( importe * 3.50 );
    }

}
```

Como puede apreciarse, el alcance de esta clase se produce en el mismo paquete, por lo tanto solo puede ser utilizada en el mismo paquete.

A continuación tenemos la implementación de la clase **Aplica**:

útil3\Aplica.java

```
package útil3;

public class Aplica{

    public static void ejecutar(){

            double soles = 5000;
            double dólares = 5000;
            TipoCambio obj = new TipoCambio();

            System.out.println( soles + " Soles = " + obj.solesToDólares(soles)
+ " Dólares" );
            System.out.println( dólares + " Dólares = " + obj.
dólaresToSoles(dólares) + " Soles" );

    }

}
```

Esta clase **Aplica** es pública y tiene un método estático, es decir que para ejecutar este método no es necesario instanciar la clase; además se puede ejecutar desde cualquier parte porque es pública.

Observe el programa que ilustra el uso de estas clases:

prog1003.java

```
import útil3.*;

class prog1003
{
    public static void main(String[] args)
    {
            Aplica.ejecutar();
    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap10>javac prog1003.java        [Enter]

C:\java2n100\cap10>java prog1003              [Enter]
5000.0 Soles = 1428.5714285714287 Dólares
5000.0 Dólares = 17500.0 Soles
```

En este programa no es posible instanciar la clase **TipoCambio** porque su alcance no es público, solo a nivel de paquete.

Si quisiéramos ejecutar el programa **prog1003.java** desde cualquier ubicación, tendríamos que establecer la variable de entorno **CLASSPATH**, como se ilustra a continuación.

1. Tratando de compilar el programa **prog1003.java** desde la raíz de C:, se reporta un error porque no encuentra el paquete **útil3**.

```
C:\>javac prog1003.java                [Enter]
prog1003.java:1: package útil3 does not exist
import útil3.*;
^
prog1003.java:7: cannot find symbol
symbol  : variable Aplica
location: class prog1003
                Aplica.ejecutar();
                ^
2 errors
```

2. Para establecer la variable de entorno **CLASSPATH** debemos tener en cuenta el valor que actualmente tiene. La instrucción es la siguiente:

```
C:\>set CLASSPATH=%CLASSPATH%;c:\java2n100\cap10;     [Enter]
```

3. Ahora estamos en condiciones de compilar y ejecutar el programa **prog1003.java** desde cualquier ubicación.

```
C:\>javac prog1003.java                    [Enter]
prog1003.java:1: package útil3 does not exist
import útil3.*;
^
prog1003.java:7: cannot find symbol
symbol  : variable Aplica
location: class prog1003
                    Aplica.ejecutar();
                    ^
2 errors
```

Clases e interfaces

CAPÍTULO 11

Un interfaz es una lista de acciones que puede llevar a cabo un determinado objeto. La diferencia con una clase, es que en una interfaz solo tenemos el prototipo del método, mientras que en una clase normalmente se tiene también la implementación.

Las interfaces son un nivel superior de abstracción que luego deben ser implementadas en clases.

11.1 ¿QUÉ ES UNA INTERFAZ?

Un interfaz es la definición de un conjunto de operaciones que caracteriza el comportamiento de un objeto y debe ser desarrollada, en las clases que la implementan; esto quiere decir que las interfaces solo contienen los prototipos de los métodos, mas no la implementación.

Una clase que implementa una interfaz debe implementar todos los métodos declarados por esta. Muchas clases pueden implementar la misma interfaz y no necesitan compartir la misma clase heredada. Además, una clase puede implementar más de una interfaz, la cual debe declarar constantes públicas, estáticas y finales.

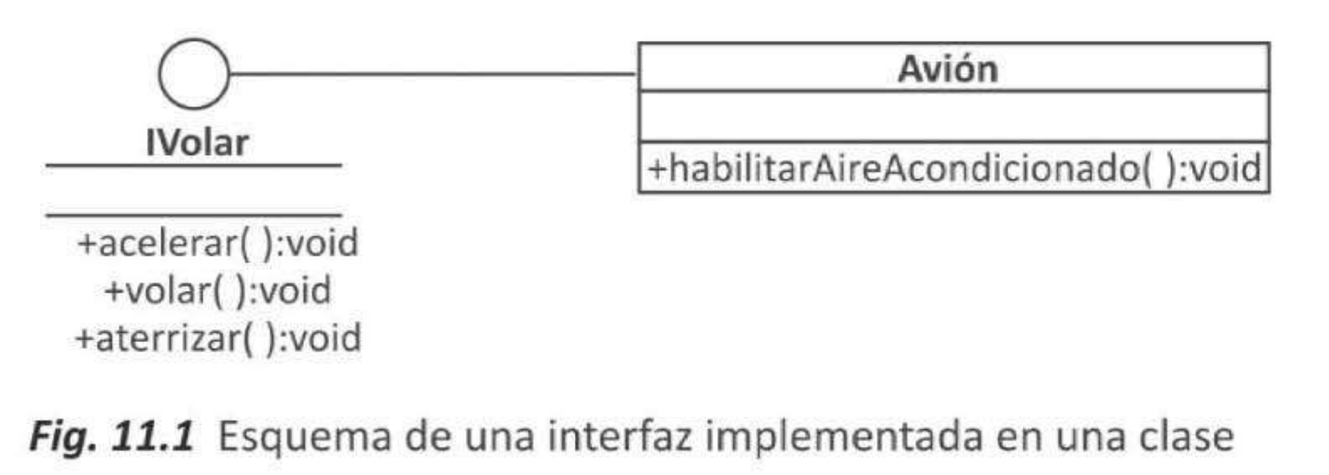

Fig. 11.1 Esquema de una interfaz implementada en una clase

En la Fig. 11.1 tenemos una interfaz **IVolar** que es implementada por la clase **Avión**. Note que la clase puede implementar métodos propios; en este caso el método **habilitarAireAcondicionado**. Esta interfaz puede ser implementada por muchas clases.

Para declarar una interfaz se utiliza la palabra clave **interface**, tal como señalamos a continuación:

```
public interface IVolar {

    public void acelerar();
    public void volar();
    public void aterrizar();

}
```

Para implementar una interfaz se debe utilizar la palabra clave **implements**, como se indica en el siguiente ejemplo:

```
public class Avión implements IVolar{

    public void acelerar(){
            // Implementación
    }

    public void volar(){
            // Implementación
    }

    public void aterrizar(){
            // Implementación
    }

    public void habilitarAireAcondicionado() {
            // Implementación
    }

}
```

Podemos tener múltiples clases que implementan la interfaz **IVolar**. Un avión, un ave pueden volar, «Superman» también puede volar, etc.

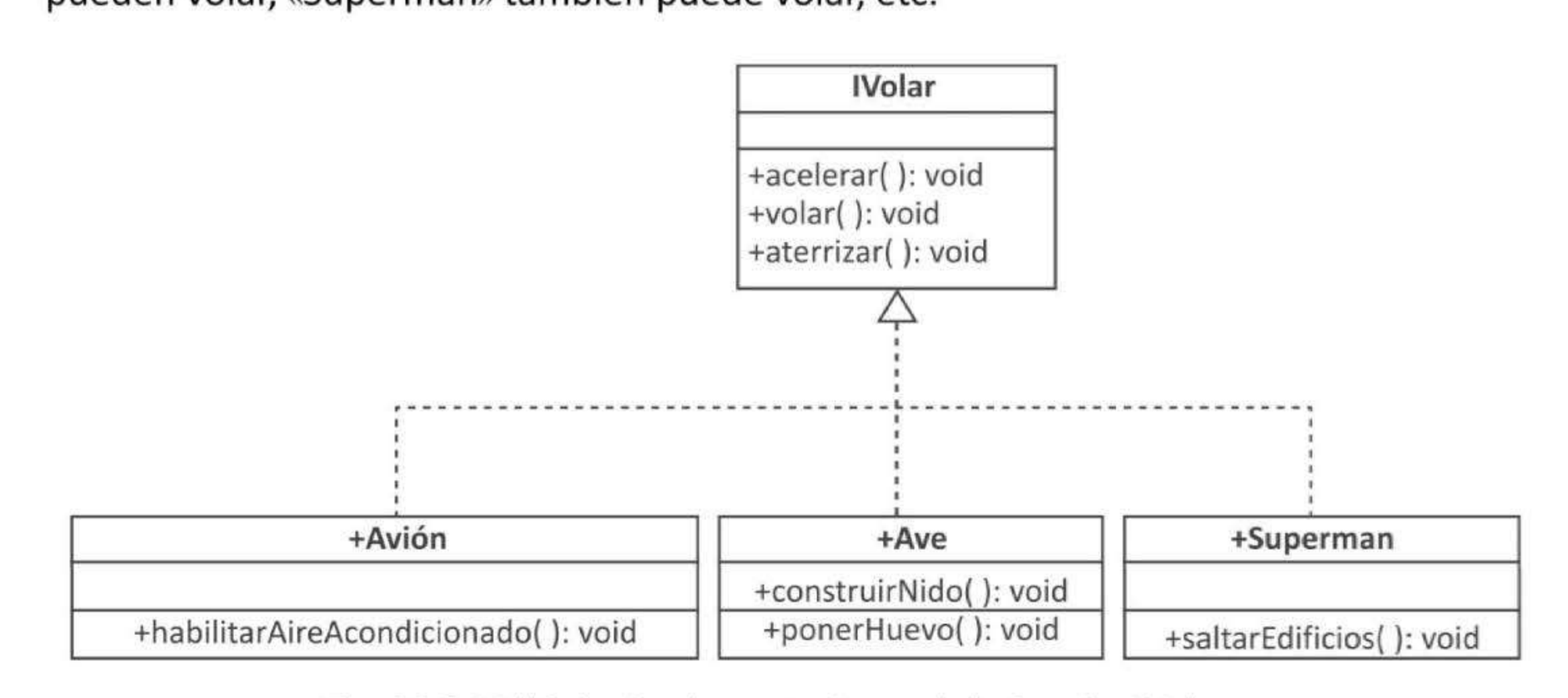

Fig. 11.2 Múltiples implementaciones de la interfaz IVolar

Un avión es un vehículo, y puede volar; un ave es un animal y también vuela. Estos ejemplos muestran que una clase puede heredarse de una superclase, pero además puede implementar algunas otras interfaces.

Esto parece herencia múltiple, pero no lo es en absoluto. El peligro de la herencia múltiple es que una clase podría heredar dos distintas implementaciones de un mismo método. Esto no es posible con las interfaces, porque una declaración de un método de la interfaz no provee ninguna implementación.

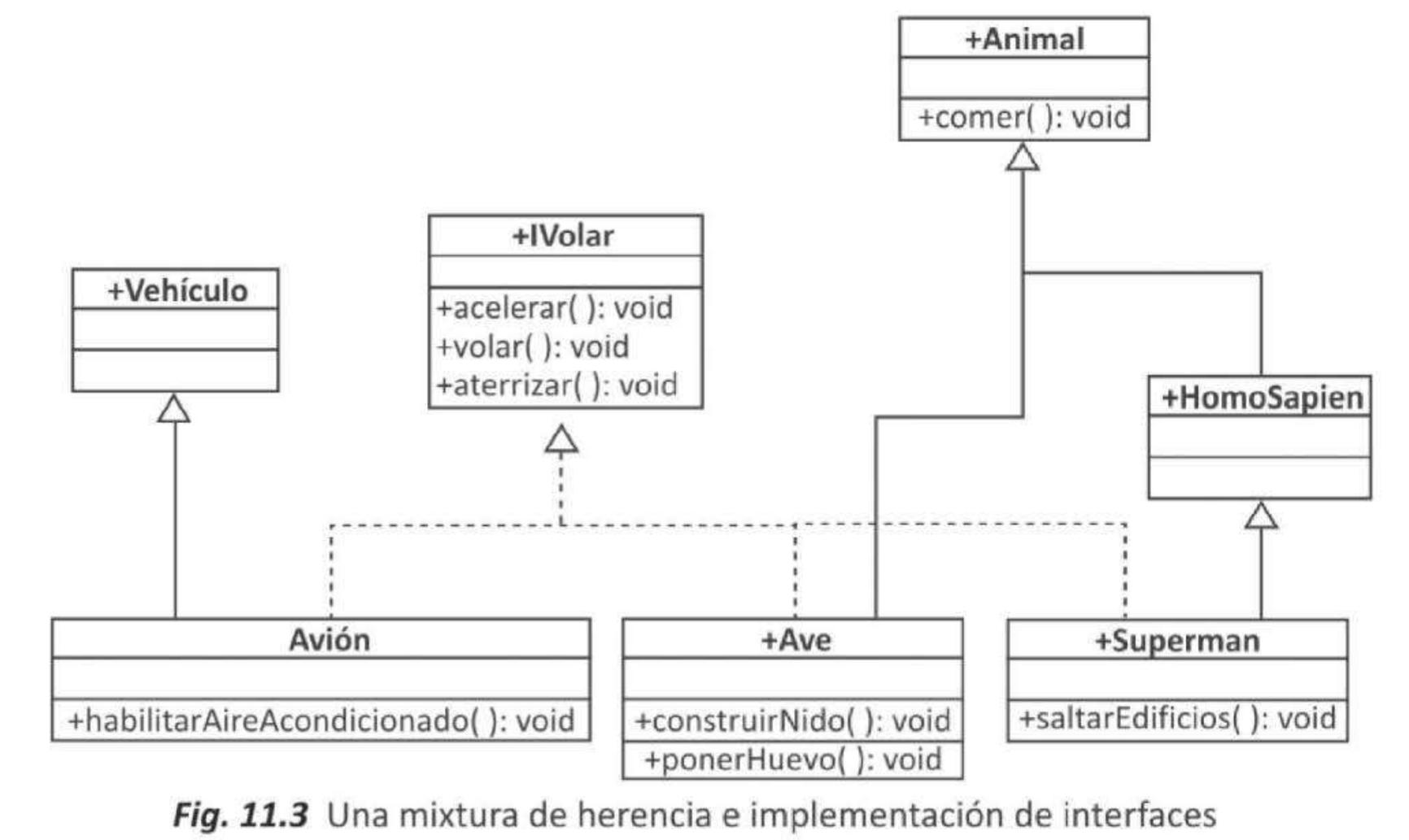

Fig. 11.3 Una mixtura de herencia e implementación de interfaces

Observe la implementación de la clase **Ave**:

```
public class Ave extends Animal implements IVolar {

    public void acelerar(){
            // Implementación
    }

    public void volar(){
            // Implementación
    }

    public void aterrizar(){
            // Implementación
    }

    public void comer() {
            // Implementación
    }

    public void construirNido() {
            // Implementación
    }
```

```
        public void ponerHuevo() {
                // Implementación
        }
    }
```

La cláusula **extends** debe estar antes de la **implements**. La clase **Ave** puede implementar sus propios métodos.

11.2 USO DE INTERFACES

11.2.1 DEFINICIÓN DE UNA INTERFAZ

Sintaxis

```
{acceso} interfaz nombre_interfaz {

    tipo_devuelto nombre_método1( lista_de_parámetros );
    tipo_devuelto nombre_método2( lista_de_parámetros );
    . . .

}
```

El acceso a una interfaz es *public* o no se usa. Cuando no se incluye el especificador de acceso, se utiliza el acceso por defecto, y la interfaz está disponible solo para otros miembros del paquete en que se declara. Cuando se declara como *public*, la interfaz puede utilizarse desde cualquier parte.

Los métodos que se declaran no tienen cuerpo y terminan con un punto y coma, son esencialmente métodos abstractos.

Observe el siguiente ejemplo ilustrativo:

```
interface CDemo1{

    int suma ( int a, int b );

}
```

A continuación tenemos otro ejemplo:

```
interface IDemo2{

    int producto( int a, int b );

}
```

11.2.2 Implementación de una interfaz

Una vez definida la interfaz, una o más clases pueden implementarla. Para ello, debe incluirse la cláusula **implements** en una definición de clase y luego crear los métodos definidos por la interfaz.

Sintaxis

```
{acceso} class nombre_clase [extends superclase] [implements interfaz1, ...] {

    // Cuerpo de la clase

}
```

En este ejemplo, acceso o es *public* o no se usa. Si una clase implementa más de una interfaz, se separan por comas. Los métodos que se implementan deben ser declarados como *public*. Además, el formato del método implementado debe coincidir exactamente con el formato especificado en la definición de la interfaz.

Señalamos un ejemplo que implementa las interfaces **IDemo1** e **IDemo2**:

```
class CDemo implements IDemo, IDemo2{

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

    public int producto( int a, int b ){
            return (a*b);
    }

}
```

Las clases que implementan interfaces pueden tener sus propias definiciones. En la siguiente versión de **CDemo** se añade el método **potencia()**:

```
class CDemo implements IDemo, IDemo2{

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

    public int producto( int a, int b ){
            return (a*b);
    }

    public int potencia( int n, int e ){
            int p = 1;
            for( int k = 1; k <= e; k++ ){
                    p = p * n;
            }
            return (p);
    }

}
```

A continuación, indicamos el programa **PDemo**, donde se ilustra el uso de la clase **CDemo**:

```
class PDemo{

    public static void main(String[] args){
            CDemo obj = new CDemo();
            System.out.println( "6 + 7 = " + obj.suma(6,7) );
            System.out.println( "5 * 8 = " + obj.producto(5,8) );
            System.out.println( "2 ^ 3 = " + obj.potencia( 2, 3 ) );
    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap11>javac PDemo.java   [Enter]

C:\java2n100\cap11>java PDemo         [Enter]
6 + 7 = 13
5 * 8 = 40
2 ^ 3 = 8
```

11.2.3 Variables e interfaces

Se pueden utilizar interfaces para declarar constantes compartidas por múltiples clases; en este caso las interfaces deben contener variables inicializadas. Cuando se implementen dichas interfaces en una clase, todas las variables estarán dentro del ámbito de la clase.

Por ejemplo, tenemos la interfaz **IConstantes**:

```
interface IConstantes{

    int PLAYA   = 1;
    int CINE    = 2;
    int BAILE   = 3;
    int ESTUDIO = 4;

}
```

La clase **CPregunta** implementa la interfaz **IConstantes**:

```
import java.util.Random;

class CPregunta implements IConstantes{

    Random rand = new Random();
```

Se tienen otras clases que comparten las mismas constantes:

```
int preguntar(){
            int prob = (int)(rand.nextDouble()*100);
            if (prob < 25)
                    return PLAYA;
            else if (prob < 50)
                    return CINE;
            else if (prob < 75)
                    return BAILE;
            else
                    return ESTUDIO;
        }

}
```

El programa ilustrativo para utilizar estas clases es **PTest**:

```
class PTest{

    public static void main(String[] args){
            CPregunta obj = new CPregunta();

            CRespuesta.responder( obj.preguntar() );
            CRespuesta.responder( obj.preguntar() );
            CRespuesta.responder( obj.preguntar() );
            CRespuesta.responder( obj.preguntar() );

        }
}
```

Compilación y ejecución:

```
C:\java2n100\cap11>javac PTest.java  [Enter]

C:\java2n100\cap11>java PTest        [Enter]
Playa
Cine
Baile
Playa
```

Observe que la clase **CPregunta** utiliza una de las clases estándares de Java, la clase **Random**. Esta proporciona números pseudoaleatorios; contiene varios métodos que permite obtener este tipo de números en la forma que requiera el programador. En el ejemplo, se utilizó el método **nextDouble()**, que retorna números aleatorios entre 0.0 y 1.0 pero menores que 1.0.

En el programa **PTest** se prueba las dos clases: **CPregunta** y **CRespuesta**; ambas implementan la interfaz **IConstantes** donde se definen las constantes PLAYA, CINE, BAILE, y ESTUDIO. Dentro de cada clase se hace referencia a estas constantes como si cada clase hubiera definido directamente estas constantes.

Observe que la salida que genera el programa varia cada vez que se ejecute el programa.

11.2.4 Interfaces pueden extenderse

Una interfaz puede heredar otra mediante la palabra clave **extends**.
La sintaxis es la misma que para la herencia de clases. Cuando una clase implementa una interfaz que hereda otra, debe implementar todos los métodos dentro de la cadena de herencia de la interfaz.

Por ejemplo, tenemos la interfaz **IA**:

```
interface IA{

    void método1();
    void método2();

}
```

Podemos tener una interfaz **IB** que extienda la interfaz **IA**:

```
interface IB extends IA{

    void método3();

}
```

La clase que implemente la interfaz **IB** debe implementar los tres métodos, tal como se ilustra a continuación:

```
class CEstudio implements IB{

            public void método1(){
                    System.out.println( "Implementación del método 1" );
            }

            public void método2(){
                    System.out.println( "Implementación del método 2" );
            }

            public void método3(){
                    System.out.println( "Implementación del método 3" );
            }

    }
```

Note que los métodos implementados en la clase **CEstudio** deben ser públicos.

El programa **CEstudio** muestra el uso de la clase **CEstudio**:

```
class PEstudio{

    public static void main(String[] args){

            CEstudio obj = new CEstudio();

            obj.método1();
            obj.método2();
            obj.método3();

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap11>javac PEstudio.java        [Enter]

C:\java2n100\cap11>java PEstudio              [Enter]
Implementación del método 1
Implementación del método 2
Implementación del método 3
```

Si a modo de prueba eliminamos el método **método3()** de la clase **CEstudio** obtenemos un error de compilación, tal como se muestra a continuación:

```
C:\java2n100\cap11>javac CEstudio.java                [Enter]
CEstudio.java:1: CEstudio is not abstract and does not override abstract method
método3() in IB
class CEstudio implements IB{
^
1 error
```

Como conclusión, cualquier clase que implemente una interfaz, debe implementar todos los métodos que defina la interfaz, incluyendo los heredados de otras interfaces.

11.3 EJEMPLO ILUSTRATIVO

Presentaremos un ejemplo donde se muestra el uso de clases e interfaces en el diseño de un sistema. Es bastante sencillo, pero sirve para obtener conclusiones de cómo aplicar estos conceptos.

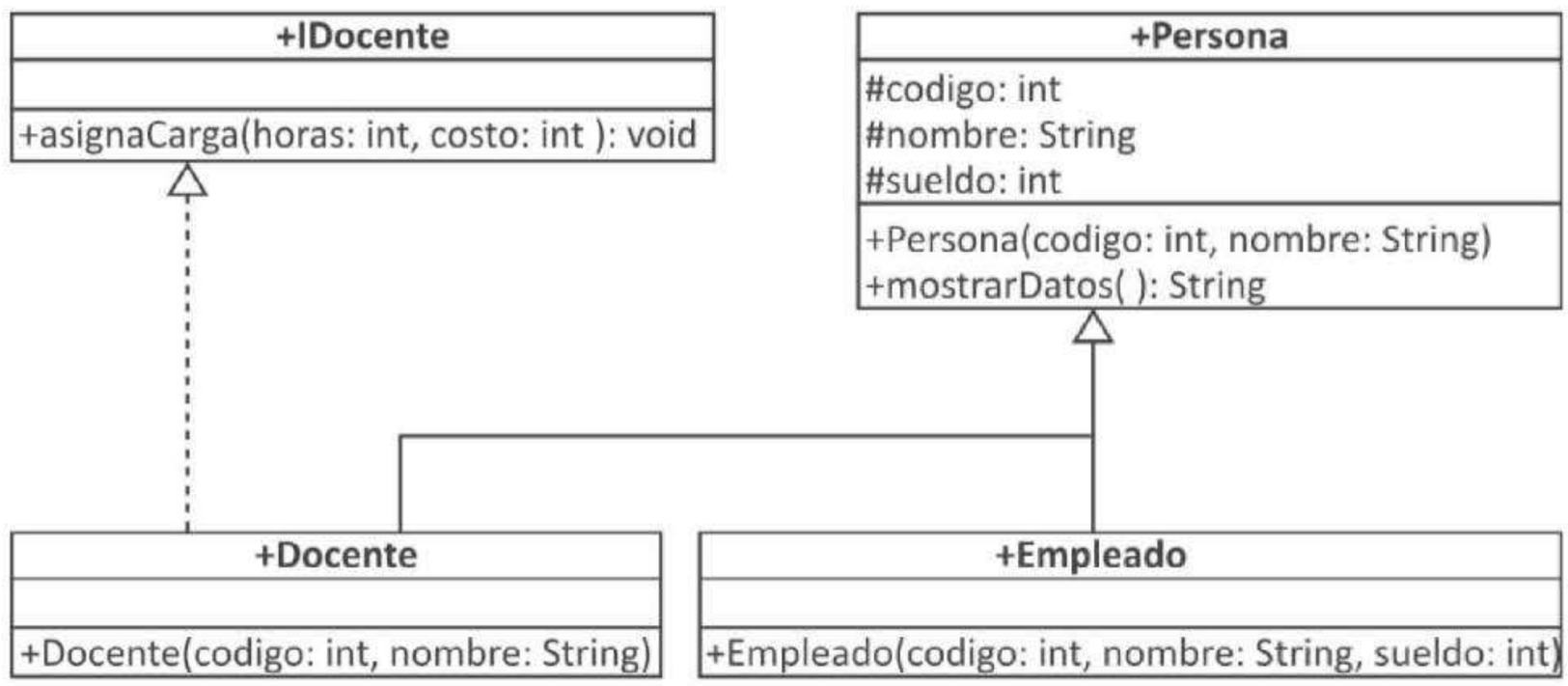

Fig. 11.4 Aplicación de clases e interfaces

Se tiene una clase abstracta *Persona* de la que derivan las clases **Docente** y **Empleado**; además la clase **Docente** implementa la interfaz **IDocente**.

La implementación de la clase **Persona** es:

Persona.java

```
public abstract class Persona {

        protected int código;
        protected String nombre;
        protected int sueldo;

        public Persona(int codigo, String nombre){

                this.código = código;
                this.nombre = nombre;

        }

        public String mostrarDatos(){
                return código + "\t" + nombre + "\t" + sueldo;
        }

}
```

La implementación de la interfaz **IDocente** es:

IDocente.java

```
public interface IDocente {

        public void asignaCarga(int horas, int costo);

}
```

La implementación de la clase **Empleado** es:

Empleado.java

```
public class Empleado extends Persona {

    public Empleado(int código, String nombre, int sueldo) {

            super( código, nombre );
            this.sueldo = sueldo;

    }
}
```

La implementación de la clase **Docente** es:

Docente.java

```
public class Docente extends Persona implements IDocente {

    public Docente(int código, String nombre) {

            super( código, nombre );

    }

    public void asignaCarga(int horas, int costo){
            this.sueldo = horas * costo;
    }

}
```

El programa que ilustra el uso de estas clases es **Aplicacion**, su código es el siguiente:

Aplicaciones.java

```
public class Aplicacion{

    public static void main(String[] args){

            Empleado emp = new Empleado( 1000, "Ricardo Marcelo", 2000 );
            Docente  doc = new Docente( 2000, "Sergio Matsukawa" );

            doc.asignaCarga( 100, 50 );

            System.out.println( emp.mostrarDatos() );
            System.out.println( doc.mostrarDatos() );

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap11>javac Aplicacion.java      [Enter]

C:\java2n100\cap11>java Aplicacion            [Enter]
1000 Ricardo Marcelo          2000
2000 Sergio Matsukawa 5000
```

En este ejemplo se ha comprobado, perfectamente, la teoría sobre Clase abstracta, herencia, interfaces y variables protegidas. Finalmente, sacará sus propias conclusiones y aplicará estos conceptos en sus futuros diseños.

Excepciones

CAPÍTULO 12

Uno de los mayores problemas en la programación es el tratamiento de errores, ya sean causados por fallas o limitaciones del hardware (por ejemplo errores de lectura de archivos) y/o por el software (casos en los que no se cumple una determinada condición).

Para facilitar el tratamiento de errores en Java se ha creado el concepto de **Excepciones**, el cual se refiere a una situación de error en la ejecución de un programa. Cada vez que sucede una excepción (un error) el programa debe enmendarla, normalmente mostrando un mensaje de error y ejecutando alguna rutina de tratamiento de errores.

12.1 FUNDAMENTOS

12.1.1 Tipos de errores

Cuando desarrollamos programas nos encontramos con tres tipos de errores:

- Errores de sintaxis
- Errores de lógica
- Errores de ejecución

a. Errores de sintaxis

Los errores en la sintaxis son causados cuando el compilador de Java (javac) no puede reconocer una instrucción. Esto causa que el compilador devuelva un mensaje de error, usualmente, con una línea de código de referencia. Los errores de sintaxis también son conocidos como errores en tiempo de compilación.

Por ejemplo, si tenemos la siguiente clase:

```
class Test1{

    public static void main(String[] args){
            System.out.println("Hello World!")
    }

}
```

Cuando compilemos esta clase tenemos el siguiente error:

```
C:\java2n100\cap12>javac Test1.java  [Enter]
Test1.java:5: ';' expected
        }
        ^
1 error
```

Este error nos está indicando que en el archivo **Test1.java** en la línea 5 se espera un punto y coma (;) y como podemos comprobar, exactamente falta un punto y coma.

b. Errores de lógica

Son conocidos como BUGS; estos nos tomarán tiempo hasta encontrarlos. Si damos a elegir cualquier programador, elegiría los errores en tiempo de compilación.

Para encontrar un error de lógica debemos identificar la clase donde podría estar el error, y si es posible, el métodos o posibles métodos donde podría estar; luego de eso hacer una depuración (*debug*) línea por línea (*trace*) en cada uno de los métodos.

Por ejemplo, tenemos el siguiente programa que sirve para calcular el factorial de un número:

```
class Test2{

    public static void main(String[] args){

            int n = Integer.parseInt( args[0] );
            int f = 1;
            for( ; n > 1; n-- );
            {
                    f *= n;
            }
            System.out.println( "factorial = " + f );

    }

}
```

No tenemos ningún error de sintaxis, y aparentemente tampoco de lógica, pero qué sucede cuando compilamos y ejecutamos este programa:

```
C:\java2n100\cap12>javac Test2.java          [Enter]

C:\java2n100\cap12>java Test2 5              [Enter]
factorial = 1
```

Podemos comprobar que el resultado obtenido no es el esperado.

El error está en la siguiente instrucción:

```
for( ; n > 1; n-- );
```

No debe ir punto y coma al final de esta línea, porque estaría terminando la instrucción **for**; por lo tanto, se trata de un bucle vacío.

c. Errores de ejecución

Se producen cuando la aplicación está en producción o prueba (*testing*), y aparecen por una situación anormal durante la ejecución de alguna instrucción.

El siguiente programa permite obtener la división de dos números:

```
class Test3{

    public static void main(String[] args){

            int n1 = Integer.parseInt( args[0] );
            int n2 = Integer.parseInt( args[1] );
            int rpta = n1 / n2;

            System.out.println( "n1 = " + n1 );
            System.out.println( "n2 = " + n2 );
            System.out.println( "rpta = " + rpta );

    }

}
```

La compilación del programa no presenta ningún error:

```
C:\java2n100\cap12>javac Test3.java  [Enter]
```

Cuando le pasamos dos valores diferentes de cero, tampoco tenemos problemas:

```
C:\java2n100\cap12>java Test3 7 2     [Enter]
n1 = 7
n2 = 2
rpta = 3
```

El problema se presenta cuando el segundo valor que se le pasa es cero (0), se produce un error y los errores en Java están asociados a excepciones; por lo tanto se desencadena una excepción:

```
C:\java2n100\cap12>java Test3 7 0     [Enter]
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Test3.main(Test3.java:7)
```

Note que la división por cero es una situación anormal, por lo tanto genera un error. El problema está en controlar este tipo de errores y evitar que el programa interrumpa su ejecución abruptamente.

d. Errores típicos

Los siguientes errores aparecen en la mayoría de programadores cada cierto tiempo, pero podemos verlo regularmente en el código de los programadores novatos.

- **El nombre de la Clase pública no coincide con el nombre del archivo**
  Cada archivo **.java** puede contener solo una Clase pública. El nombre de esa Clase pública debe coincidir exactamente con el nombre del archivo antes de la extensión **.java**, respetando incluso las mayúsculas y minúsculas. Por ejemplo, una Clase pública llamada **MyClass** debe estar en un archivo **MyClass.java** y no en **myclass.java**. Este es un error de sintaxis.

- **Una Clase no está en el directorio correcto**
  Este error de sintaxis ocurre si el comando **javac** no puede encontrar un archivo **.java** en el directorio esperado. Si una clase está en el paquete por defecto (no tiene la declaración del **package** en la primera línea), entonces esta clase pertenece al directorio actual de trabajo o al directorio donde apunta **SOURCEPATH** de *javac*. Si una clase está en un paquete, entonces pertenece a un subdirectorio debajo del directorio actual de trabajo o a un subdirectorio debajo del directorio donde apunta **SOURCEPATH** de *javac*. Por ejemplo, si existe una clase llamada **prog.hello.HelloWorld**, entonces su nombre es **HelloWorld** y su paquete es **prog.hello.** Si **SOURCEPATH** está configurado a **c:\src**, entonces el archivo **HelloWorld.java** debe estar ubicado en el subdirectorio **c:\src\prog\hello**.

- **Uso incorrecto del operador de asignación (== versus =)**
  En Java este error puede ser incluso un error de lógica o sintaxis. Para comparar dos valores se usa el operador == (el operador de igualdad). Para asignar el valor de la derecha a la variable de la izquierda se usa el operador = (operador de asignación). Los programadores novatos a veces escriben: **if** (**miValor = valorEsperado**) este código intenta evaluar el **valorEsperado** como un valor lógico en lugar de intentar la evaluación de igualdad entre **miValor** y **valorEsperado**. Si **valorEsperado** es de tipo lógico, entonces el código tendrá un error de lógica y probará si el **valorEsperado** es verdadero o falso. Si **valorEsperado** no es del tipo lógico, entonces el código lanzará un error de compilación debido a que la instrucción **if** requiere un valor lógico que sea retornado de la comparación (**miValor = valorEsperado**), pero en Java el operador = siempre retorna el valor de la derecha.

- **Error en mayúsculas y minúsculas**
  Java es un lenguaje *case sensitive*, es decir que debemos respetar las mayúsculas y minúsculas, **miVar** no es igual a **mivar**. Si hacemos referencia a un identificador que no está bien escrito, entonces obtendremos un error de sintaxis.

- **Olvidar que los índices en Java empiezan en 0**
  Los índices de los arreglos de Java y las listas empiezan en 0, **myArray**[0] o **myList.get**(0). Debe asegurarse que su bucle **for** no cause errores por este motivo. Si hacemos más iteraciones de las que son posibles, entonces obtendremos el error: **ArrayIndexOutOfBoundsException**. Si hacemos menos bucles de los requeridos, entonces tendremos un error de lógica.

- **NullPointerException**
  Los **NullPointerExceptions** son errores de lógica causados cuando un programa intenta acceder a métodos o variables en una referencia que es nula. Si nuestro objeto no ha sido inicializado o ha sido establecido a **null** con el operador =, entonces la llamada a un método o el acceso a una de sus variables no es válido.

12.1.2 ¿Qué es una excepción?

Una excepción en Java es un objeto que describe una condición excepcional, es decir, un error que se ha dado en una parte del código. Cuando se produce una condición de excepción, se crea un objeto que representa esta excepción y se lanza al método que ha causado el error. Dicho método puede elegir entre gestionar la excepción él mismo o pasarla al método que lo ha invocado. De cualquiera de las dos formas, en un punto determinado se capturará la excepción y se procesará.

Las excepciones pueden ser generadas por el intérprete de Java o de forma manual por el propio código. Normalmente, las excepciones generadas por Java están relacionadas con errores fundamentales que violan las reglas del lenguaje Java o las restricciones del entorno de ejecución Java. Las excepciones generadas de forma manual se utilizan generalmente para informar acerca de alguna condición de error personalizada; por ejemplo, un error en el proceso de la lógica del negocio.

12.1.3 Gestión de excepciones

La gestión de excepciones se realiza a través de cinco palabras claves: **try, catch, throw, throws** y **finally**.

try	Dentro del bloque **try** se deben incluir las sentencias que se quieren controlar, en otras palabras, las que podrían desencadenar un error.
catch	El bloque **catch** captura las excepciones que se desencadenan en el bloque **try**; se debe especificar el tipo de excepción. En este bloque se procesa la excepción.
throw	Para generar excepciones en forma manual debemos utilizar una instrucción **throw**.
throws	La palabra clave **throws** se utiliza para indicar las excepciones que un método no puede gestionar y que deben ser controladas por el código que llame a estos métodos.
finally	El bloque **finally** se utiliza para indicar el bloque se instrucciones que siempre se debe ejecutar.

12.1.4 Esquema general

El siguiente es el esquema general de un bloque de gestión de excepciones:

```
try{
    // Bloque de instrucciones a controlar
}
catch ( TipoDeExcepción1 var1 ) {
    // Gestión de excepción de tipo TipoDeExcepción1
}
catch ( TipoDeExcepción2 var2 ) {
    // Gestión de excepción de tipo TipoDeExcepción2
}
. . .
finally {
    // Bloque de instrucciones que siempre se debe ejecutar
}
```

En este caso **TipoDeExcepción** representa un tipo de excepción específico que se pretende controlar.

12.1.5 Tipos de excepciones

Todos los tipos de excepciones son subclases de la clase **Throwable**, incorporada en Java. Por lo tanto, **Throwable** ocupa el primer lugar en la jerarquía de clases de excepciones (ver Fig. 12.1), inmediatamente después se encuentran dos subclases que dividen las excepciones en dos grupos.

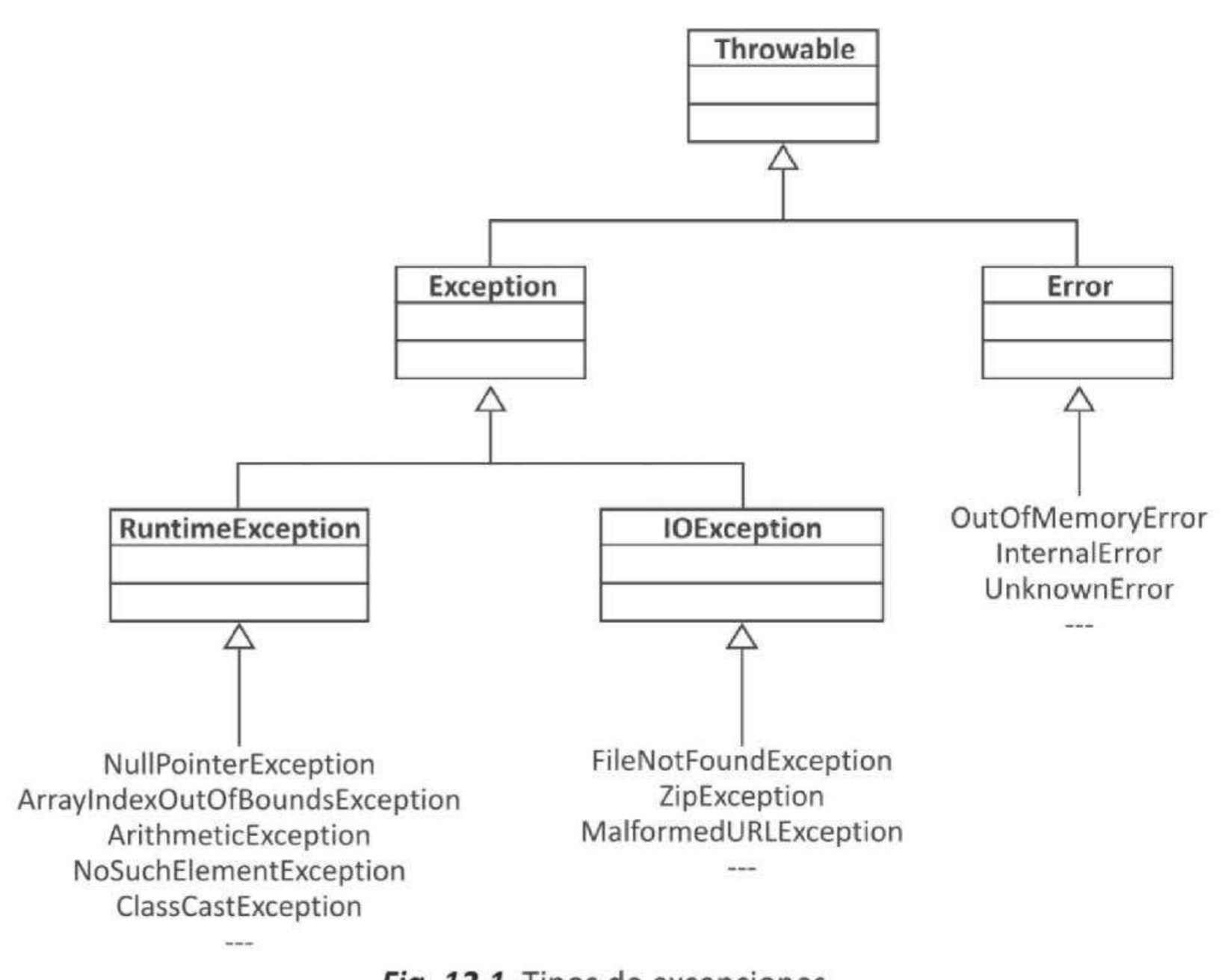

Fig. 12.1 Tipos de excepciones

La clase **Exception** y sus subclases representan las condiciones excepcionales (errores) que normalmente debemos controlar en nuestros programas. También es la clase desde donde derivan las clases personalizadas de excepciones.

Existe una importante subclase de **Exception** llamada **RuntimeException**. **RuntimeException** es una superclase de excepciones que puede generarse durante la operación normal de la máquina virtual de Java. Incluye excepciones para errores como la división por cero, la utilización de un índice no válido para un arreglo.

La clase **Error** es una subclase de **Throwable** que indica problemas serios que en una aplicación aceptable no debería encontrarse. La mayor parte de estos errores son condiciones anormales. El intérprete de Java utiliza estas excepciones para indicar los errores relacionados con el momento de la ejecución, el desbordamiento de la pila (**StackOverflowError**) es un ejemplo de este tipo de error.

12.2 GESTIONANDO EXCEPCIONES

12.2.1 EXCEPCIONES NO CAPTURADAS

Lo primero que debemos saber es qué sucede cuando la máquina virtual de Java genera una excepción y esta no es controlada en el programa; para eso utilizaremos el siguiente programa, en él se está provocando un error de manera intencional, y además las líneas de código han sido enumeradas para una mejor identificación:

Cuando el intérprete de Java detecta un intento de división por cero en la línea 7, instancia un objeto de excepción y luego genera esa excepción. Esto provoca que la ejecución de **Test4** se detenga, porque una vez generada la excepción, debe capturarse por un gestor de excepciones y tratarse de forma inmediata. Las líneas 9 y 10 nunca llegan a ejecutarse.

En este ejemplo, no se ha programado un gestor de excepciones, de manera que la excepción es tratada por el gestor por defecto que proporciona el intérprete de Java. Cualquier excepción que el programa no capture, será en última instancia procesada por el gestor por defecto, que presenta un mensaje que describe la excepción, imprime el trazado de la pila y finaliza la ejecución del programa.

A continuación tenemos la excepción generada cuando el programa **Test4** se ejecuta:

```
C:\java2n100\cap12>javac Test4.java                    [Enter]

C:\java2n100\cap12>java Test4                          [Enter]
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test4.main(Test4.java:7)
```

Note usted que en el trazado de la pila se incluye:

- El nombre de la clase: **Test4**
- El nombre del método: **main**
- El nombre del archivo: **Test4.java**
- El número de línea: **7**
- Tipo de excepción: **ArithmeticException**

El tipo de excepción describe de manera más específica el tipo de error producido.

Java proporciona una gran cantidad de tipos de excepciones que se ajustan a las distintas clases de errores de ejecución que se pueden generar.

El trazado de la pila siempre mostrará la secuencia de llamadas a métodos que preceden al error. Por ejemplo, tenemos el programa **Test5**, una versión diferente del mismo tipo de error; en este caso el error se produce en el método **operacion**:

```
1.  class Test5{
2.
3.          public static void main(String[] args){
4.
5.                  operacion();
6.                  System.out.println( "FIN" );
7.          }
8.
9.          static void operacion(){
10.
11.                 int n1 = 17;
12.                 int n2 = 0;
13.                 int rpta = n1 / n2;
14.
15.                 System.out.println( "rpta = " + rpta );
16.
17.         }
18.
19. }
```

A continuación tenemos el trazado de la pila que muestra el gestor de excepciones por defecto:

```
C:\java2n100\cap12>javac Test5.java          [Enter]

C:\java2n100\cap12>java Test5                          [Enter]
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test5.operacion(Test5.java:13)
        at Test5.main(Test5.java:5)
```

Podemos comprobar que el final de la pila es la línea 5 de **main**, que es desde donde se llama al método **operacion()**, que provoca la excepción en la línea 13. La pila de llamadas es muy útil para la depuración porque muestra la secuencia exacta de pasos que generó al error.

12.2.2 Captura de errores

Si bien es cierto que el gestor de excepciones por defecto es útil para la depuración, no resulta práctico ver que un programa termine abruptamente y muestre el trazado de la pila que generalmente los usuarios finales no entienden nada; por lo tanto, debemos implementar un gestor de errores.

Por ejemplo tenemos el programa **Test6** donde implementamos el gestor de errores para la división por cero:

```
1.  class Test6{
2.
3.          public static void main(String[] args){
4.
5.                  int n1 = 17;
6.                  int n2 = 0;
7.                  try{
8.                          int rpta = n1 / n2;
9.                          System.out.println( "rpta = " + rpta );
10.                 }
11.                 catch( ArithmeticException e ){
12.                         System.out.println( "Error: Division por cero"
);
13.                 }
14.                 System.out.println( "FIN" );
15.         }
16.
17. }
```

En el bloque **try** se incluye las instrucciones que queremos controlar, y en **catch** se especifica el tipo de excepción que se desea controlar.

La ejecución del programa genera la siguiente salida:

```
C:\java2n100\cap12>javac Test6.java                    [Enter]

C:\java2n100\cap12>java Test6                          [Enter]
Error: Division por cero
FIN
```

Note que la llamada a **println** en la línea 9 nunca llega a ejecutarse. Una vez generada la excepción en la línea 8, el control del programa se transfiere del bloque **try** al bloque **catch** y no regresa a **try**. Después de ejecutar el bloque **catch**, el control del programa continúa con las líneas después del bloque **catch**, en este caso la línea 14.

Un bloque **try** y su correspondiente **catch** forman una unidad. El campo de acción del bloque **catch** se limita a aquellas instrucciones especificadas dentro del bloque **try** que le precede inmediatamente. Un bloque **catch** no puede capturar una excepción generada dentro de otro bloque **try**, excepto en el caso que se encuentren anidadas.

El objetivo de implementar un gestor de excepciones, es capturar y resolver la excepción, luego continuar como si nada hubiera pasado. Por ejemplo, tenemos el programa **Test7**:

```
1.  import java.util.Random;
2.
3.  class Test7 {
4.
5.     public static void main(String[] args){
6.
7.          nt a = 0, b = 0, c = 0;
8.          Random rand = new Random();
9.          for( int i = 0; i < 5; i++ ) {
10.             try{
11.                a = rand.nextInt(100);
12.                b = rand.nextInt(100);
13.                c = 100 / (a/b);
14.             }
15.             catch ( ArithmeticException e ){
16.                System.out.println( "Error: Division por cero" );
17.                c = 0;
18.             }
19.             System.out.println( a + "\t" + b + "\t" + c );
20.          }
21.          System.out.println( "FIN" );
22.       }
23.
24.  }
```

Por cada iteración del bucle **for** se obtienen dos números aleatorios, estos dos se dividen y el cociente se utiliza como divisor del valor 100; el resultado se almacena en la variable **c**. Si en cualquiera de las dos operaciones se produce una división por cero, se captura el error, se asigna a **c** el valor cero y el programa continúa.

A continuación tenemos la ejecución de este programa:

```
C:\java2n100\cap12>javac Test7.java                    [Enter]

C:\java2n100\cap12>java Test7                          [Enter]
50      48      100
50      8       16
Error: Division por cero
15      65      0
86      70      100
Error: Division por cero
41      80      0
FIN
```

12.2.3 Descripción de la excepción

La clase **Throwable** sobreescribe el método **toString()** definido por la clase **Object** y retorna una cadena con la descripción de la excepción. La descripción puede mostrarse a través de **println()** pasándole simplemente la excepción como un argumento. Por ejemplo, el bloque **catch** del programa **Test7** puede volverse a escribir de la siguiente manera:

```
catch ( ArithmeticException e ){
    System.out.println( e );
    c = 0;
}
```

La ejecución del programa se presentaría de la siguiente manera:

```
C:\java2n100\cap12>javac Test7.java                    [Enter]

C:\java2n100\cap12>java Test7                          [Enter]
75      27      50
java.lang.ArithmeticException: / by zero
49      80      0
81      61      100
java.lang.ArithmeticException: / by zero
29      34      0
97      88      100
FIN
```

Aunque en este contexto el mensaje no tiene un gran interés, la posibilidad de presentar la descripción de una excepción es de gran utilidad en otras ocasiones; particularmente cuando se está experimentando con las excepciones o se está depurando un programa.

12.2.4 Cláusulas catch múltiples

En muchos casos, una única secuencia de instrucciones puede generar más de un tipo de excepción. Al manejar este tipo de situaciones podemos especificar varias cláusulas **catch**, para que cada una de ellas capture un tipo diferente de excepción. Cuando se genera una excepción se verifica cada cláusula **catch** en el mismo orden que se ha escrito y se ejecuta la primera, cuyo tipo coincida con la excepción. Después de ejecutarse una cláusula **catch**, las demás son ignoradas y la ejecución continúa después del bloque **try/catch**.

En el siguiente programa se capturan dos tipos diferentes de excepciones:

```
1. class Test8{
2.
3.        public static void main(String[] args){
4.
5.          try{
6.
7.              int n = Integer.parseInt( args[0] );
8.              System.out.println( "n = " + n );
9.              int a = 100 / n;
10.                 int b[] = { 15 };
11.                 b[5] = 50;
12.             }
13.             catch( ArithmeticException e1 ){
14.                 System.out.println( "Division por cero.\n" + e1 );
15.             }
16.             catch( ArrayIndexOutOfBoundsException e2 ) {
17.             System.out.println( "Indice fuera de rango.\n" + e2 );
18.             }
19.             System.out.println( "FIN" );
20.
21.         }
22.
23.       }
```

Compilación del programa:

```
C:\java2n100\cap12>javac Test8.java  [Enter]
```

Este programa genera las siguientes excepciones:

– Excepción de tipo **ArrayIndexOutOfBoundsException** en la línea 7 cuando no se le pasa ningún parámetro.

```
C:\java2n100\cap12>java Test8        [Enter]
Indice fuera de rango.
java.lang.ArrayIndexOutOfBoundsException: 0
FIN
```

– Excepción de tipo **NumberFormatException** en la línea 7 cuando se le pasa como parámetro una cadena no numérica. Como se trata de una excepción no controlada se muestra el trazado de la pila y finaliza su ejecución.

```
C:\java2n100\cap12>java Test8 Hugo          [Enter]
Exception in thread "main" java.lang.NumberFormatException: For input
string: "Hugo"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at Test8.main(Test8.java:7)
```

– Excepción de tipo **ArithmeticException** en la línea 9 cuando se le pasa como parámetro el valor cero (0).

```
C:\java2n100\cap12>java Test8 0                    [Enter]
n = 0
Division por cero.
java.lang.ArithmeticException: / by zero
FIN
```

– Excepción de tipo **ArrayIndexOutOfBoundsException** en la línea 11 cuando se le pasa como parámetro un valor numérico diferente de cero (0).

```
C:\java2n100\cap12>java Test8 3                    [Enter]
n = 3
Indice fuera de rango.
java.lang.ArrayIndexOutOfBoundsException: 5
FIN
```

Cuando se utilizan varias cláusulas **catch**, es importante tener en cuenta que las excepciones deben ir antes de cualquiera de sus superclases. Esto es así porque la cláusula **catch** que utilice una superclase capturará excepciones de este tipo y cualquiera de sus subclases; por lo tanto, estas no se ejecutarán si están después de la superclase. Finalmente en Java se produce un error si hay un código no alcanzable.

Por ejemplo, tenemos el programa **Test9**:

```
1.  class Test9{
2.
3.                  public static void main(String[] args){
4.
5.                          try{
6.                                  int a = 100;
7.                                  int b = 0;
8.                                  int c = a / b;
9.                                  System.out.println( "c = " + c );
10.
11.                         }
12.                         catch( Exception e1 ){
13.                                 System.out.println( "Error.\n" + e1
);
14.                         }
15.                         catch( ArithmeticException e2 ) {
16.                                 System.out.println(  "Division  por
cero.\n" + e2 );
17.                         }
18.                         System.out.println( "FIN" );
19.
20.                 }
21.
22.         }
```

Si compilamos este programa tenemos el siguiente error:

```
C:\java2n100\cap12>javac Test9.java          [Enter]
Test9.java:15: exception java.lang.ArithmeticException has already
been caught
                catch( ArithmeticException e2 ) {
                ^
1 error
```

Este mensaje de error nos indica que la excepción **ArithmeticException** que se pretende capturar en la línea 15, ya ha sido capturada, en este caso en la línea 12 porque la **Exception** es la clase padre de **ArithmeticException**; esto quiere decir que el segundo **catch** nunca se ejecutará. Para solucionar este problema se debe invertir el orden de las cláusulas **catch**.

12.2.5 Instrucción throw

Hasta ahora solamente se han capturado excepciones generadas por el intérprete de Java; sin embargo, con la instrucción **throw** podemos generar una excepción de forma explícita.

Sintaxis

```
throw instancia_de_una_excepción;
```

Donde **instancia_de_una_excepción** debe ser un objeto de tipo **Throwable** o una subclase de **Throwable**. Existen dos maneras de obtener un objeto **Throwable**, utilizando un parámetro en la cláusula **catch** o creando uno con el operador **new**.

A continuación tenemos el programa **Test10**, aquí se muestra un ejemplo ilustrativo del uso de la instrucción **throw**.

Detallamos una descripción de su funcionamiento:

1. Desde la línea 6 se llama al método **lanzaExcepcion()**, esta línea pertenece al método **main()**.
2. Desde la línea 20 que pertenece al método **lanzaExcepción()**, se lanza una excepción de tipo **NullPointerException**.
3. En la línea 22 se captura la excepción, en la 23 se imprime un mensaje y en la 24 se vuelve a lanzar la misma excepción.
4. La excepción es capturada en el método **main**, específicamente en la línea 9; luego en la línea 10 imprime el mensaje de error.
5. Finalmente ejecuta la línea 12 y termina el programa.

Listado del programa **Test10**:

```
1.  class Test10{
2.
3.          public static void main(String[] args){
4.
5.              try{
6.                  lanzaExcepcion();
7.                  System.out.println( "main: esta linea no se ejecuta" );
8.              }
9.              catch( NullPointerException e ){
10.                 System.out.println( "Error.\n" + e );
11.             }
12.             System.out.println( "FIN" );
13.
14.         }
15.
16.         static void lanzaExcepcion() {
17.
18.             try {
19.                 // Lanza una excepción
20.                 throw new NullPointerException( "Excepción de Usuario" );
21.             }
22.             catch ( NullPointerException e ) {
23.                 System.out.println( "Mensaje desde lanzaExcepcion()"
);
24.                 throw e; // Vuelve a lanzar la excepción
25.             }
26.
27.         }
28.
29.     }
```

Compilación y ejecución:

```
C:\java2n100\cap12>javac Test10.java          [Enter]

C:\java2n100\cap12>java Test10                [Enter]
Mensaje desde lanzaExcepcion()
Error.
java.lang.NullPointerException: Excepci%n de Usuario
FIN
```

Este programa también sirve de ejemplo para mostrar cómo se crea un objeto de la clase **NullPointerException** de Java. La instrucción es la siguiente:

```
throw new NullPointerException( "Excepción de usuario" );
```

Aquí se utiliza el operador **new** para construir una instancia de **NullPointerException**. Muchas de las excepciones incorporadas por Java tienen dos constructores, uno sin parámetros y otro con un parámetro de tipo **String**. Cuando se utiliza la segunda

forma, el argumento especifica una cadena que describe la excepción, esta cadena se imprime cuando utiliza el objeto como argumento de **print()**o **println()**. También se puede obtener con el método **getMessage()** que está definido en **Throwable**.

12.2.6 Palabra clave throws

Si un método es capaz de generar una excepción que él mismo no puede gestionar, se debe especificar el comportamiento, de manera que los métodos que llamen a ese primer método puedan protegerse contra esa excepción. Para ello se debe incluir la cláusula **throws** en la definición del método.

La cláusula **throws** declara una lista de excepciones que el método puede lanzar. Esto es necesario para todas las excepciones, excepto las de tipo **Error** o **RuntimeException**, o cualquiera de sus subclases. Todas las demás excepciones que un método puede lanzar, se deben declarar en la cláusula **throws**; si no se producirá un error de compilación.

Sintaxis

```
tipo nombre_método ( lista_de_parametros ) throws lista_de_excepciones {

    // Implementación

}
```

Donde **lista_de_excepciones** es una lista de las excepciones separadas por comas que el método puede lanzar.

A continuación tenemos como ejemplo el programa **Test11**, este programa es incorrecto porque el método **lanzaExcepcion()** trata de lanzar una excepción que no está declarada en una cláusula **throws**, simplemente el programa no compilará.

```
class Test11{

    public static void main(String[] args){

            lanzaExcepcion();
            System.out.println( "FIN" );

    }

    static void lanzaExcepcion() {

            System.out.println( "Mensaje desde lanzaExcepcion()" );
            throw new IllegalAccessException( "Nueva Excepcion" );

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap12>javac Test11.java [Enter]
Test11.java:13: unreported exception java.lang.IllegalAccessException; must
be c
aught or declared to be thrown
              throw new IllegalAccessException( "Nueva Excepcion" );
              ^
1 error
```

Para conseguir que este programa pueda compilar, se deben realizar dos indicaciones:

1. Declarar que **lanzaExcepcion()** pueda lanzar excepciones de tipo `I`**llegalAccessException.**
2. Hacer que el método **main()** pueda capturar excepciones, a través de un bloque **try / catch**.

Los cambios se pueden ver en el programa **Test12**:

```
class Test12{

    public static void main(String[] args){

            try     {
                    lanzaExcepcion();
            }
            catch ( IllegalAccessException e ) {
                    System.out.println( "Error: " + e );
            }
            System.out.println( "FIN" );

    }

    static void lanzaExcepcion() throws IllegalAccessException {

            System.out.println( "Mensaje desde lanzaExcepcion()" );
            throw new IllegalAccessException( "Nueva Excepcion" );

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap12>javac Test12.java [Enter]

C:\java2n100\cap12>java Test12       [Enter]
Mensaje desde lanzaExcepcion()
Error: java.lang.IllegalAccessException: Nueva Excepcion
FIN
```

Se puede comprobar que con estos cambios no tenemos ningún problema de compilación, ni tampoco de ejecución. La excepción lanzada por el método **lanzaExcepcion()** es capturada y procesada en el método **main()**.

12.2.7 Palabra clave finally

Cuando se lanzan excepciones, la ejecución de un método sigue un camino no lineal y bastante brusco que altera el flujo normal. Dependiendo la manera en que se haya codificado el método, es posible que una excepción provoque que el método finalice de forma prematura. Esto puede suponer un problema en algunos métodos; por ejemplo si un método abre un archivo cuando inicia su ejecución y lo cierra cuando termina, entonces puede suceder que el mecanismo de gestión de excepciones, omita la ejecución de las instrucciones de cierre de archivo. La palabra clave **finally** está diseñada para resolver este tipo de situaciones.

finally crea un bloque de instrucciones que se ejecutarán después de haber completado un bloque **try / catch** y antes de que se ejecute el código que sigue al bloque **try / catch**. El bloque **finally** se ejecutará si se lanza o no una excepción.

Si se lanza una excepción, el bloque **finally** se ejecuta incluso aunque ninguna cláusula `catch` coincida con la excepción. Cuando un método está a punto de devolver el control al método que realiza la llamada desde un bloque **try / catch** por medio de un excepción no capturada o de una sentencia return explícita, también se ejecuta la cláusula **finall**, justo antes de que el método retorne el control. Esto puede ser útil para cerrar archivos o liberar cualquier recurso que se hubiera asignado al inicio de un método con la intensión de liberarlo antes de devolver el control.

La cláusula **finally** es opcional, sin embargo, cada sentencia **try** necesita por lo menos una cláusula **catch** o **finally**. A continuación presentamos el programa **Test13** donde se muestran tres métodos que terminan su ejecución de diferentes maneras.

```
class Test13{

    public static void main(String[] args){

            try     {
                    rutinaA();
            }
            catch ( Exception e ) {
                    System.out.println( “Excepcion Capturada” );
            }
            rutinaB();
            rutinaC();
            System.out.println( “FIN” );

    }
static void rutinaA() {
            try {
                    System.out.println( “Mensaje desde try de rutinaA” );
                    throw new RuntimeException( “rutinaA” );
            }
```

```
finally {
                System.out.println( "Mensaje desde finally de rutinaA"
);
            }
        }

        static void rutinaB(){
            try {
                System.out.println( "Mensaje desde try de rutinaB" );
                return;
            }
            finally {
                System.out.println( "Mensaje desde finally de rutinaB"
);
            }
        }

        static void rutinaC(){
            try {
                System.out.println( "Mensaje desde try de rutinaC" );
            }
            finally {
                System.out.println( "Mensaje desde finally de rutinaC"
);
            }
        }

    }
```

Compilación y ejecución:

```
C:\java2n100\cap12>javac Test13.java

C:\java2n100\cap12>java Test13
Mensaje desde try de rutinaA
Mensaje desde finally de rutinaA
Excepcion Capturada
Mensaje desde try de rutinaB
Mensaje desde finally de rutinaB
Mensaje desde try de rutinaC
Mensaje desde finally de rutinaC
FIN
```

En este programa, el método **rutinaA()** sale prematuramente del bloque **try** lanzando una excepción; la cláusula **finally** se ejecuta durante su salida. El método **rutinaB** sale del bloque **try**, a través de la instrucción **return**, la cláusula **finally** se ejecuta antes que el método devuelva el control. En el método **rutinaC**, el bloque **try** se ejecuta sin error; sin embargo, el bloque **finally** se ejecuta de todas maneras.

12.2.8 Creación de excepciones personalizadas

Aunque las excepciones que incorpora Java gestiona la mayoría de errores más comunes, es probable que se presenten situaciones, en las que no encontremos una excepción apropiada para cierto tipo de errores muy específicos de un proceso en particular. Por ejemplo, supongamos que debemos controlar el rango de una nota, en este caso de 0 a 20, no contamos con una excepción para este caso en particular.

En casos muy específicos, podemos crear nuestra propia excepción; para ello debemos crear una subclase de la clase **Exception**.

A continuación creamos la excepción **NotaFueraDeRango**:

```
public class NotaFueraDeRango extends Exception {

    public NotaFueraDeRango () {
            super( "Error: Nota debe estar en el rango de [0,20]." );
    }

    public NotaFueraDeRango( String msg ) {
            super(msg);
    }

}
```

Observe cómo se usa la excepción **NotaFueraDeRango**, el programa es **Test14**:

```
1.  class Test14 {
2.
3.          public static void main(String[] args) {
4.
5.                  try {
6.
7.                          Double nota1 = Double.parseDouble( args[0] );
8.                          Double nota2 = Double.parseDouble( args[1] );
9.                          Double prom;
10.                         prom = calculaPromedio( nota1, nota2 );
11.                         System.out.println( "Promedio: " + prom );
12.
13.                 }
14.                 catch( NotaFueraDeRango e1 ) {
15.
16.                         System.out.println( e1 );
17.
18.                 }
19.                 catch( Exception e2 ) {
20.
21.                         System.out.println( e2 );
22.
23.                 }
24.
25.                 System.out.println( "--- FIN ---" );
26.
27.         }
```

```
27.         }
28.
29.         static Double calculaPromedio( Double n1, Double n2 )
30.         throws NotaFueraDeRango {
31.
32.                 if (n1 < 0.0 | n1 > 20.0 ) {
33.                         throw new NotaFueraDeRango();
34.                 }
35.
36.                 if (n2 < 0.0 | n2 > 20.0 ) {
37.                         throw new NotaFueraDeRango( "Nota fuera de ran-
go." );
38.                 }
39.
40.                 return ( (n1+n2) / 2 );
41.
42.         }
43.
44. }
```

Este programa recibe dos notas en la línea de comando, y luego muestra el promedio de ambas notas.

En la compilación no tenemos problemas:

```
C:\java2n100\cap12>javac Test14.java [Enter]
```

Si ejecutamos el programa sin ningún parámetro, se produce la excepción de tipo **ArrayIndexOutOfBoundsException** en la línea 7, que es capturada en el segundo bloque **catch** en la línea 19.

```
C:\java2n100\cap12>java Test14       [Enter]
java.lang.ArrayIndexOutOfBoundsException: 0
--- FIN ---
```

Si ejecutamos el programa con dos notas en la línea de comando, pero la primera nota está fuera de rango, se lanza una excepción desde el método **calculaPromedio()**, específicamente en la línea 33, que es propagada hasta el método **main()**, donde se le da el tratamiento respectivo.

```
C:\java2n100\cap12>java Test14 25 15 [Enter]
Not aFueraDeRango: Error: Nota debe estar en el rango de [0,20].
--- FIN ---
```

Si ejecutamos el programa con dos notas en la línea de comando, pero la segunda nota está fuera de rango, se lanza una excepción desde el método **calculaPromedio()**,

específicamente en la línea 37, que es propagada hasta el método **main()** donde se le da el tratamiento respectivo.

```
C:\java2n100\cap12>java Test14 25 15 [Enter]
Not aFueraDeRango: Error: Nota debe estar en el rango de [0,20].
--- FIN ---
```

Pero si las notas son correctas, tenemos el resultado esperado:

```
C:\java2n100\cap12>java Test14 14 18 [Enter]
Promedio: 16.0
--- FIN ---
```

12.2.9 Excepciones encadenadas

Esta característica en el subsistema de excepciones, permite asociar una excepción a otra; la segunda excepción describe la causa de la primera. Por ejemplo, imagine una situación en la que un método lanza una excepción de tipo **ArithmeticException** a causa de un intento de división por cero, pero la causa real del problema es un error de E/S, que ocasiona que el divisor tome valor 0. Aunque el método debe lanzar la excepción **ArithmeticException**, porque ese es el error existente, es probable que el programador quiera saber también la causa subyacente del problema o sea un error de E/S. Las excepciones encadenadas permiten gestionar esta y otras situaciones en las que hay niveles de excepciones.

Para permitir que existan excepciones encadenadas, se añaden dos constructores y dos métodos a la clase **Throwable**.

Los constructores son los siguientes:

```
public Throwable ( Throwable cause )
public Throwable( String message, Throwable cause )
```

En el primer caso, **cause** es la excepción que provoca la excepción actual, es decir, la excepción subyacente, motivo de que exista una excepción.

En el segundo caso, permite especificar una descripción a través de **message**, y a la vez la excepción subyacente a través de **cause**. Estos dos constructores también se han añadido a las clases **Error, Exception** y **RuntimeException**.

Los métodos de excepciones encadenadas añadidos a **Throwable** son **getCause()** e **initCause()**. Tenemos el prototipo de estos métodos:

```
public Throwable getCause()
public Throwable initCause(Throwable cause)
```

El método **getCause()** devuelve la excepción subyacente en la actual excepción. Si no hay excepción subyacente devuelve **null**.

El método **nitCause()** asocia cause con la excepción actual y devuelve una referencia a la excepción. Por lo tanto, podemos asociar una excepción subyacente con una excepción después que esta es creada. Sin embargo, la excepción subyacente se puede asociar una sola vez, esto quiere decir que podemos llamar una sola vez por objeto a **initCause()**. Asimismo, si la excepción subyacente ha sido establecida mediante un constructor, no es posible volver a establecerla.

En general, **initCause()** se utiliza para establecer la excepción subyacente para la herencia de las clases de excepción que no soportan los dos constructores vistos anteriormente. La mayoría de excepciones incorporadas en Java, como **ArithmeticException**, no definen los constructores adicionales, por lo tanto, debemos utilizar el método **initCause()**.

Cuando creamos clases personalizadas, debemos considerar si es necesario incluir o no estos constructores. A continuación tenemos el programa **Test15** que es una variante del programa **Test14**, donde observamos que el método **calculaPromedio()** lanza una excepción de tipo **ArithmeticException** a la que se le añadió una excepción de tipo **NotaFueraDeRango**.

```
class Test15 {

    public static void main(String[] args) {

            try {
                    Double nota1 = Double.parseDouble( args[0] );
                    Double nota2 = Double.parseDouble( args[1] );
                    Double prom;
                    prom = calculaPromedio( nota1, nota2 );
                    System.out.println( "Promedio: " + prom );
            }
            catch( ArithmeticException e1 ) {
                    System.out.println( "Caso 1" );
                    System.out.println( "Excepcion: " + e1 );
                    System.out.println( "Causa: " + e1.getCause() );
            }
            catch( NotaFueraDeRango e2 ) {
                    System.out.println( "Caso 2" );
                    System.out.println( e2 );
            }
            catch( ArrayIndexOutOfBoundsException e3 ) {
                    System.out.println( "Caso 3" );
                    System.out.println( e3 );
            }
            catch( NumberFormatException e4 ) {
                    System.out.println( "Caso 4" );
                    System.out.println( e4 );
            }
            System.out.println( "--- FIN ---" );

    }
```

El método **getCause()** devuelve la excepción subyacente en la actual excepción. Si no hay excepción subyacente devuelve **null**.

El método **nitCause()** asocia cause con la excepción actual y devuelve una referencia a la excepción. Por lo tanto, podemos asociar una excepción subyacente con una excepción después que esta es creada. Sin embargo, la excepción subyacente se puede asociar una sola vez, esto quiere decir que podemos llamar una sola vez por objeto a **initCause()**. Asimismo, si la excepción subyacente ha sido establecida mediante un constructor, no es posible volver a establecerla.

En general, **initCause()** se utiliza para establecer la excepción subyacente para la herencia de las clases de excepción que no soportan los dos constructores vistos anteriormente. La mayoría de excepciones incorporadas en Java, como **ArithmeticException**, no definen los constructores adicionales, por lo tanto, debemos utilizar el método **initCause()**.

Cuando creamos clases personalizadas, debemos considerar si es necesario incluir o no estos constructores. A continuación tenemos el programa **Test15** que es una variante del programa **Test14**, donde observamos que el método **calculaPromedio()** lanza una excepción de tipo **ArithmeticException** a la que se le añadió una excepción de tipo **NotaFueraDeRango**.

```
class Test15 {

    public static void main(String[] args) {

            try {
                    Double nota1 = Double.parseDouble( args[0] );
                    Double nota2 = Double.parseDouble( args[1] );
                    Double prom;
                    prom = calculaPromedio( nota1, nota2 );
                    System.out.println( "Promedio: " + prom );
            }
            catch( ArithmeticException e1 ) {
                    System.out.println( "Caso 1" );
                    System.out.println( "Excepcion: " + e1 );
                    System.out.println( "Causa: " + e1.getCause() );
            }
            catch( NotaFueraDeRango e2 ) {
                    System.out.println( "Caso 2" );
                    System.out.println( e2 );
            }
            catch( ArrayIndexOutOfBoundsException e3 ) {
                    System.out.println( "Caso 3" );
                    System.out.println( e3 );
            }
            catch( NumberFormatException e4 ) {
                    System.out.println( "Caso 4" );
                    System.out.println( e4 );
            }
            System.out.println( "--- FIN ---" );

    }
```

```
}

    static Double calculaPromedio( Double n1, Double n2 )
    throws NotaFueraDeRango {

            ArithmeticException e;
            e = new ArithmeticException( "Error al calcular el promedio." );
            e.initCause( new NotaFueraDeRango() );

            if (n1 < 0.0 | n1 > 20.0 ) {
                    throw e;
            }

            if (n2 < 0.0 | n2 > 20.0 ) {
                    throw e;
            }

            return ( (n1+n2) / 2 );

    }

}
```

Compilación y ejecución

```
C:\java2n100\cap12>javac Test15.java [Enter]

C:\java2n100\cap12>java Test15 23 12 [Enter]
Caso 1
Excepcion: java.lang.ArithmeticException: Error al calcular el promedio.
Causa: NotaFueraDeRango: Error: Nota debe estar en el rango de [0,20].
--- FIN ---
```

Cuando ejecutamos el programa, la primera nota está fuera del rango permitido; motivo por el cual se genera la excepción de tipo **ArithmeticException**, pero esta tiene asociada la excepción **NotaFueraDeRango**.

Tipos enumerados

CAPÍTULO 13

Desde las primeras versiones de Java, muchos programadores extrañaban esta característica, que sí tienen otros lenguajes. Incluso se puede encontrar información en Internet que nos muestra cómo construir datos enumerados en Java, pero para beneplácito de todos los programadores de Java, la versión 5 ya incorpora esta característica.

Un tipo enumerado define un conjunto de valores constantes, de tal manera que pueda usarse para definir variables, entre otras cosas.

13.1 FUNDAMENTOS

Un tipo enumerado define una lista de constantes con nombre. Aunque Java ofrece otras características que de forma similar aportan esta funcionalidad como la variable **final**; esta es una característica bastante esperada sobre todo porque muchos lenguajes si la tienen implementada. A partir de Java2 Versión 5 ya contamos con esta característica.

A diferencia de otros lenguajes, en Java una enumeración define un tipo de clase. Introduciendo las enumeraciones en las clases, el concepto se ha extendido de forma considerable; así tenemos que las enumeraciones pueden tener constructores, métodos y variables de instancia.

Para crear tipos enumerados debemos considerar tres componentes básicos como mínimo:

- La palabra clave **enum**
- El nombre del nuevo tipo
- Una lista de valores para el nuevo tipo

Hay varios componentes opcionales que se pueden definir también:

- Una interfaz o conjunto de interfaces que **enum** implementa
- Definición de variables
- Definición de métodos

13.2 USAR DATOS ENUMERADOS

13.2.1 PALABRA CLAVE ENUM

Para definir un tipo enumerado se utiliza la palabra clase **enum**.

Sintaxis

```
enum Tipo_Enumerado {
    Lista_de_Datos
}
```

Donde:

Tipo_Enumerado	Es el nombre del tipo de dato enumerado.
Lista_de_Datos	Son los datos que define el nuevo tipo de dato enumerado.

Como ejemplo definiremos un tipo enumerado para los días de la semana:

DíasSemana.java

```
enum DiaSemana{
    DOMINGO, LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO
}
```

Cada identificador **DOMINGO**, **LUNES**, etc., recibe el nombre de **constante de enumeración**. Cada una se declara implícitamente como *public*, *static* y **final**, miembro de **DiaSemana**.

En la mayoría de casos solo necesitamos la funcionalidad básica de los tipos enumerados.

La convención es utilizar mayúsculas para los valores de un tipo enumerado. **DiaSemana** será usado como otro tipo de dato de Java.

13.2.2 USO DE TIPOS ENUMERADOS

Después de definir un tipo enumerado, podemos crear variables de este tipo. Sin embargo, incluso cuando una enumeración define un tipo de clase, no hay que crear

una instancia con el operador **new**. En su lugar, solo debemos declarar y usar variables del mismo modo que lo hacemos con los tipos simples. A continuación declaramos la variable **dia** del tipo **DiaSemana**:

```
DiaSemana dia;
```

Como **día** es de tipo **DiaSemana**, los únicos valores que puede tomar son aquellos que define el tipo enumerado **DiaSemana**. Por ejemplo, la siguiente sentencia le asigna el valor JUEVES:

```
dia = DiaSemana.JUEVES;
```

Podemos utilizar el operador == para realizar comparaciones. En la siguiente sentencia comparamos si la variable **dia** tiene valor **SABADO**:

```
 if ( dia == DiaSemana.SABADO ) {

     // Implementación

}
```

Un valor enumerado también puede utilizarse para controlar una instrucción **switch**. En este caso, las cláusulas **case** deben utilizar constantes del mismo tipo enumerado empleado por la expresión del **switch**.

Observe que la siguiente instrucción **switch** es correcta:

```
switch ( dia ) {
    case DOMINO:

            // . . .

    case LUNES:

            // . . .

    case MARTES:

            // . . .

    . . .
    . . .

}
```

Note que no es necesario calificar las constantes en las cláusulas **case**, porque el tipo de la expresión del **switch** (en este caso la variable **dia**), especifica implícitamente el tipo. Si calificamos las constantes de la cláusula case generará un error de compilación.

Cuando se visualiza una constante enumerada con el método **println( )**, se muestra su nombre. Por ejemplo:

```
System.out.println( dia );
```

En este caso, muestra **JUEVES**.

El siguiente programa ilustra lo explicado:

prog1301.java

```
class prog1301 {

    public static void main(String[] args)
    {

            DiaSemana dia;
            dia = DiaSemana.MARTES;

            System.out.println( "El valor de dia es: " + dia );
            System.out.println();

            if ( dia == DiaSemana.JUEVES ) {
                    System.out.println( "Es dia JUEVES" );
            } else {
                    System.out.println( "No es dia JUEVES" );
            }
            System.out.println( );

            switch ( dia ) {
                    case SABADO: case DOMINGO:
                            System.out.println( "Es dia de Playa" );
                            break;
                    default:
                            System.out.println( "Es dia laborable" );
            }
            System.out.println( );

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap13>javac prog1301.java        [Enter]

C:\java2n100\cap13>java prog1301              [Enter]
El valor de dia es: MARTES

No es dia JUEVES

Es dia laborable
```

13.2.3 Métodos values() y valueOf()

Todos los tipos enumerados contienen automáticamente dos métodos predefinidos: **values()** y **s**. Sus prototipos son:

```
public static tipo_enumerado[] values()

public static tipo_enumerado valueOf( String name )
```

A continuación, presentamos una descripción de estos métodos:

values()	Retorna un arreglo que contiene una lista de las constantes del tipo enumerado.
valueOf()	Retorna un valor enumerado que coincide con la cadena que se le pasa como parámetro en la cadena **name**.

En ambos casos **tipo_enumerado** es el tipo enumerado al que hacemos referencia. Por ejemplo, podría ser **DiaSemana** que desarrollamos anteriormente en este capítulo.

Véase en el siguiente ejemplo el uso de **values()** y **valueOf()**:

prog1302.java

```
class prog1302 {

    public static void main(String[] args)
    {

            System.out.println( "Listado de todas las constantes de DiaSemana:" );
            DiaSemana elementos[] = DiaSemana.values();
            for( DiaSemana ele: elementos ){
                    System.out.println( ele );
            }
            System.out.println( );

            DiaSemana dia;
            dia = DiaSemana.valueOf( "LUNES" );
            System.out.println( "dia = " + dia );

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap13>javac prog1302.java                    [Enter]

C:\java2n100\cap13>java prog1302                          [Enter]
Listado de todas las constantes de DiaSemana:
DOMINGO
LUNES
MARTES
MIERCOLES
JUEVES
VIERNES
SABADO

dia = LUNES
```

En este ejemplo se utiliza el bucle **for** de estilo **for-each** para hacer un bucle, a través de todos los elementos del arreglo que se obtiene con **values()**, cuyos valores se han guardado en la variable **elementos**. Sin embargo, este paso no es necesario, puesto que el bucle **for** se puede escribir de la siguiente manera:

```
for( DiaSemana ele: DiaSemana.values() ) {
    System.out.println( ele );
}
```

El parámetro que se le pasa **a valueOf()** debe coincidir con la definición de la constante, en este caso se le pasa LUNES; si se escribe en minúsculas se obtendrá un error en compilación.

13.2.4 Tipos enumerados en línea

Se pueden definir tipos enumerados en la misma clase, el esquema es el siguiente:

```
public class NombreClase {

    public enum TipoEnumerado { Lista_de_Constantes };

    // Implementación

}
```

Los tipos enumerados son por defecto estáticos, por lo tanto pueden utilizarse sin necesidad de instanciar la clase, tal como se ilustra a continuación.

Definiremos la clase **CMeses**, donde se ha definido el tipo enumerado **Meses** que define todos los meses del año.

CMeses.java

```
class prog1302 {

    public static void main(String[] args)
    {

            System.out.println( "Listado de todas las constantes de DiaSema-
na:" );
            DiaSemana elementos[] = DiaSemana.values();
            for( DiaSemana ele: elementos ){
                    System.out.println( ele );
            }
            System.out.println( );

            DiaSemana dia;
            dia = DiaSemana.valueOf( "LUNES" );
            System.out.println( "dia = " + dia );

    }

}
```

El método **mostrarMeses()** imprime un listado de los valores del tipo enumerado **Meses**. Los otros métodos **setDato()** y **getDato()** son para acceder a la variable privada **dato**.

El programa **prog1303** ilustra cómo se utiliza la clase **CMeses** y el tipo enumerado en ella definida:

prog1303.java

```
1.          class prog1303 {
2.
3.                  public static void main(String[] args)
4.                  {
5.
6.                          CMeses.mostrarMeses();
7.
8.                          CMeses obj = new CMeses();
9.                          obj.setDato( CMeses.Meses.JULIO );
10.                         System.out.println( "obj = " + obj.getDato() );
11.
12.                 }
13.
14.         }
```

Compilación y ejecución:

```
C:\java2n100\cap13>javac prog1303.java        [Enter]

C:\java2n100\cap13>java prog1303              [Enter]
---Listado de Meses---
ENERO
FEBRERO
MARZO
ABRIL
MAYO
JUNIO
JULIO
AGOSTO
SEPTIEMBRE
OCTUBRE
NOVIEMBRE
DICIEMBRE
--- F I N ---
obj = JULIO
```

En la línea 6 del programa, ejecutamos el método **mostrarMeses()** de la clase **CMeses**, sin necesidad de instanciar la clase porque se trata de un método estático, también llamado método de clase.

En la línea 8 se crea una instancia de la clase **CMeses**, a la que accedemos a través de la variable **obj**.

En la línea 9 se establece el valor de la variable dato, a través del método **setDato()**. Note que se está accediendo al tipo enumerado **Meses** mediante la clase **CMeses**, esto es posible porque el tipo enumerado en la clase es por defecto estático.

En la línea 10 se imprime el valor de la variable dato, su valor se lee a través del método **getDato()**.

13.3 LOS TIPOS ENUMERADOS SON CLASES

13.3.1 CONSTRUCTORES Y MÉTODOS

Como ya se explicó, una enumeración de Java es un tipo de clase. Aunque no se realice una instancia a **enum** utilizando **new**, esta tiene las mismas capacidades que otras clases.

Por ejemplo, se le puede dar constructores, añadir variables de instancia y métodos; incluso implementar interfaces. Es importante comprender que cada constante de enumeración es un objeto de su tipo; por lo tanto, cuando se define un constructor para un tipo enumero, se lo invoca cuando se crea cada constante de enumeración, es decir tiene su propia copia.

Por ejemplo, veamos el siguiente caso, tenemos un tipo enumerado de nombre **Meses** que define todos los meses del año:

Meses.java

```
enum Meses{

    ENERO(1),
    FEBRERO(2),
    MARZO(3),
    ABRIL(4),
    MAYO(5),
    JUNIO(6),
    JULIO(7),
    AGOSTO(8),
    SEPTIEMBRE(9),
    OCTUBRE(10),
    NOVIEMBRE(11),
    DICIEMBRE(12);

    private int id;

    Meses( int id ){
            this.id = id;
    }


    int getId(){
            return id;
    }

    public int getTrimestre(){
            int rpta = 0;
            switch (this) {
                    case ENERO: case FEBRERO: case MARZO:
                            rpta = 1;
                            break;
                    case ABRIL: case MAYO: case JUNIO:
                            rpta = 2;
                            break;
                    case JULIO: case AGOSTO: case SEPTIEMBRE:
                            rpta = 3;
                            break;
                    case OCTUBRE: case NOVIEMBRE: case DICIEMBRE:
                            rpta = 4;
                            break;
            }
            return rpta;
    }

}
```

Este tipo enumerado, también define su número de mes y mediante los métodos **getId()** y **getTrimestre()** podemos obtener el número de mes y el trimestre al que pertenece cada constante de enumeración, respectivamente.

El programa **prog1304** ilustra cómo podemos utilizar estas características de los tipos enumerados, específicamente del tipo **Meses**.

prog1304.java

```
class prog1304 {

    public static void main(String[] args)
    {

            // Mostrar datos del mes de ABRIL
            System.out.println( "Mes: Abril" );
            System.out.println( "Nro. de Mes: " + Meses.ABRIL.getId() );
            System.out.println(     "Trimestre: " + Meses.ABRIL.getTrimes-
tre() );

            // Mostrar todos los meses
            System.out.println();
            System.out.println("Nro. Mes                    Trimestre" );
            for( Meses obj : Meses.values() ) {
                    System.out.println( obj.getId() + "\t" + obj + "\t" +
obj.getTrimestre() );
            }

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap13>javac prog1304.java        [Enter]

C:\java2n100\cap13>java prog1304              [Enter]
Mes: Abril
Nro. de Mes: 4
Trimestre: 2

Nro. Mes                    Trimestre
1           ENERO           1
2           FEBRERO         1
3           MARZO           1
4           ABRIL           2
5           MAYO            2
6           JUNIO           2
7           JULIO           3
8           AGOSTO          3
9           SEPTIEMBRE      3
10          OCTUBRE         4
11          NOVIEMBRE       4
12          DICIEMBRE       4
```

13.3.2 Implementación de interfaces

También es posible definir una interfaz y luego implementarla en uno, dos o más tipos enumerados. Es posible tener interfaces que sean implementadas, no solo por enumeraciones sino por clases. Es muy simple porque los tipos enumerados son clases que heredan de **java.lang.enum** implícitamente, y como toda clase puede implementar interfaces, solo se debe agregar la cláusula **implements**.

Por ejemplo, tenemos la interfaz **IBasica** que define dos métodos que deben ser implementados por todas las enumeraciones:

getId()	Para obtener el código asociado con el valor enumerado.
getDescripcion()	Para obtener una descripción del valor enumerado.

A continuación, indicamos el código de **IBasica**:

IBasica.java

```
interface IBasica {

    int getId();

    String getDescripcion();

}
```

Luego se define la enumeración **Puestos** que refiere a los puestos de trabajo de una empresa, pero además, cada valor enumerado tiene asociado un código, un sueldo y podemos obtener una descripción de cada puesto.

Observe el código de la enumeración **Puestos**:

Puestos.java

```
enum Puestos implements IBasica {

    GERENTE( 10, 15000 ),
    EMPLEADO( 20, 4000 ),
    ANALISTA( 30, 6000 ),
    PROGRAMADOR( 40, 5000 );

    private int id;
    private int sueldo;

    Puestos( int id, int sueldo ){
            this.id = id;
            this.sueldo = sueldo;
    }
```

```
private int id;
    private int sueldo;

    Puestos( int id, int sueldo ){
            this.id = id;
            this.sueldo = sueldo;
    }

    public int getId(){
            return id;
    }

    int getSueldo(){
            return sueldo;
    }

    public String getDescripcion(){
            String rpta = "";
            switch (this) {
                    case GERENTE:
                            rpta = "Encargado de dirigir un area de la em-
presa";
                            break;
                    case EMPLEADO:
                            rpta = "Realizar las tareas rutinarias de la em-
presa";
                            break;
                    case ANALISTA:
                            rpta = "Desarrolla el analisis de los nuevos
sistemas";
                            break;
                    case PROGRAMADOR:
                            rpta = "Desarrolla los nuevos sistemas";
                            break;
            }
            return rpta;
    }

}
```

El programa **prog1305** señala cómo se utiliza la enumeración **Puestos**. El código es el siguiente:

prog1305.java

```
class prog1305 {

    public static void main(String[] args)
    {

            // Mostrar datos del puesto PROGRAMADOR
            System.out.println( "Puesto: Programador" );
            System.out.println( "Codigo: " + Puestos.PROGRAMADOR.getId() );
            System.out.println(       "Sueldo: " + Puestos.PROGRAMADOR.getSueldo() );
            System.out.println(       "Descripcion: " + Puestos.PROGRAMADOR.ge-
tDescripcion() );
```

```
// Mostrar todos los puestos
            System.out.println();
            System.out.println( "Todos los puestos" );
            for( Puestos obj : Puestos.values() ) {
                    System.out.println();
                    System.out.println( "Puesto: " + obj );
                    System.out.println( "Codigo: " + obj.getId() );
                    System.out.println(     "Sueldo: " + obj.getSueldo() );
                    System.out.println(     "Descripcion: " + obj.getDescrip-
cion() );
            }

      }

}
```

Compilación y ejecución:

```
C:\java2n100\cap13>javac prog1305.java        [Enter]

C:\java2n100\cap13>java prog1305              [Enter]
Puesto: Programador
Codigo: 40
Sueldo: 5000
Descripcion: Desarrolla los nuevos sistemas

Todos los puestos

Puesto: GERENTE
Codigo: 10
Sueldo: 15000
Descripcion: Encargado de dirigir un area de la empresa

Puesto: EMPLEADO
Codigo: 20
Sueldo: 4000
Descripcion: Realizar las tareas rutinarias de la Empresa

Puesto: ANALISTA
Codigo: 30
Sueldo: 6000
Descripcion: Desarrolla el analisis de los nuevos sistemas

Puesto: PROGRAMADOR
Codigo: 40
Sueldo: 5000
Descripcion: Desarrolla los nuevos sistemas
```

Autoboxing y Unboxing

CAPÍTULO 14

Uno de los problemas que tenemos cuando trabajamos con clases envolventes, es la conversión explícita que debemos hacer cada vez que queremos asignarlos a su tipo primitivo equivalente o viceversa. Pues este problema en la presente versión de Java ya fue resuelto, y el presente capítulo trata precisamente de este tema y sus implicancias.

14.1 CLASES ENVOLVENTES

Java utiliza tipos simples como **int** o **double** para definir los tipos básicos de datos que proporciona el lenguaje. Los tipos simples se utilizan en lugar de objetos por razones de rendimiento. Si se utilizan objetos para estos valores, se añadirá una sobrecarga inaceptable incluso en los cálculos más sencillos. Por lo tanto, los tipos simples no son parte de la jerarquía de objetos y no heredan de **Object**.

A pesar de la ventaja en el rendimiento que aportan los tipos simples, hay ocasiones en que se necesita una representación de objeto. Por ejemplo, muchas de las estructuras estándares de datos que implementa Java funcionan sobre objetos, lo que significa que no se pueden utilizar estas estructuras de datos para almacenar tipos simples. Como una solución al manejo de estas y otras situaciones, Java proporciona unas clases envolventes (*wrapper*) de tipos que encapsulan un tipo simple dentro de un objeto.

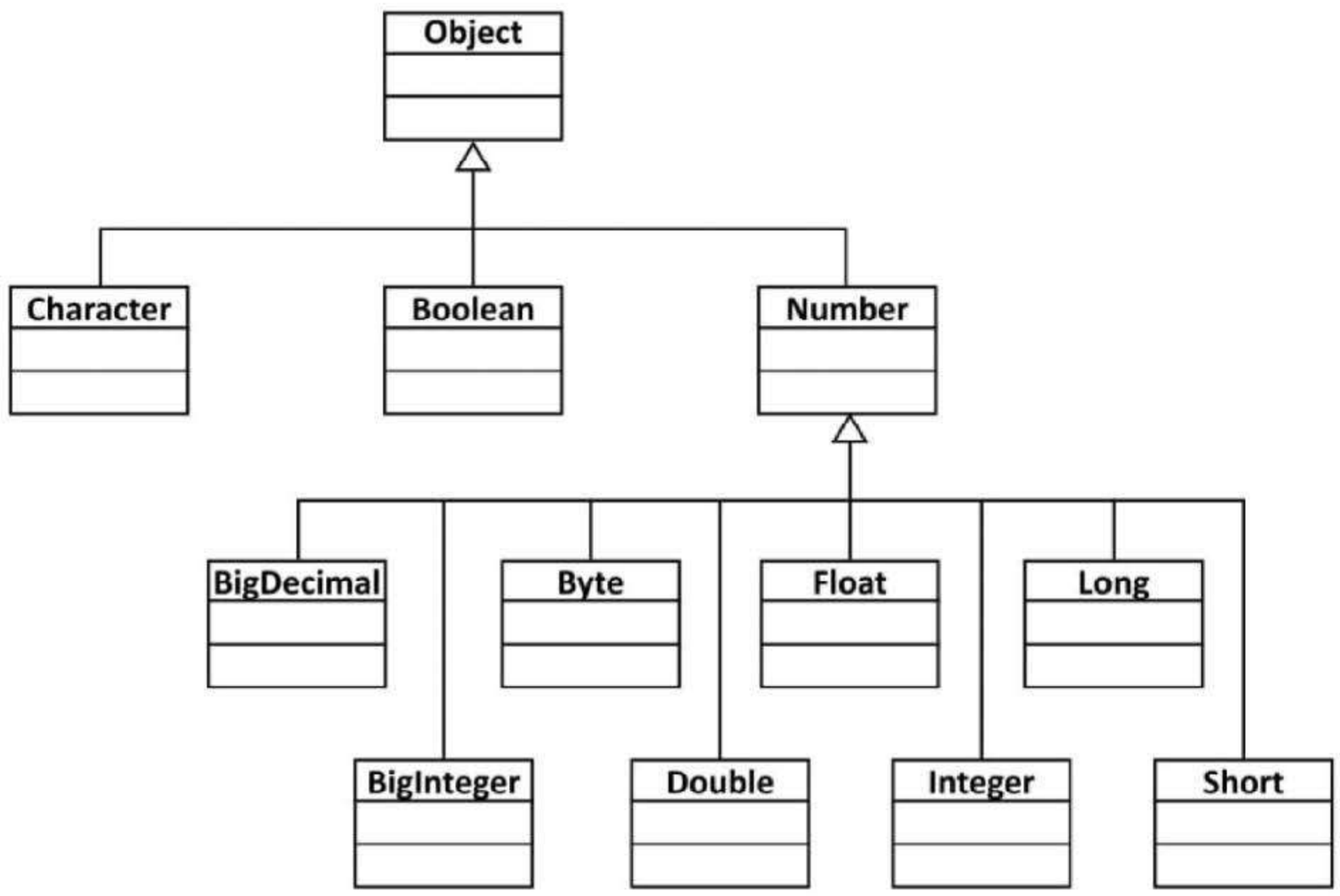

Fig. 14.1 Diagrama de clases de las clases envolventes

La clase **Object** es la raíz de todas las clases heredadas. Cada clase tiene a **Object** como una superclase. Todos los objetos, incluidos los arreglos, implementan los métodos de esta clase.

En la Fig. 14.1 se muestra el diagrama de las clases envolventes que son ejemplos de clases inmutables. Por definición, no se puede alterar el valor de un objeto inmutable después de su construcción.

14.1.1 Clase number

La clase abstracta **Number** es la superclase de las clases **BigDecimal**, **BigInteger**, **Byte**, **Double**, **Float**, **Integer**, **Long** y **Short**.

Las subclases de **Number** proporcionan métodos para convertir un valor numérico representado por un objeto a **byte**, **double**, **float**, **int**, **long** y **short**.

14.1.2 Double y Float

Double y **Float** son clase envolventes para valores de coma flotante de tipo **double** y **float**, respectivamente.

Los constructores de **Float** son los siguientes:

```
public Float(float value)
public Float(double value)
public Float(String s) throws NumberFormatException
```

Observe que los objetos **Float** se pueden construir con valores tipo **float** o **double** y a partir de la representación como cadena de un número de coma flotante.

El programa **prog1401** ilustra cómo se puede utilizar la clase envolvente **Float**.

El código es el siguiente:

prog1401.java

```
class prog1401 {

    public static void main(String[] args) {

            Float n1 = new Float( 13.5 );
            Float n2 = new Float( 16.8f );
            Float n3 = new Float( "25.45" );
            float n4 = Float.parseFloat( "15.75" );

            System.out.println("n1 = " + n1.toString() );
            System.out.println("n2 = " + n2.toString() );
            System.out.println("n3 = " + n3.toString() );
            System.out.println("n4 = " + n4 );

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap14>javac prog1401.java        [Enter]

C:\java2n100\cap14>java prog1401              [Enter]
n1 = 13.5
n2 = 16.8
n3 = 25.45
n4 = 15.75
```

Los constructores de **Double** son los siguientes:

```
public Double(double value)

public Double(String s) throws NumberFormatException
```

Los objetos **Double** pueden construirse con valores **double** o una cadena que contenga el valor de coma flotante.

El programa **prog1402** ilustra cómo se puede utilizar la clase envolvente **Double**.

El código es el siguiente:

prog1402.java

```
class prog1402 {

    public static void main(String[] args) {

            Double n1 = new Double( 18.55 );
            Double n2 = new Double( "37.85" );
            double n3 = Double.parseDouble( "77.77" );

            System.out.println("n1 = " + n1.toString() );
            System.out.println("n2 = " + n2.toString() );
            System.out.println("n3 = " + n3 );

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap14>javac prog1402.java        [Enter]

C:\java2n100\cap14>java prog1402              [Enter]
n1 = 18.55
n2 = 37.85
n3 = 77.77
```

14.1.3 Métodos isInfinite() e isNaN()

Float y **Double** proporcionan los métodos **isInfinite()** e **isNaN()**, que pueden ser útiles cuando manipulamos dos valores especiales de **double** y **float**. Estos métodos prueban dos valores únicos definidos por la especificación de coma flotante de IEEE: **infinity** y **NaN**.

Presentamos una descripción de los métodos:

isInfinite()	Devuelve el valor **true** si el valor que se prueba es infinitamente grande o pequeño en cuanto a magnitud.
isNaN()	Devuelve el valor **true** si el valor que se prueba no es un número.

El programa **prog1403** muestra cómo usar estos métodos. El código es el siguiente:

prog1403.java

```
class prog1403 {

    public static void main(String[] args) {

            Double n1 = new Double( 15/0. );
            Double n2 = new Double( 0/0. );

            System.out.println( n1 + ": " + n1.isInfinite() + ", " + n1.isNaN()
);
            System.out.println( n2 + ": " + n2.isInfinite() + ", " + n2.isNaN()
);

    }

}
```

Compilación y ejecución;

```
C:\java2n100\cap14>javac prog1403.java        [Enter]

C:\java2n100\cap14>java prog1403              [Enter]
Infinity: true, false
NaN: false, true
```

14.1.4 Byte, Short, Integer y Long

Son clases envolventes de los tipos **byte**, **short**, **int** y **long**, respectivamente.

- Los constructores para la clase **Byte** son:

```
public Byte(byte value)

public Byte(String s) throws NumberFormatException
```

- Los constructores para la clase **Short** son:

```
public Short(short value)

public Short(String s) throws NumberFormatException
```

- Los constructores para la clase **Integer** son:

```
public Integer(int value)

public Integer(String s) throws NumberFormatException
```

- Los constructores para la clase **Long** son:

```
public Long(long value)

public Long(String s) throws NumberFormatException
```

Véase que estos objetos se pueden construir a partir de valores numéricos o cadenas que contengan valores válidos de números enteros.

El programa **prog1404** ilustra cómo utilizar la clase **Integer**; del mismo modo pueden utilizarse las otras clases.

prog1404.java

```
class prog1404 {

    public static void main(String[] args) {

            Integer n1 = new Integer( 15 );
            Integer n2 = new Integer( "18" );
            int n3 = n1.intValue() + n2.intValue();

            System.out.println( "n1: " + n1 );
            System.out.println( "n2: " + n2 );
            System.out.println( "n3: " + n3 );

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap14>javac prog1404.java        [Enter]

C:\java2n100\cap14>java prog1404              [Enter]
n1: 15
n2: 18
n3: 33
```

14.1.5 Character

Character es la clase envolvente para el tipo **char**. Su constructor es el siguiente:

```
Character(char value)
```

Donde **value** especifica el carácter que envolverá el objeto **Character** que se crea. Para obtener el valor **char** que se contiene en el objeto **Character**, aplicamos el método **charValue()**.

El programa **prog1405** ilustra cómo utilizar la clase **Character** y sus diversos métodos.

prog1405.java

```
class prog1405 {

    public static void main(String[] args) {

            char lista[] = { 'A', 'b', '9', '?', ' ', 'M' };
        String cadena = "";
        for( int k = 0; k < lista.length; k++ ){

            if( Character.isDigit( lista[k] ) ) {
                System.out.println( lista[k] + " es un digito" );
            }
            if( Character.isLetter( lista[k] ) ) {
                System.out.println( lista[k] + " es una letra" );
            }
            if( Character.isWhitespace( lista[k] ) ) {
                System.out.println( lista[k] + " es un espacio en blanco" );
            }
            if( Character.isUpperCase( lista[k] ) ) {
                System.out.println( lista[k] + " es una mayuscula" );
            }
            if( Character.isLowerCase( lista[k] ) ) {
                System.out.println( lista[k] + " es una minuscula" );
            }

        }

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap14>javac prog1405.java        [Enter]

C:\java2n100\cap14>java prog1405              [Enter]
A es una letra
A es una mayuscula
b es una letra
b es una minuscula
9 es un digito
  es un espacio en blanco
M es una letra
M es una mayuscula
```

14.1.6 Boolean

La clase **Boolean** envuelve un valor primitivo **boolean** en un objeto. Un objeto del tipo **Boolean** contiene un solo campo de tipo **boolean**.

Además, esta clase proporciona muchos métodos para convertir un **boolean** a **String** y un **String** a **boolean**; así como otras constantes y métodos útiles al tratar un **boolean**.

Los constructores para esta clase son:

```
public Boolean(boolean value)

public Boolean(String s)
```

El programa **prog1406** muestra una forma de cómo se utiliza la clase **Boolean**. **prog1406.java**.

class prog1406 {

```
class prog1404 {

    public static void main(String[] args) {

            Integer n1 = new Integer( 15 );
            Integer n2 = new Integer( "18" );
            int n3 = n1.intValue() + n2.intValue();

            System.out.println( "n1: " + n1 );
            System.out.println( "n2: " + n2 );
            System.out.println( "n3: " + n3 );

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap14>javac prog1406.java        [Enter]

C:\java2n100\cap14>java prog1406              [Enter]
Son Diferentes
```

Si en el programa cambiamos el valor de la variable **ok2** a **true**, y compilamos y ejecutamos nuevamente el programa, tendríamos el mensaje: "son **iguales**".

14.2 AUTOBOXING

Es el proceso por el cual un tipo simple se encapsula de forma automática dentro de su envoltorio equivalente cuando se necesita un objeto de ese tipo; de manera que no es necesario construir un objeto de forma explícita.

En versiones anteriores, para crear un objeto de tipo **Integer**, la sentencia que se usaba era:

```
Integer num = new Integer( 15);
```

En este caso el valor de **num** es 15.

En esta nueva versión ya no es necesario utilizar el operador **new**, porque la característica de **autoboxing** encapsula automáticamente un tipo **int** en su envoltorio equivalente; en el caso **Integer**, la instrucción quedaría así:

```
Integer num = 15;
```

También es válida la siguiente instrucción:

```
int n = 15;
Integer num = n;
```

En estos ejemplos se ha utilizado la clase **Integer**, pero lo mismo sucede con las otras clases envolventes.

14.3 UNBOXING

Es el proceso en donde el valor de un objeto encapsulado se extrae de forma automática del envoltorio cuando se necesita su valor; de manera que no es necesario llamar al método respectivo para extraer su valor.

En versiones anteriores, por ejemplo a datos de tipo **int,las instrucciones correctas** son:

```
Integer num = new Integer(18);
int n = num.intValue();
```

En esta nueva versión, las cosas se han simplificado, las instrucciones son las siguientes:

```
Integer num = 18;
int n = num;
```

En estos breves ejemplos se ha utilizado la clase **Integer**, pero lo mismo sucede con las otras clases envolventes.

14.4 OPERACIONES CON TIPOS ENVOLVENTES

14.4.1 INCREMENTO Y DECREMENTO CON TIPOS ENVOLVENTES

Cuando se piensa en las implicaciones del **autoboxing** y **unboxing**, notará que son de gran envergadura. Pues cada operación disponible para un tipo primitivo debe ser accesible para su contraparte del tipo envolvente y viceversa. Los usos inmediatos son las operaciones de incremento y decremento: ++ y --, ambas trabajan también con tipos envolventes.

El programa **prog1407** ilustra cómo se utiliza estos operadores.

A continuación tenemos el listado del programa **prog1407**:

class prog1407 .java

```
class prog1407 {

    public static void main(String[] args) {

            Integer i = 1;

            while (true){
                    System.out.printf( "%d al cuadrado es %d%n", i, i*i++ );
                    if (i>5) break;
            }

    }

}
```

La variable **n** se trata como un **int** en este ejemplo.

Compilación y ejecución:

```
C:\java2n100\cap14>javac prog1407.java        [Enter]

C:\java2n100\cap14>java prog1407              [Enter]
1 al cuadrado es 1
2 al cuadrado es 4
3 al cuadrado es 9
4 al cuadrado es 16
5 al cuadrado es 25
```

14.4.2 BOOLEAN VERSUS BOOLEAN

El tipo **boolean** es de un solo bit, es un caso especial de los tipos primitivos de Java; sobre todo porque tiene varios operadores lógicos asociados a él, por ejemplo: **! (not)**, **|| (or)**, **&& (and)**. Con unboxing, estos son útiles también para los valores **Boolean**.

En algún momento podemos tener expresiones que utilicen: **!**, **||**, o **&&**, con valores primitivos de tipo **boolean** que son extraídos de valores tipos **Boolean**.

Por ejemplo, el siguiente código escrito en versiones anteriores:

```
Boolean caso1 = new Boolean("TRUE");
Boolean caso2 = new Boolean("TRUE");
boolean caso3 = false;

Boolean rpta  = new Boolean( (caso1.booleanValue() || caso2.booleanValue()) &&
caso3 );
```

En la nueva versión, el código quedaría así:

```
Boolean caso1 = true;
Boolean caso2 = true;
boolean caso3 = false;

Boolean rpta  = (caso1 || caso2) && caso3;
```

En este caso, la respuesta es de tipo **boolean**, y es encapsulada dentro de un objeto de tipo **Boolean**.

14.4.3 Condicionales y Unboxing

Una de las características más singulares de Java es el operador condicional, también llamado operador ternario. Esta es la versión de una instrucción **if/else** en operador, representada en este caso por el carácter **?** Puesto que evalúa una expresión, las características *unboxing* de Java la afectan; además se puede utilizar con todas las clases de nuevos tipos.

La sintaxis del operador condicional es:

```
(Expresión Condicional) ? expresión1 : expresión2
```

Si **(Expresión Condicional)** es evaluada como **true**, entonces **expresión1** es ejecutada; en caso contrario se ejecuta **expresión2**. En versiones anteriores de Java, la **Expresión Condicional** tiene que dar un resultado de tipo **boolean**. Esto era un problema cuando se tenía un método que retorna un tipo **Boolean** o una expresión que encapsula a un **boolean**. En la nueva versión de Java, esto ya no es un problema.

Por ejemplo, el siguiente código:

```
Boolean caso1 = false;
Boolean caso2 = true;

String cad = (caso1)?(caso2?"Ingresas":"Falta Poco"):"Falta Mucho";
System.out.println(cad);
```

El resultado tiene los siguientes casos:

1. Si **caso1** y **caso2** son **true**, el resultado es **Ingresas**.
2. Si **caso1** es **true** y **caso2** es **false**, el resultado es **Falta Poco**.
3. Si **caso1** es **false**, el resultado es **Falta Mucho**, es indiferente del valor de **caso2**.

14.4.4 Resolución de métodos sobrecargados

Boxing y *unboxing* ofrece muchas alternativas a problemas comunes en la programación de Java. Sin embargo, estas soluciones introducen algunas excepciones, particularmente en el área de resolución de métodos.

La resolución de un método es el proceso por el cual el compilador de Java determina qué método se está invocando. Se debe tener cuidado porque boxing y unboxing pueden afectar este proceso.

En un caso normal, Java maneja la resolución de un método usando el nombre del método. En los casos donde se sobrecarga un método deberá tomarse una medida adicional. Los argumentos del método son examinados y se emparejan hacia arriba con los argumentos de una versión específica que el método acepte. Si no se encuentra ninguna lista de argumentos que empareja, el compilador reporta un error de compilación.

Consideremos los siguientes métodos:

```
public void hacerAlgo( double num );

public void hacerAlgo( Integer num );
```

Ahora supongamos que invocamos al método de la siguiente manera:

```
int n = 1;
hacerAlgo( n );
```

- *¿Qué método es invocado?*
  En una versión anterior de Java es fácil de determinar. El **int** se extiende un **double** y se invoca a **hacerAlgo(double num)**. Sin embargo, en la versión actual parece que ocurre un *boxing*, y **hacerAlgo(Integer num)** resolvería la invocación del método. Sin embargo, no sucede así.

  Imagínese escribir un programa como este, compilar y probarlo en Java 1,4, y luego recompilarlo en la versión actual. ¡Repentinamente, las cosas comienzan a desordenarse! Por esa razón, la resolución del método en la versión actual seleccionará siempre el mismo método que seleccionó en Java 1,4.

En general, debe evitar sobrecargar métodos de esta forma, a menos que sea necesario. Sea tan específico como sea posible al momento de dar nombre a sus métodos y argumentos.

En Java debido a estas restricciones, la resolución del método es un proceso de dos pasos:

1. El compilador procura localizar el método correcto sin ningún boxing ni unboxing. Por ello encontrará cualquier método que hubiera sido invocado bajo las reglas de Java 1,4.
2. Si el primer paso falla, el compilador intentará la resolución del método otra vez, pero esta vez permitiendo conversiones *boxing* y *unboxing*.

Hilos (Threads)

CAPÍTULO 15

Muchas veces necesitamos ejecutar una secuencia de instrucciones, pero mientras se ejecuta queremos hacer otras tareas. Esto se logra programando hilos (Threads). En este capítulo aprenderá a crear hilos y cómo sincronizar su ejecución.

15.1 ¿QUÉ SON HILOS?

La Máquina Virtual Java (JVM) es un sistema multihilo, es decir, es capaz de ejecutar varias secuencias de instrucciones (rutinas) simultáneamente. La JVM tramita todos los detalles, asignación de tiempos de ejecución, prioridades, etc., de forma similar a un sistema operativo, ya que gestiona múltiples procesos (programas). La diferencia básica entre ambos es que los **Hilos** corren dentro de la JVM, que es un proceso del sistema operativo y, por tanto, comparten todos los recursos, incluyendo la memoria, las variables y objetos definidos. A este tipo de procesos donde se comparten los recursos se les llama "**procesos ligeros**" (lightweight process).

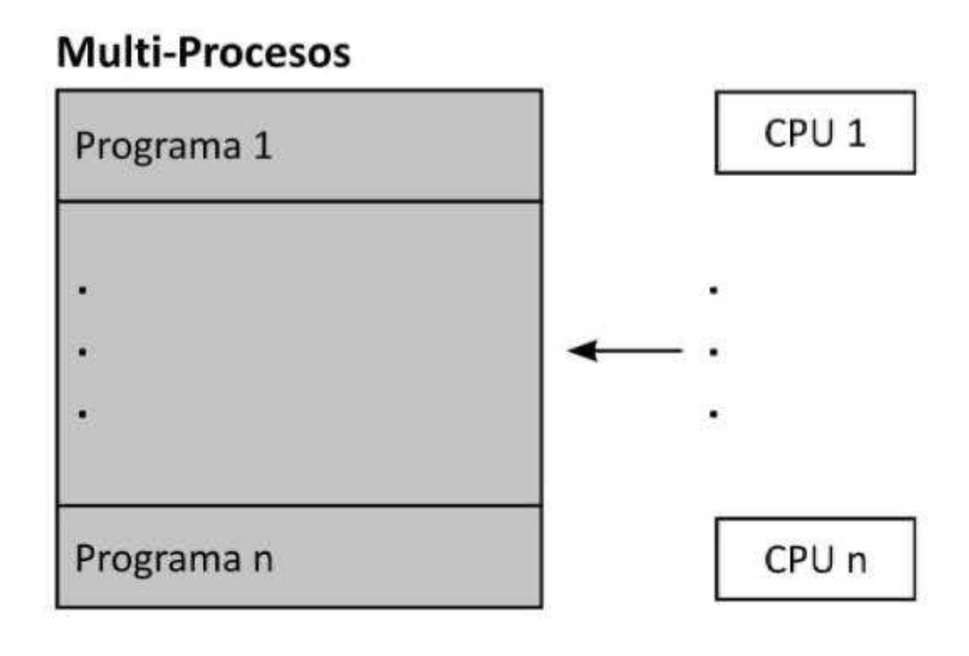

Fig. 15.1 Sistema multiprocesos

Un sistema operativo multiproceso, también denominado multitarea es fundamentalmente un programa que se está realizando. Por lo tanto, se pueden ejecutar dos o más programas a la vez. Por ejemplo, si está funcionando el compilador de Java, un cálculo en una hoja electrónica y bajando correos electrónicos a la vez.

En un sistema multiproceso, un programa es la unidad mínima de código que se puede ejecutar.
Java da soporte al concepto de **Hilo** desde el mismo lenguaje, con algunas clases e interfaces definidas en el paquete **java.lang** y con métodos específicos para la manipulación de **Hilos** en la clase **Object**.

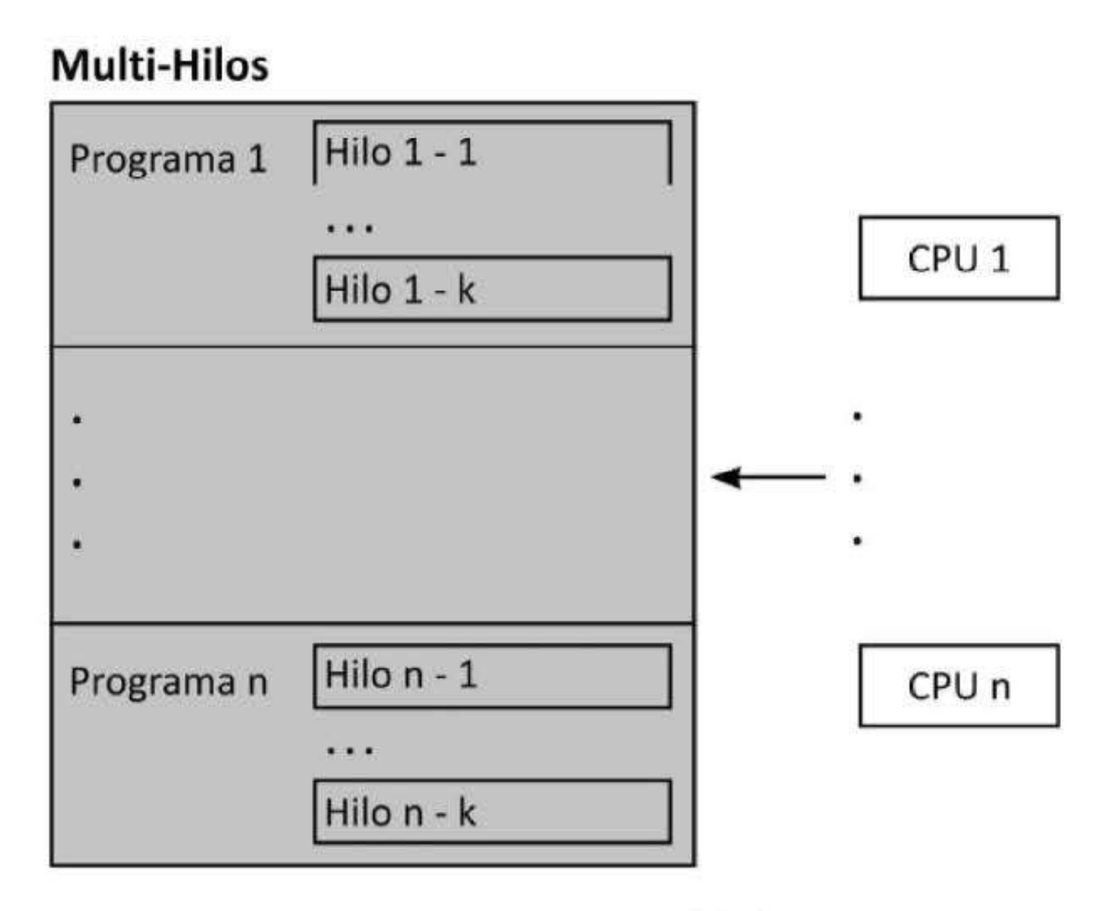

Fig. 15.2 Sistema multihilos

En un sistema multihilos, el **hilo** es la mínima unidad que se puede ejecutar. Esto significa que un único programa puede ejecutar dos o más tareas en forma simultánea. Por ejemplo, una hoja de cálculo puede estar realizando un cálculo al mismo tiempo que está imprimiendo; puesto que estas dos acciones se realizan por **hilos** diferentes. Por lo tanto, un sistema multiprocesos trata de gestiones generales y un sistema multihilos acerca de los detalles.

Desde el punto de vista de las aplicaciones, los **hilos** son útiles porque permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea. Por ejemplo, un **hilo** puede encargarse de la comunicación con el usuario, mientras otros actúan en segundo plano, realizando la transmisión de un archivo, accediendo a recursos del sistema (cargar sonidos, leer archivos, etc.). De hecho todos los programas con interfaz gráfica (AWT o Swing) son multihilos porque los eventos y las rutinas de dibujado de las ventanas corren en un hilo distinto al principal.

Los **hilos** requieren menos sobrecarga que los procesos. Los procesos son tareas más pesadas que necesitan espacio de direccionamiento propio. La comunicación entre procesos es costosa y limitada; el intercambio de contexto, de un proceso a otro, también

es costoso. Mientras que los **hilos** son tareas ligeras, comparten el mismo espacio de direccionamiento y el mismo proceso. La comunicación entre **hilos** requiere un bajo costo. Aunque los programas Java se ejecutan bajo un proceso, no están bajo el control de Java, sin embargo, los **hilos** sí.

La programación basada en **hilos** permite escribir programas muy eficientes que permiten el uso óptimo del CPU. Esto es especialmente importante para los entornos de red interactivos en los que trabaja Java y en los que el CPU se mantiene libre durante mucho tiempo. Por ejemplo, la tasa de transmisión de datos en la red es mucho menor que la velocidad a la que el CPU puede procesarlos. Incluso la velocidad de lectura y escritura en el sistema local de archivos es mucho más baja que la velocidad de proceso del CPU. Y, naturalmente, la velocidad a la que el usuario introduce la información es mucho más baja que la del computador. En un entorno tradicional de un único **hilo**, el programa tiene que esperar a que se realice cada una de estas tareas, antes de procesar la siguiente. Aunque el CPU esté inactivo la mayor parte del tiempo, los **hilos** permiten acceder y aprovechar este tiempo.

- **Ejemplo 1**
  Crearemos una clase que ejecuta un hilo, el proceso es un simple bucle, el número de iteraciones se lo pasamos como parámetro en el constructor, la clase se llama HiloSimple y el código es el siguiente:

HiloSimple .java

```
public class HiloSimple extends Thread {

        private int n;

        // Constructor
        public HiloSimple( String msg, int n){
                super( msg );
                this.n = n;
        }


        // Código del hilo
        public void run(){
                for( int k=1; k <= n; k++ ){
                        System.out.println("Hilo: " + getName() + " Iteracion: " + k);
                }
        }

}
```

El programa que ilustra la aplicación de esta clase es **prog1501**, el código es:

prog1501 .java

```
public class prog1501 {

    public static void main(String[] args) {

            // Creación de los Objetos
            HiloSimple hilo1 = new HiloSimple("Hilo 1", 10);
            HiloSimple hilo2 = new HiloSimple("Hilo 2", 5);
            HiloSimple hilo3 = new HiloSimple("Hilo 3", 8);

            // Lanzar los Hilos
            hilo1.start();
            hilo2.start();
            hilo3.start();

            // Fin
            System.out.println("Fin");

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1501.java                [Enter]

C:\java2n100\cap15>java prog1501              [Enter]
Fin
Hilo: Hilo 1 Iteracion: 1
Hilo: Hilo 2 Iteracion: 1
Hilo: Hilo 3 Iteracion: 1
Hilo: Hilo 1 Iteracion: 2
Hilo: Hilo 2 Iteracion: 2
Hilo: Hilo 3 Iteracion: 2
Hilo: Hilo 1 Iteracion: 3
Hilo: Hilo 2 Iteracion: 3
Hilo: Hilo 3 Iteracion: 3
Hilo: Hilo 1 Iteracion: 4
Hilo: Hilo 2 Iteracion: 4
Hilo: Hilo 3 Iteracion: 4
Hilo: Hilo 1 Iteracion: 5
Hilo: Hilo 2 Iteracion: 5
Hilo: Hilo 3 Iteracion: 5
Hilo: Hilo 1 Iteracion: 6
Hilo: Hilo 3 Iteracion: 6
Hilo: Hilo 1 Iteracion: 7
Hilo: Hilo 3 Iteracion: 7
Hilo: Hilo 1 Iteracion: 8
Hilo: Hilo 3 Iteracion: 8
Hilo: Hilo 1 Iteracion: 9
Hilo: Hilo 1 Iteracion: 10
```

Se han ejecutado tres hilos, puede apreciarse que la primera instrucción que se ejecuta es la última del método **main()**; luego empieza a intercalarse la ejecución de cada hilo.

Existen dos formas de crear un hilo. La primera es mediante herencia, en este caso debemos crear subclases de la clase **Thread**. La segunda forma es implementando la interfaz Runnable.

15.2 IMPLEMENTACIÓN DE HILOS

15.2.1 LA CLASE THREAD

Un **thread** es un hilo en ejecución en un programa. La JVM permite a una aplicación tener múltiples hilos en ejecución concurrentemente.

Cada hilo tiene una prioridad. Los hilos con prioridad más alta se ejecutan antes que los hilos de menor prioridad. Cada hilo puede también ser marcado como *demonio*. Cuando el código que ejecuta un **hilo** crea un nuevo objeto **Thread**, la prioridad inicial del nuevo hilo es igual a la prioridad del hilo que lo crea. Entonces, el hilo es un *demonio*, si y solo si el hilo que lo crea es un *demonio*.

Cuando la JVM se inicia, este es usualmente un hilo simple (que típicamente llama al método **main()** de la clase indicada). La JVM continúa la ejecución del hilo hasta que ocurra alguna de las siguientes situaciones:

- El método **exit()** de la clase **Runtime** es llamada y el manejador de seguridad permite que la operación de la salida ocurra.
- Todos los hilos que no son hilos demonios han finalizado. Entonces el control vuelve al método que realizó la llamada o lanza una excepción que se propaga más allá del método en ejecución.

La clase **Thread** define muchos métodos para la gestión de hilos. Entre los que podemos mencionar:

Método	Descripción
activeCount()	Retorna el número de hilos activos en el hilo actual.
currentThread()	Vuelve una referencia al objeto del hilo actualmente en ejecución.
dumpStack()	Imprime el trazado de pila del hilo actual.
getAllStackTraces()	Retorna el trazado de pila de todos los hilos activos.
getId()	Retorna el identificador del hilo.
getName()	Obtiene el nombre del hilo.
getPriority()	Retorna la prioridad del hilo.

getState()	Retorna el estado del hilo.
interrupt()	Interrumpe este hilo.
interrupted()	Prueba si se ha interrumpido el hilo actual.
isAlive()	Testea si este hilo está vivo.
isDaemon()	Prueba si este hilo es un hilo demonio.
isInterrupted()	Prueba si se ha interrumpido este hilo.
join()	Espera la finalización del hilo.
run()	El método **run()**constituye el cuerpo de un hilo en ejecución. Este es el único método de la interfaz **Runnable**. Es llamado por el método **start()** después que el hilo apropiado del sistema se ha iniciado. Siempre que el método **run()** devuelva el control, el hilo actual se detendrá.
setDaemon()	Marca este hilo de rosca como un hilo del demonio o usuario.
setName()	Cambia el nombre del hilo.
setPriority()	Cambia la prioridad del hilo.
sleep()	Suspende la ejecución del hilo.
start()	Inicia la ejecución del hilo, se ejecuta el método **run()**.
toString()	Retorna una cadena con información del hilo, incluye su nombre y prioridad.
yield()	Hace que el hilo actual se detenga brevemente y permita que otros hilos se ejecuten temporalmente.

Para crear un hilo debemos extender la clase **Thread**, el formato es el siguiente:

```
public class nombre_clase extends Thread {

    // Constructor
    public nombre_clase( . . . ){

            // Implementación

    }

    public void run() {

            // Implementación

    }

    . . .

}
```

Es necesario implementar el método **run()**, de lo contrario el hilo no ejecutará nada. Para ejecutar el hilo debemos invocar el método **start()**.

- **Ejemplo 2**
  Observe la implementación de un hilo de nombre **TablaMultiplicar**, este hilo calcula la tabla de multiplicar. El cálculo lo realiza en una cadena y luego imprime en la consola.

  El código del hilo es el siguiente:

TablaMultiplicar .java

```
public class TablaMultiplicar extends Thread {

    private int n;

    // Constructor
    public TablaMultiplicar( String msg, int n){
            super( msg );
            this.n = n;
    }

    // Código del hilo
    public void run(){
            String tabla = "\nNombre: " + this.getName() + "\n";
            int r;
            for( int k=1; k <= 12; k++ ){
                    r = k * n;
                    tabla =  tabla + "\n" + k + " * " + n + " = " + r;
            }
            System.out.println( tabla );
    }

}
```

El programa que ilustra el uso de este hilo es **prog1502** y el código es el siguiente:

prog1502.java

```
public class prog1502 {

    public static void main(String[] args) {

            // Creación de los Objetos
            TablaMultiplicar t1 = new TablaMultiplicar("Tabla del 5", 5);
            TablaMultiplicar t2 = new TablaMultiplicar("Tabla del 9", 9);

            // Inicio
            System.out.println("Inicio");

            // Lanzar los Hilos
            t1.start();
            t2.start();

            // Fin
            System.out.println("Fin");

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1502.java        [Enter]

C:\java2n100\cap15>java prog1502              [Enter]
Inicio
Fin

Nombre: Tabla del 5

1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
11 * 5 = 55
12 * 5 = 60

Nombre: Tabla del 9

1 * 9 = 9
2 * 9 = 18
3 * 9 = 27
4 * 9 = 36
5 * 9 = 45
6 * 9 = 54
7 * 9 = 63
8 * 9 = 72
9 * 9 = 81
10 * 9 = 90
11 * 9 = 99
12 * 9 = 108
```

15.2.2 El hilo principal

Cuando un programa Java inicia su ejecución, uno de los hilos se ejecuta de forma inmediata. A este hilo suele llamarse hilo principal del programa porque es el único que se ejecuta al comenzar el programa. Es importante por dos motivos:

- Es el hilo desde el que se crea el resto de los hilos del programa.
- A menudo debe ser el último que finaliza su ejecución. Cuando el hilo principal finaliza, el programa termina.

Aunque el hilo principal se crea automáticamente cuando el programa se inicia, se puede controlar a través del objeto **Thread**. Para ello debe obtenerse una referencia a este objeto llamando al método **currentThread()**, que es un miembro **public static de Thread**. Su prototipo es:

```
public static Thread currentThread()
```

Este método retorna una referencia al hilo desde donde se le invocó. Una vez que se tiene una referencia a un hilo principal, puede controlarse del mismo modo que a otro hilo.

- **Ejemplo 3**
  Aplicación de los métodos **currentThread()**, **setName()**, y **sleep()**.

  El programa es **prog1503** y el código es el siguiente:

prog1503.java

```
public class prog1503 {

    public static void main(String[] args) {

            // Obtener referencia al hilo actual
            Thread t = Thread.currentThread();
            System.out.println("Hilo Actual: " + t );

            // Cambiando de nombre al hilo
            t.setName("Super Hilo");
            System.out.println("Hilo Actual: " + t );

            try{
                    for(int n = 5; n > 0; n--){
                            System.out.println(n + "\n");
                            t.sleep(1000);
                    }
            }catch(InterruptedException e){
                    System.out.println("Super Hilo - Detenido");
            }

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1503.java        [Enter]

C:\java2n100\cap15>java prog1503              [Enter]
Hilo Actual: Thread[main,5,main]
Hilo Actual: Thread[Super Hilo,5,main]
5

4

3

2

1
```

Observe que cuando utilizamos **t** en el método **println()**para imprimir el hilo, la información que se imprime es el nombre del hilo, su prioridad y el nombre de su grupo, en este mismo orden.

El método **sleep()**hace que se suspenda el hilo; en este caso por **1000** milisegundos. Por ello es necesario incluirlo en un **try – catch**, porque este método podría lanzar una excepción de tipo **InterruptedException** que es necesario controlar.

15.2.3 La interfaz runnable

Otra forma de ejecutar un hilo es crear una clase que implemente la interfaz **Runnable**. En cualquier clase que aplique esta interfaz se puede construir un hilo. Solo debe implementarse un método de nombre **run()**. El esquema de la implementación del método es el siguiente:

```
public void run() {

    // implementacion
}
```

Dentro de **run()** se define el código que debe ejecutar el nuevo hilo. Es importante entender que dentro de **run()** puede invocarse a otros métodos, usar otras clases y declarar variables de la misma manera que se realiza en el hilo principal. La única diferencia es que **run()** establece el punto de inicio para otro hilo de ejecución concurrente dentro del programa. Este hilo termina cuando **run()** retorna el control.

Para ejecutar el hilo debemos crear un objeto de tipo **Thread**. Dentro de esta clase se definen muchos constructores, pero el que debemos utilizar es:

Donde:

target	Es una instancia de la clase que implementa la interfaz Runnable. Esto define donde iniciará el hilo.
name	Es el nombre del nuevo hilo.

Para iniciar el nuevo hilo debemos ejecutar el método **start()**de la clase **Thread**. El método **start()** invoca el método **run()** implementado en la clase que ejecuta la interfaz **Runnable**.

- **Ejemplo 4**
  Observe la clase **SimpleRunnable**, la cual implementa la interfaz **Runnable**; por lo tanto, se está implementando el método **run()** que es el método con que debe codificarse las instrucciones del hilo.

El código de esta clase es el siguiente:

SimpleRunnable.java

```
class SimpleRunnable implements Runnable{

    private int n;
    private String nombre;

    public SimpleRunnable(String nombre, int n){

            this.nombre = nombre;
            this.n =n;

    }

    public void run(){

            Thread t = Thread.currentThread();
            t.setName(nombre);

            System.out.println("Ejecutando el hilo: " + nombre + "\n");
            int suma = 0;
            for(int i = 1; i <= n; i++){
                    suma += i;
            }

            String rpta = "La suma de los " + n + " primeros numeros es " + suma;
            System.out.println(rpta);

    }

}
```

El hilo de esta clase calcula la suma de los "**n**" primeros números enteros positivos. Para ejecutar el hilo que implementa esta clase, debe crearse un objeto de la clase **Thread** y en su constructor pasar una instancia de dicha clase, y luego ejecutar el método **start()**, tal como se ilustra en el programa **prog1504**.

El código del programa **prog1504** es el siguiente:

prog1504.java

```
public class prog1504 {

    public static void main(String[] args) {

            // Inicio
            System.out.println("Inicio");

            // Creación del objeto
            SimpleRunnable obj = new SimpleRunnable("Suma de Numeros", 25);

            // Creación del Hilo
            Thread hilo = new Thread(obj);
```

```
// Lanzar el hilo
            hilo.start();

            // Espera a que finalice el hilo
            try{
                    hilo.sleep(1000);
                    hilo.join();
            }
            catch(InterruptedException e){
                    System.out.println("Hilo " + hilo.getName() + " interrum-
pido.");
            }

            // Fin
            System.out.println("Fin");

    }

}
```

En este programa estamos creando un objeto **obj**, que es una instancia de la clase **SimpleRunnable**, también está creandose el objeto **hilo**, el cual es una instancia del objeto **Thread** y está pasando en su constructor el objeto **obj**.

Luego iniciamos la aplicación del hilo ejecutando el método **start()**; el método **sleep()** produce un retardo en el hilo, y el método **join()** espera a que el hilo termine para realizar las siguientes sentencias:

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1504.java                    [Enter]

C:\java2n100\cap15>java prog1504                          [Enter]
Inicio
Ejecutando el hilo: Suma de Números

La suma de los 25 primeros números es 325
Fin
```

Cuando ejecute este programa, notará que existe un retardo; sin embargo, la última instrucción del programa no se ejecuta hasta que finalice el hilo.

- **Ejemplo 5**
  Creamos la clase **TablaSumar** que implementa la interfaz **Runnable**.

  En el método **run()** estamos implementado un hilo que calcula la tabla de sumar de un determinado númer y lo muestra en una sola cadena.

  La diferencia con la clase **SimpleRunnable** (ver Ejemplo 4) es que en esta, el hilo se lanza desde el mismo constructor de la clase.

El método **getFinalizado()**retorna **true** si el hilo finaliza su ejecución.

Cuando el hilo finaliza, la variable **finalizado** toma valor **true**; por lo tanto a partir de ese instante el método **getFinalizado()** retornará **true**.

El código de la clase es el siguiente:

TablaSumar.java

```
public class TablaSumar implements Runnable {

    private int n;
    private boolean finalizado = false;

    // Constructor
    public TablaSumar( String msg, int n){
            this.n = n;
            Thread t = new Thread( this, msg );
            t.start();
    }

    // Retorna el estado del hilo
    public boolean getFinalizado(){  return finalizado;        }

    // Código del hilo
    public void run(){

            int r;
            Thread t = Thread.currentThread();
            String tabla = "\nNombre: " + t.getName() + "\n";

            try{
                    for( int k=1; k <= 12; k++ ){
                            r = k + n;
                            tabla =  tabla + "\n" + k + " + " + n + " = " +
r;
                            t.sleep(1);
                    }
                    System.out.println( tabla );
            }
            catch(InterruptedException e){
                    System.out.println("Error en el hilo: " + t.getName());
            }
            finalizado = t.isAlive();

    }

}
```

El programa que ilustra la aplicación de esta clase es **prog1505**, el código es el siguiente:

prog1505.java

```
public class prog1505 {

    public static void main(String[] args) {

            // Inicio
            System.out.println("Inicio");

            // Creación de los Objetos
            TablaSumar t1 = new TablaSumar("Tabla del 5", 5);

            while(!t1.getFinalizado()){
                    System.out.println("Hilo: Tabla del 5 en ejecucion.");
            }

            // Fin
            System.out.println("Fin");

    }

}
```

Note que no es necesario crear el objeto de tipo **Thread**, porque desde el mismo constructor de la clase se lanza el hilo.

Estamos verificando en este programa si el hilo hijo ha finalizado, esto se realiza en el bucle **while**; de tal manera, que la instrucción de finalización solo se ejecuta después que el bucle ha finalizado, es decir, cuando el hilo finalice. Esto puede ser importante, porque podemos necesitar que el hilo principal finalice después que finalizan los hilos hijos.

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1505.java        [Enter]

C:\java2n100\cap15>java prog1505              [Enter]
Inicio
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
```

```
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.
Hilo: Tabla del 5 en ejecucion.

Nombre: Tabla del 5

1 + 5 = 6
2 + 5 = 7
3 + 5 = 8
4 + 5 = 9
5 + 5 = 10
6 + 5 = 11
7 + 5 = 12
8 + 5 = 13
9 + 5 = 14
10 + 5 = 15
11 + 5 = 16
12 + 5 = 17
Hilo: Tabla del 5 en ejecucion.
Fin
```

15.3 SINCRONIZACIÓN

Cuando dos o más hilos requieren acceder a un recurso compartido, es necesario asegurar de alguna forma que accedan a dicho recurso de uno en uno. El proceso por el que se consigue esto se denomina sincronización.

La clave de la sincronización es el concepto de monitor, también llamado semáforo. Un monitor es un objeto que se utiliza como cerrojo exclusivo. Solo uno de los hilos puede tener un monitor en un momento determinado. Cuando un hilo adquiere un cerrojo, se dice que ha entrado en el monitor. Todos los demás hilos que intenten acceder al monitor bloqueado serán suspendidos hasta que el primero salga del monitor; se dice que los otros hilos están esperando al monitor. Un hilo que posee un monitor puede volver a entrar en el mismo monitor si así lo requiere.

- **Ejemplo 6**
  Ilustraremos que podría suceder cuando no se sincroniza el acceso a un método por varios hilos.

  En primer lugar, tenemos la clase **Mensaje** que solo tiene el método **showMsg()**. Este método recibe una cadena y los imprime agregándole caracteres al inicio y al final de la impresión; también captura el hilo donde se está ejecutando y aplica un retardo con el método **sleep()**.

Por ejemplo, si la cadena es **El Perú está Primero**, debería imprimirse:

```
.::El Perú está Primero::.
```

El código de esta clase es el siguiente:

Mensaje.java

```
class Mensaje{

    public void showMsg(String msg){

            Thread t;
            t = Thread.currentThread();
            try{
                    System.out.print(".::");
                    System.out.print(msg);
                    t.sleep(1000);
                    System.out.println("::.");
            }
            catch(InterruptedException e){
                    System.out.println("Hilo " + t.getName() + " Interrumpido");
            }

    }

}
```

También contamos con la clase **MuestraMensaje**, esta clase lanza un hilo para que muestre un mensaje. Su constructor recibe como parámetros:

- Una instancia de la clase **Mensaje**
- El mensaje a mostrar
- El título del hilo

El hilo se lanza desde su constructor, es decir, que cuando se crea el objeto automáticamente se lanza el hilo. El acceso al hilo que es lanzado está disponible a través de la variable pública **hilo**.

El código de esta clase es el siguiente:

MuestraMensaje.java

```
class MuestraMensaje implements Runnable{

    private String msg;
    private Mensaje obj;
    public Thread hilo;

    public MuestraMensaje(Mensaje obj, String msg, String titulo){

            this.obj = obj;
            this.msg = msg;
            hilo = new Thread(this);
            hilo.setName(titulo);
            hilo.start();

    }

    public void run(){

            obj.showMsg(msg);

    }
}
```

El programa que ilustra el uso de estas clases es **prog1506**, el código es el siguiente:

prog1506.java

```
public class prog1506 {

    public static void main(String[] args) {

            // Inicio
            System.out.println("Inicio\n");

            // Creación de los Onjetos
            Mensaje objMsg = new Mensaje();
            MuestraMensaje hilo1 = new MuestraMensaje(objMsg,"Java es el
Mejor","Hilo 1");
            MuestraMensaje hilo2 = new MuestraMensaje(objMsg,"JSP las mas
Seguras","Hilo 2");
            MuestraMensaje hilo3 = new MuestraMensaje(objMsg,"Java es
Garantia","Hilo 3");

            // Espera a que los hilos terminen
            try{

                    hilo1.hilo.join();
                    hilo2.hilo.join();
                    hilo3.hilo.join();

            }
            catch(InterruptedException e){
                    System.out.println("Programa Interrumpido");
            }

            // Fin
            System.out.println("\nFin");

    }

}
```

Note que se crean tres hilos y se espera a que los hilos terminen para que finalice el hilo principal. Se espera el siguiente resultado:

```
.:: Java es el Mejor::.
.:: JSP las mas Seguras::.
.:: Java es Garantia::.
```

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1506.java        [Enter]

C:\java2n100\cap15>java prog1506              [Enter]
Inicio

.::.::.::Java es el MejorJSP las mas SegurasJava es Garantia::.
::.
::.

Fin
```

Como puede observarse, al llamar al método **sleep()**, el método **showMsg()** permite que la ejecución se conmute a otro hilo. El resultado es una salida en la que se mezclan tres mensajes. En este programa no hay nada que impida que los tres hilos llamen al mismo método, del mismo objeto y al mismo tiempo. Esto es lo que se conoce como condición de carrera, porque los tres hilos compiten para completar el método. Este método utiliza **sleep()** para que los efectos sean notorios y obvios. En la mayoría de las situaciones, la condición de carrera es más sutil y menos predecible, ya que no hay seguridad de cuando se produce el intercambio de contexto. Además, esto puede hacer que un programa se ejecute correctamente algunas veces e incorrectamente otras.

Para corregir este problema, se debe hacer un acceso en serie a **showMsg()**. Es decir, debe restringirse el acceso a un solo hilo a la vez. Esto se puede conseguir de dos maneras:

- Construyendo métodos sincronizados.
- Utilizando la instrucción de sincronización.

15.3.1 Palabra clave synchronized

La palabra clave **synchronized** asegura que solo un hilo ejecute un método y, a la vez, este método es bloqueado.

La forma de utilizar esta palabra clave se establece en la declaración de los métodos, su sintaxis es la siguiente:

```
[acceso] synchronized tipo nombre_método ( parámetros ) {

    // Implementación

}
```

- **Ejemplo 7**
  En este ejemplo modificaremos el Ejemplo 6 para ilustrar el uso de la palabra clave **synchronized**, los nombres de las nuevas clases son:

Ejemplo 6	Ejemplo 7
Mensaje	Mensaje2
MuestraMensaje	MuestraMensaje2
prog1506	prog1507

El cambio principal se produce en la declaración del método **showMsg()** en la clase Mensaje, específicamente en la clase **Mensaje2**.

Para reconocer rápidamente los cambios realizados estos están con letra de diferente tipo (**Trebuchet MS**), al igual que las nuevas líneas de código que se puedan haber agregado.

El código de la clase **Mensaje2** es el siguiente:

Mensaje2.java

```
class Mensaje2{

    public synchronized void showMsg(String msg){

            Thread t;
            t = Thread.currentThread();
            try{
                    System.out.print(".::");
                    System.out.print(msg);
                    t.sleep(1000);
                    System.out.println("::.");
            }
            catch(InterruptedException e){
                    System.out.println("Hilo " + t.getName() + " Interrumpido");
            }

    }

}
```

El código de la clase **MuestraMensaje2** es el siguiente:

MuestraMensaje.java

```
class MuestraMensaje2 implements Runnable{

    private String msg;
    private Mensaje2 obj;
    public Thread hilo;

    public MuestraMensaje2(Mensaje2 obj, String msg, String titulo){

            this.obj = obj;
            this.msg = msg;
            hilo = new Thread(this);
            hilo.setName(titulo);
            hilo.start();

    }

    public void run(){

            obj.showMsg(msg);

    }
}
```

El código del programa **prog1507** es el siguiente:

prog1507.java

```
public class prog1507 {

    public static void main(String[] args) {

            // Inicio
            System.out.println("Inicio\n");

            // Creación de los Onjetos
            Mensaje2 objMsg = new Mensaje2();
            MuestraMensaje2 hilo1 = new MuestraMensaje2(objMsg,"Java es el
Mejor","Hilo 1");
            MuestraMensaje2 hilo2 = new MuestraMensaje2(objMsg,"JSP las mas
Seguras","Hilo 2");
            MuestraMensaje2 hilo3 = new MuestraMensaje2(objMsg,"Java es
Garantia","Hilo 3");

            // Espera a que los hilos terminen
            try{

                    hilo1.hilo.join();
                    hilo2.hilo.join();
                    hilo3.hilo.join();

            }
```

```
catch(InterruptedException e){
                    System.out.println("Programa Interrumpido");
            }

            // Fin
            System.out.println("\nFin");

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1507.java        [Enter]

C:\java2n100\cap15>java prog1507              [Enter]
Inicio

.::Java es el Mejor::.
.::JSP las mas Seguras::.
.::Java es Garantia::.

Fin
```

Como puede comprobarse, el resultado que se obtiene es el esperado.

15.3.2 Instrucción synchronized

Aunque la creación de métodos sincronizados dentro de clases creadas por el propio programador es un método sencillo y eficaz para lograr la sincronización, lamentablemente no funciona con todas las clases. Para comprender el por qué, analicemos los siguientes casos:

1. Se quiere sincronizar el acceso a objetos de una clase que no fue diseñada para acceso multihilo, es decir, esta clase no utiliza métodos sincronizados.
2. Se quiere sincronizar el acceso a objetos de clases de terceros, de las cuales no tenemos el código fuente; por lo tanto, no podemos módificar los métodos.

Afortunadamente tenemos la instrucción **synchronized** para resolver estos problemas. La instrucción **synchronized** se usa para indicar que ciertas partes del código (un método) están sincronizadas, es decir, que solamente un hilo puede acceder a dicho método a la vez.

Cada método sincronizado posee una especie de llave que puede cerrar o abrir la puerta de acceso. Cuando el hilo intenta acceder al método sincronizado; primero verificará si la llave está cerrada, en cuyo caso no podrá accederlo. Si el método no tiene cerrada la llave, entonces el hilo puede acceder a dicho método sincronizado.

La forma de utilizar esta instrucción es la siguiente:

```
synchronized( objeto ) {

    // Instrucciones

}
```

Así, **objeto** es una referencia al objeto que se quiere sincronizar. Un bloque sincronizado asegura que solo se producirá una llamada al método miembro de objeto después que el hilo actual haya iniciado.

- **Ejemplo 8**
  Modificaremos el Ejemplo 6 para ilustrar el uso de la instrucción **synchronized**, los nombres de las nuevas clases son:

Ejemplo 6	Ejemplo 8
Mensaje	Mensaje
MuestraMensaje	MuestraMensaje3
prog1506	prog1508

En la **clase Mensaje** no será necesario realizar cambios, el cambio principal se hará en la declaración del método **run()** en la clase **MuestraMensaje**, específicamente en la clase **MuestraMensaje3**.

Para reconocer rápidamente los cambios realizados, estos están con tipo de letra diferente (**Trebuchet MS**), al igual que las nuevas líneas de código que se puedan haber agregado.

Para recordar la definición de la clase **Mensaje**, su código es el siguiente:

Mensaje.java

```
class Mensaje{

    public void showMsg(String msg){

            Thread t;
            t = Thread.currentThread();
            try{
                    System.out.print(".::");
                    System.out.print(msg);
                    t.sleep(1000);
                    System.out.println("::.");
            }
            catch(InterruptedException e){
                    System.out.println("Hilo " + t.getName() + " Interrumpido");
            }

    }

}
```

El código de la clase **MuestraMensaje3** es el siguiente:

MuestraMensaje.java

```
class MuestraMensaje3 implements Runnable{

    private String msg;
    private Mensaje obj;
    public Thread hilo;

    public MuestraMensaje3(Mensaje obj, String msg, String titulo){

            this.obj = obj;
            this.msg = msg;
            hilo = new Thread(this);
            hilo.setName(titulo);
            hilo.start();

    }


    public void run(){
            synchronized( obj ){
                    obj.showMsg(msg);
            }

    }
}
```

El código del programa **prog1508** es el siguiente:

prog1508.java

```
public class prog1508 {

    public static void main(String[] args) {

            // Inicio
            System.out.println("Inicio\n");

            // Creación de los Onjetos
            Mensaje objMsg = new Mensaje();
            MuestraMensaje3 hilo1 = new MuestraMensaje3(objMsg,"Java es el
Mejor","Hilo 1");
            MuestraMensaje3 hilo2 = new MuestraMensaje3(objMsg,"JSP las mas
Seguras","Hilo 2");
            MuestraMensaje3 hilo3 = new MuestraMensaje3(objMsg,"Java es
Garantia","Hilo 3");

            // Espera a que los hilos terminen
            try{

                    hilo1.hilo.join();
                    hilo2.hilo.join();
                    hilo3.hilo.join();
```

```
        }
                catch(InterruptedException e){
                        System.out.println("Programa Interrumpido");
                }

                // Fin
                System.out.println("\nFin");

        }

}
```

Compilación y ejecución:

```
C:\java2n100\cap15>javac prog1508.java        [Enter]

C:\java2n100\cap15>java prog1508              [Enter]
Inicio

.::Java es el Mejor::.
.::Java es Garantia::.
.::JSP las mas Seguras::.

Fin
```

En este caso también obtenemos el resultado esperado.

I/O (Entrada/Salida)

CAPÍTULO 16

Los flujos permiten transferir datos de un origen al programa o viceversa. Las clases para trabajar con flujos están en **java.io**; por lo tanto, será necesario importar las clases que se encuentran en este paquete a los programas que necesitan trabajar con flujos.

16.1 FLUJOS (STREAMS)

Cualquier programa realizado en Java que necesite llevar a cabo una operación de I/O, lo hará a través de un flujo. Un flujo es una abstracción de todo aquello que produzca o consuma información. Podemos ver a este flujo como una entidad lógica. La vinculación de este flujo al dispositivo físico la hace el sistema de entrada y salida de Java. Se ve la eficacia de esta implementación, pues las clases y métodos de I/O que necesitamos emplear son las mismas, independientemente del dispositivo con el que estemos actuando; luego, el núcleo de Java, sabrá si tiene que tratar con el teclado, el monitor, un sistema de ficheros o un *socket* de red, liberando a nuestro código de tener que saber con quién está interactuando.

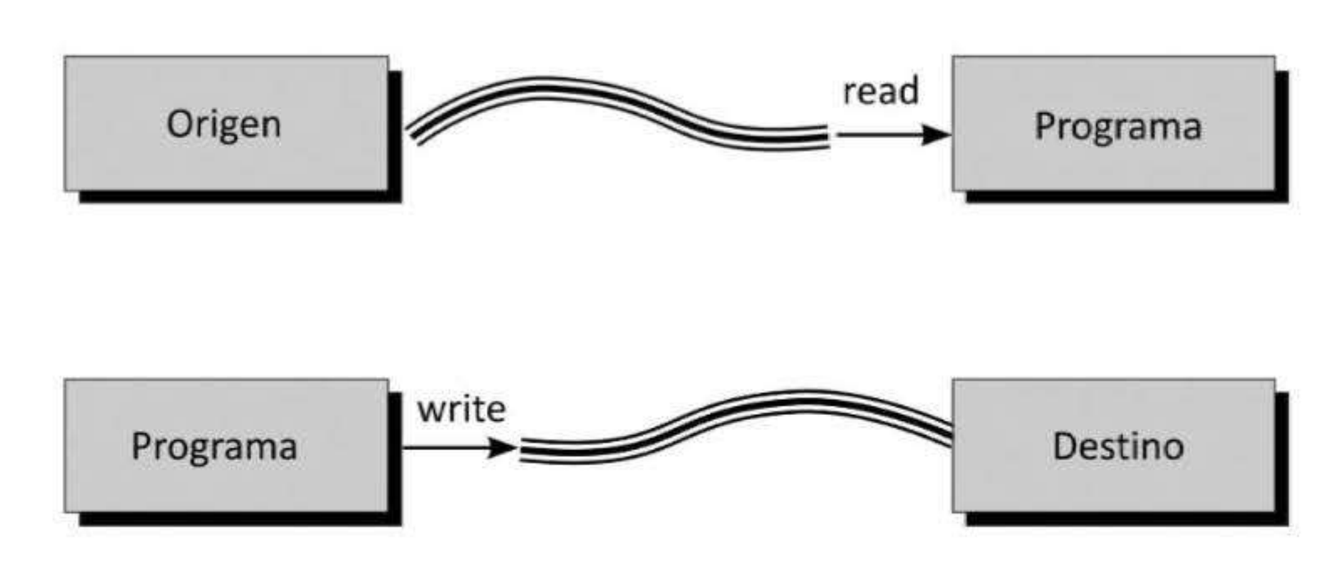

Fig. 16.1 Representación de flujos

16.1.1 Flujos de bytes

Nos proporciona un medio adecuado para el manejo de entradas y salidas de bytes y su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene gobernado por dos clases abstractas que son **InputStream** y **OutputStream**. Cada una de estas tiene varias subclases concretas que controlan las diferencias entre los distintos dispositivos de I/O que se puedan utilizar. Asimismo, son las que definen los métodos que las subclases tendrán que implementar; destacan los métodos **read()** y **write()** que leen y escriben bytes de datos, respectivamente.

16.1.2 Flujos de caracteres

Nos proporcionan un medio conveniente para el manejo de entradas y salidas de caracteres. Dichos flujos usan codificación Unicode y, por tanto, se pueden internacionalizar. Este es un modo que Java nos proporciona para manejar caracteres, pero al nivel más bajo todas las operaciones de I/O son orientadas a byte. Al igual que la anterior, el flujo de caracteres también viene gobernado por dos clases abstractas: **Reader** y **Writer.** Dichas clases manejan flujos de caracteres **Unicode**. Y también derivan subclases concretas que implementan los métodos definidos en ellas, siendo los más destacados los métodos **read()** y **write()** que, en este caso, leen y escriben caracteres de datos, respectivamente.

16.1.3 Flujos predefinidos

Todos los programas Java importan el paquete **java.lang** automáticamente. Este paquete define una clase llamada **System**, que encapsula varios aspectos del entorno de ejecución. Por ejemplo, empleado alguno de sus métodos, se puede obtener la hora actual o el valor de varias propiedades asociadas al sistema. La clase **System** también tiene tres variables de flujo predefinidas: **in**, **out** y **err**. Estos campos se declaran como **public** y **static** dentro de **System**, lo que significa que pueden ser utilizados en cualquier otra parte del programa, sin necesidad de hacer referencia a un objeto específico de **System**.

System.out hace referencia al flujo de salida estándar que por defecto es la consola. **System.in** hace referencia a la entrada estándar, que por defecto es el teclado. **System.err** hace referencia al flujo de error estándar, que por defecto también es la consola. Todos estos flujos pueden redirigirse a cualquier dispositivo compatible con I/O.

System.in es un objeto de tipo **InputStream**, **System.out** y **System.err**; además son objetos de tipo **PrintStream**. Todos ellos son flujos de bytes, incluso cuando se utilicen para leer y escribir caracteres desde y a la consola. Estos flujos se pueden envolver en flujos basados en caracteres.

- **Ejemplo 1**
  Mostramos como se redirecciona la salida estándar a un archivo de texto; en este caso **outfile.txt**.

  El código del programa es el siguiente:

prog1601.java

```
import java.io.*;

class prog1601{

    public static void main(String[] args) {

            File outFile = new File("outfile.txt");

            FileOutputStream fw;
            try {
                    fw = new FileOutputStream(outFile);
            } catch (IOException e) {
                    fw = null;
            }


              PrintStream output = new PrintStream(fw);
            System.setOut(output);

            System.out.println("Salida Redireccionada");
            System.out.println("al archivo outfile.txt");
            System.out.println("Este es un ejemplo.");


    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap16>javac prog1601.java        [Enter]

C:\java2n100\cap16>java prog1601              [Enter]

C:\java2n100\cap16>type outfile.txt           [Enter]
Salida Redireccionada
al archivo outfile.txt
Este es un ejemplo.
```

Al redireccionarse la salida estándar a un archivo de texto, todos los **println()** serán enviados al archivo de texto, en este ejemplo **outfile.txt**.

16.2 ENTRADA / SALIDA POR CONSOLA

16.2.1 MÉTODOS: PRINT() Y PRINTLN()

Estos métodos se utilizan para imprimir por defecto en la consola, la diferencia es que **print()**no hace un cambio de línea, mientras que **println()** sí lo hace; tal como se ha utilizado a lo largo de todo el libro.

16.2.2 MÉTODO READ()

Véase el siguiente byte de datos del flujo de entrada. El valor del byte se vuelve como *int* en el rango de **0** a **255**. Si no hay ningún byte disponible, retorna el valor -1.

- **Formato:**

```
int tecla = System.in.read();
```

o

```
int tecla;
tecla = System.out.read();
```

- **Ejemplo 2**

Ilustramos la forma de utilizar el método **read()**.

El programa es **prog1602** y su código es el siguiente:

prog1602.java

```
import java.io.*;

class prog1602{

    public static void main(String[] args) {

            try{

                    System.out.print("Ingrese un caracter => ");
                    int letra1 = System.in.read();
                    System.in.skip(4);

                    System.out.print("Ingrese otro caracter => ");
                    int letra2 = System.in.read();

                    System.out.println((char)letra1 + " codigo " + letra1);
                    System.out.println((char)letra2 + " codigo " + letra2);

            }
            catch( IOException e){
                    System.out.println("Error en la lectura.");
            }

    }
}
```

Compilación y ejecución:

```
C:\java2n100\cap16>javac prog1602.java        [Enter]

C:\java2n100\cap16>java prog1602              [Enter]
Ingrese un caracter => A                      [Enter]
Ingrese otro caracter => X                    [Enter]
A codigo 65
X codigo 88
```

Este programa solicita que ingrese dos caracteres (uno por uno) en la ejecución, se observa que han ingresado las letras **A** y **X**; luego muestra el mismo carácter y el valor de su código ASCII.

- **Ejemplo 3**

  El siguiente programa ilustra como leer una cadena de caracteres utilizando el método **read()**, en este caso, se ha creado el método **readLine()**, el cual lee una cadena y lo retorna como tal. El programa es el siguiente:

prog1603.java

```
import java.io.*;

class prog1603{

      public static void main(String[] arg) {

              String nombre;
              System.out.print("Ingrese su nombre: ");
              nombre = leerLinea();
              System.out.println("Hola: " + nombre);

      }

      static String leerLinea () {

              String s = new String();
              char car;

              try {

                      car = (char) System.in.read();
                      while ( car != '\n' ){
                              s += car;
                              car = (char) System.in.read();
                      }

              }
              catch (java.io.IOException e) {
                      s = null;
              }

              return s;

      }

}
```

Compilación y ejecución:

```
C:\java2n100\cap16>javac prog1603.java        [Enter]

C:\java2n100\cap16>java prog1603              [Enter]
Ingrese su nombre: Gustavo                    [Enter]
Hola: Gustavo
```

Como se muestra en su ejecución, solicita un nombre y luego de escribirlo debe presionar la tecla**[Enter]**. Finalmente muestra un saludo.

La forma de leer una cadena es utilizando un flujo de bytes, se recomienda utilizar un flujo de caracteres para leer una cadena de caracteres.

16.3 TRABAJANDO CON FLUJOS DE CARACTERES

16.3.1 Clase reader

Se trata de una clase abstracta para leer flujos de caracteres. Los únicos métodos que las subclases necesitan implementar son **read(char[], int, int)** and **close()**. La mayoría de subclases, sin embargo, sobreescriben algunos de los métodos definidos aquí para proporcionar una mejor eficacia, funcionalidad adicional o ambos.

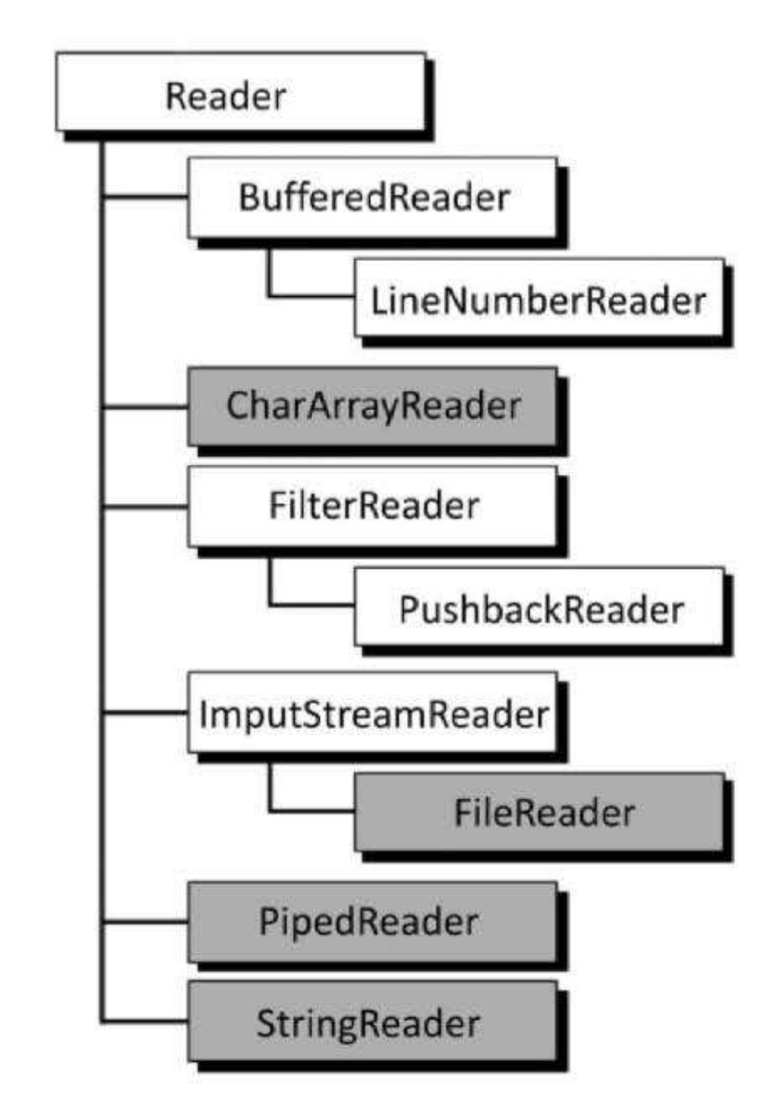

Fig. 16.2 La clase Reader y subclases

De esta clase derivan una serie de subclases (ver Fig. 16.2) y todas se encuentran en el paquete **java.io**.

Las clases con fondo gris, indican desde donde se leen los datos, es decir, el dispositivo que se conecta con el flujo. Las demás (fondo blanco) añaden características particulares a la forma de leer los datos. La intención es que se combinen para obtener el comportamiento deseado.

16.3.2 Entrada por consola

En los ejemplos anteriores, se ha utilizado un flujo de bytes para realizar la lectura desde la consola. Pero lo más recomendable, es utilizar un flujo de caracteres que hace que el programa sea más fácil de internacionalizar y mantener.

La entrada por consola se realiza a través de **System.in**. Para obtener un flujo basado en caracteres conectado a la consola, se envuelve **System.in** en un objeto **BufferedReader** que facilita un *buffer* para el flujo de entrada. El constructor más utilizado es el siguiente:

```
BufferedReader(Reader in)
```

Donde **in** es el flujo asociado a la instancia de **BufferedReader** que se está creando. **Reader** es una clase abstracta y una de sus clases concretas es **InputStreamReader**, la cual convierte bytes en caracteres. Para obtener un objeto **InputStreamReader** ligado a **System.in** hay que utilizar el siguiente constructor:

```
InputStreamReader(InputStream in)
```

Como **System.in** hace referencia a un objeto de tipo **InputStream**, se puede utilizar como argumento en el constructor. Las siguientes instrucciones crean un **BufferedReader** conectado al teclado:

La Fig. 16.3 ilustra gráficamente cómo funciona la lectura desde el teclado.

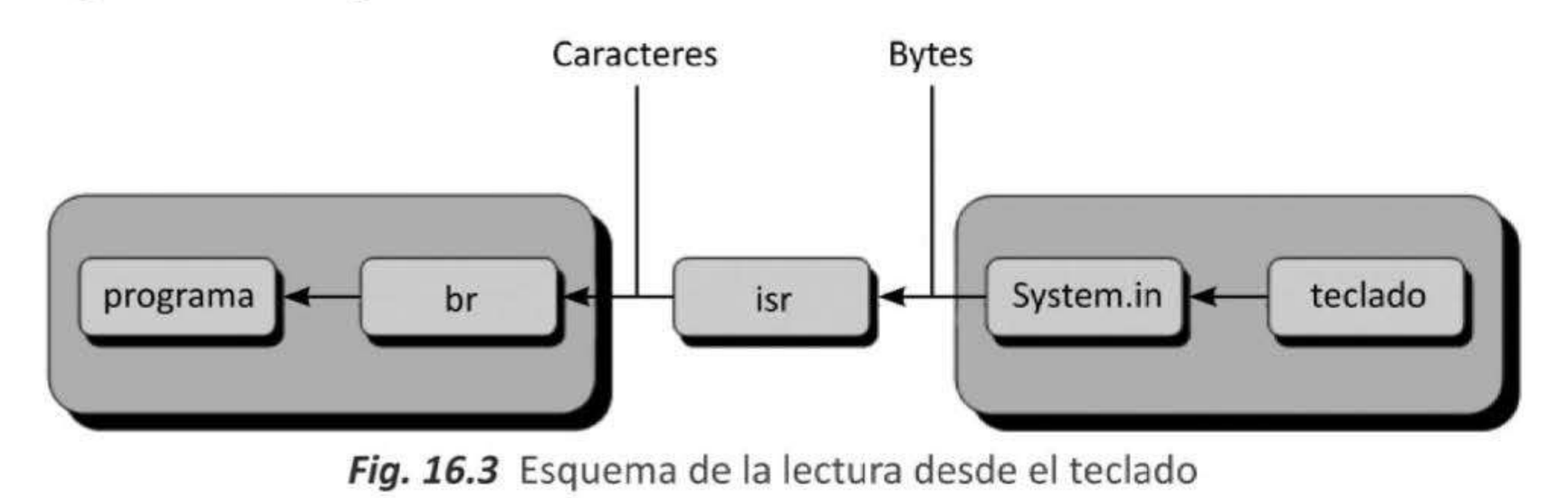

Fig. 16.3 Esquema de la lectura desde el teclado

Las dos instrucciones anteriores se pueden simplificar en una sola, como se muestra a continuación:

```
BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );
```

Después de crear la instancia de **BufferedReader**, en este caso **br**, podemos utilizar el método **readLine()** para leer un cadena de caracteres:

```
String cadena = br.readLine();
```

o

```
String cadena;

cadena = br.readLine();
```

El método **readLine()** puede generar una excepción de **IO** de tipo **IOException**; por lo tanto debe estar en un bloque **try – catch**.

- **Ejemplo 4**
  Creamos el método lectura, para leer una cadena desde el teclado, pero esta vez utilizando un flujo de caracteres.

  El código del programa es el siguiente:

prog1604.java

```
import java.io.*;

class prog1604{

    public static void main(String[] arg) {

            String nombre;
            String apellido;

            System.out.println("Ingrese sus Datos");
            System.out.print("Nombre: ");
            nombre = lectura();
            System.out.print("Apellido: ");
            apellido = lectura();

            System.out.println("Hola " + nombre + " " + apellido);

    }

    static String lectura() {

            InputStreamReader isr = new InputStreamReader( System.in );
            BufferedReader br = new BufferedReader(isr);

            String nombre = new String();
            try {
                    nombre = br.readLine();
            }catch (java.io.IOException e) {
                    System.out.println(e.getMessage());
            }

            return nombre;
    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap16>javac prog1604.java        [Enter]

C:\java2n100\cap16>java prog1604              [Enter]
Ingrese sus Datos
Nombre: Gustavo                               [Enter]
Apellido: Coronel                             [Enter]
Hola Gustavo Coronel
```

Como puede comprobarse, el uso de un flujo de caracteres es más sencillo y transparente.

16.3.3 Clase Writer

Se trata de una clase abstracta para escribir flujos de caracteres. Los únicos métodos que las subclases necesitan implementar son: **write(char[], int, flush()** y **close().** La mayoría de subclases, sin embargo, sobreescriben algunos de los métodos definidos aquí para proporcionar una mejor eficacia, funcionalidad adicional o ambos.

De esta clase derivan una serie de subclases (ver Fig. 16.4) y todas se encuentran en el paquete **java.io**.

Las clases con fondo gris indican donde se escriben los datos, es decir, el dispositivo que se conecta con el flujo. Las demás (fondo blanco) añaden características particulares a la forma de escribir datos. La intención es que se combinen para obtener el comportamiento deseado.

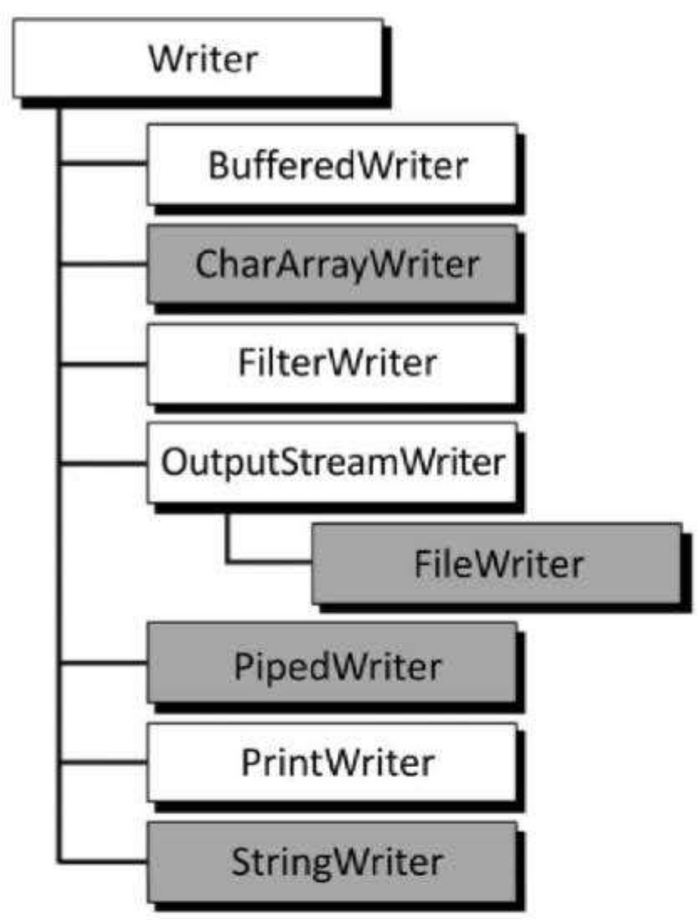

Fig. 16.4 La clase Writer y subclases

16.3.4 Salida por consola

La salida por consola se realiza de forma sencilla con los métodos **print()** y **println()**, descritos anteriormente, y que se utiliza en los ejemplos de este libro. Estos métodos se definen en la clase **PrintStream** que es el tipo de objeto al que hace referencia **System.out**. Incluso cuando **System.out** es un flujo de bytes, su uso se acepta para las salidas sencillas de un programa, como por ejemplo, para verificar el valor de una variable, depurar un programa o ejemplos de aprendizaje.

Otra alternativa es utilizar la clase **PrintWriter**, que está basada en un flujo de caracteres. **PrintWriter** define varios constructores, pero debe utilizarse el siguiente:

```
PrintWriter(OutputStream out, boolean autoFlush)
```

out es un objeto de tipo **OutputStream** y **autoFlush** que controla si Java limpia automáticamente el flujo de salida.

PrintWriter implementa dos métodos sobrecargados: **print()** y **println()**, para todos los tipos, incluyendo **Object**. Por lo tanto pueden utilizarse estos métodos de la misma manera como se han utilizado con **System.out**. Si un argumento no es un tipo simple, los métodos de **PrintWriter** llaman al método **toString()** del objeto y luego se imprime el resultado.

Para escribir en la consola utilizando la clase **PrintWriter**, se debe especificar **System.out** como flujo de salida y limpiar el flujo después de cada nueva línea.

Por ejemplo, el siguiente código crea un objeto **PrintWriter** que se conecta con la salida por consola y luego imprime el mensaje "**Esto está Chevere**":

```
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("Esto está Chevere.");
```

Recuerde que para efectos de aprendizaje o depuración, utilizar **System.out** o **PrintWriter** es indiferente, no hay ninguna ventaja de uno sobre el otro. Usted elije el que más le conviene.

16.3.5 Lectura y escritura de archivos

Java proporciona clases y métodos que permiten leer y escribir sobre archivos. Dos de estas clases son **FileInputStream** y **FileOutputStream**, que crean flujos de bytes ligados a otros archivos. Para abrir un archivo simplemente hay que crear un objeto de una de

estas clases, especificando el nombre del archivo como argumento al constructor. Los prototipos de los constructores a utilizar son:

```
public FileInputStream(String name) throws FileNotFoundException

public FileOutputStream(String name) throws FileNotFoundException

public FileOutputStream(String name, boolean append)
    throws FileNotFoundException
```

name especifica el nombre del archivo; **append** indica si se sobreescribirá el archivo o adicionarán más datos. En caso de errores, se generará la excepción **FileNotFoundException**. Cuando se ha terminado de trabajar con un archivo se debe cerrar el mismo llamando al método **close()**, que está definido en las clases **FileInputStream** y **FileOutputStream**.

El prototipo del método close() es el siguiente:

```
public void close()throws IOException
```

Para leer los datos de un archivo debemos utilizar el método **read()** definido en **FileInputStream**, se trata de un método sobrecargado, pero su prototipo más simple es el siguiente:

```
public int read() throws IOException
```

Para escribir debemos utilizar el método **write()** definido en **FileOutputStream**, se trata de un método sobrecargado, pero su prototipo más simple es el siguiente:

```
public void write(int b) throws IOException
```

- **Ejemplo 5**
  Leemos el contenido de un archivo y lo mostramos en la consola. El nombre del archivo lo pasamos como parámetro al programa.

El código del programa es el siguiente:

prog1605.java

```
import java.io.*;

class prog1605{

    public static void main(String[] arg) {

            int car;
            FileInputStream fin;

            try {

                    fin = new FileInputStream( args[0] );
                    car = fin.read();
                    while(car > -1){
                            System.out.print((char)car);
                            car = fin.read();
                    }
                    fin.close();

            } catch (FileNotFoundException e) {
                    System.out.println("Archivo no encontrado.");
            } catch (ArrayIndexOutOfBoundsException e){
                    System.out.println("No ha especificado el nombre del ar-
chivo.");
            } catch(IOException e) {
                    System.out.println("Error en acceso al archivo");
            }

    }

}
```

Compilación y ejecución:

```
C:\java2n100\cap16>javac test.java          [Enter]

C:\java2n100\cap16>java test mensaje.txt    [Enter]
Despues de estudiar este
libro, puedo decir que
estoy listo para dar
el gran salto en Java.
```

Dentro del bucle **while** se lee el contenido del archivo, byte por byte y se escribe en la consola.

- **Ejemplo 6**
  El siguiente programa copia un archivo origen en un archivo destino. Los nombres de los archivos se pasan como parámetros en la línea de comando de la consola.

El código del programa es el siguiente:

prog1606.java

```
import java.io.*;

class prog1606{

    public static void main(String[] arg) {

            int car;
            FileInputStream fin = null;                // Archivo Origen
            FileOutputStream fout = null;    // Archivo Destino

            try{

                    // Abre el archivo de entrada
                    try {
                            fin = new FileInputStream( arg[0] );
                    } catch (FileNotFoundException e) {
                            System.out.println("Archivo de entrada no encontrado.");
                            return;
                    }

                    // Abre el archivo de salida
                    try {
                            fout = new FileOutputStream( arg[1], true );
                    } catch (FileNotFoundException e) {
                            System.out.println("Archivo de salida no encontrado.");
                            return;
                    }

            } catch (ArrayIndexOutOfBoundsException e){
                    System.out.println("No ha especificado el nombre del ar-
chivo.");
            }

            // Inicio de Copiado
            int archivo = 1;
            System.out.println("Inicio de Copia.");

            try{
                    archivo = 1;
                    car = fin.read();
                    while(car > -1){
                            archivo = 2;
                            fout.write(car);
                            car = fin.read();
                    }
    }

            } catch (IOException e) {
                    if(archivo==1){
                            System.out.println("Error en el archivo de entrada.");
                    } else {
                            System.out.println("Error en el archivo de salida.");
                    }
            }
```

```
            try {
                    fin.close();
                    fout.close();
            } catch(IOException e){
                    System.out.println("Error en uno de los archivos.");
            }

            // Fin de Copiado
            System.out.println("Fin de Copia.");

        }

    }
```

En este programa se esta utilizando **try – catch** anidados, dentro del bucle **while** se lee carácter por carácter del archivo origen, y se escribe en el archivo destino.

Compilación y ejecución:

```
C:\java2n100\cap16>javac prog1606.java                    [Enter]

C:\java2n100\cap16>java prog1606 mensaje.txt copia.txt    [Enter]
Inicio de Copia.
Fin de Copia.
```

El archivo origen en este caso es **mensaje.txt**, y el archivo destino es **copia.txt** (la copia se ha realizado con éxito). Para ver el contenido del archivo **copia.txt** debemos utilizar el comando de consola **type**, como se ilustra a continuación:

```
C:\java2n100\cap16>type copia.txt    [Enter]
Después de estudiar este
libro, puedo decir que
estoy listo para dar
el gran salto en Java.
```

Si ejecutamos nuevamente el programa, el contenido del archivo **mensaje.txt** se añade al archivo **copia.txt**, esto porque ya existe; luego cuando se abre el archivo con el constructor **FileOutputStream()**, el segundo parámetro es **true** y este valor indica que debe agregarse y no sobreescribir cuando el archivo ya existe:

```
C:\java2n100\cap16>java prog1606 mensaje.txt copia.txt        [Enter]
Inicio de Copia.
Fin de Copia.

C:\java2n100\cap16>type copia.txt                             [Enter]
Después de estudiar este
libro, puedo decir que
estoy listo para dar
el gran salto en Java.
Después de estudiar este
libro, puedo decir que
estoy listo para dar
el gran salto en Java.
```

Manejo de cadenas

CAPÍTULO 17

Un tema importante en todo lenguaje es que proporcione herramientas para el manejo de cadenas, en Java una cadena es un objeto de la clase **String**, y a través de esta clase, Java proporciona una gran cantidad de métodos estáticos para un correcto manejo de cadenas.

17.1 FUNDAMENTOS

Las cadenas son una secuencia de caracteres, pero en Java se implementan como objetos de tipo **String**.

La implementación de cadenas como objetos incorporados en Java proporciona un conjunto completo de características que facilitan su manipulación. Por ejemplo, la clase **String** implementa métodos para comparar dos cadenas, buscar subcadenas, concatenar dos cadenas e intercambiar las mayúsculas y las minúsculas dentro de una cadena. Además, los objetos de tipo **String** se pueden construir de varias formas, lo que facilita la obtención de una cadena cuando se necesita.

Sin embargo, los objetos de tipo **String** son inmutables, es decir son fijos y no pueden modificarse. Cuando operamos con cadenas, obtenemos otro objeto de tipo **String**, los objetos originales quedan inalterables. Esta característica que parece una desventaja, no lo es porque de esta manera su implementación es más eficiente. Para los casos en que se realice una cadena modificable, Java proporciona dos opciones: **StringBuffer** y **StringBuilder**, estas implementan cadenas que pueden modificarse.

Las clases **String**, **StringBuffer** y **StringBuilder** se definen en el paquete **Java.lang**, y se declaran como **final**; significa que no se pueden crear subclases a partir de ellas. Esto permite optimizaciones que aumentan su rendimiento en las operaciones con cadenas. Si bien es cierto que un objeto de tipo **String** no puede modificarse, sin embargo, una

variable de tipo **String** puede cambiar en cualquier momento para que apunte a otro objeto de tipo **String**; de manera que el siguiente código sea perfectamente válido:

```
C:\java2n100\cap16>javac prog1606.java                    [Enter]

C:\java2n100\cap16>java prog1606 mensaje.txt copia.txt    [Enter]
Inicio de Copia.
Fin de Copia.
```

17.2 CONSTRUCTORES DE LA CLASE STRING

La clase **String** admite varios constructores. A continuación tenemos los prototipos de estos constructores, una breve descripción de ellos y un ejemplo ilustrativo.

public String()

Crea un objeto de tipo **String** con una cadena vacía. Crear un objeto **String** vacío no es útil, puesto que los objetos **String** son inmutables.

- **Ejemplo 1**

```
String cadena = new String();
```

public String(String original)

Crea un objeto de tipo **String** y lo inicializa con la cadena que recibe como argumento.

- **Ejemplo 2**

```
String cadena = new String("Esta es la cadena");
```

public String(char[] value)

Crea un nuevo objeto de tipo **String** a partir del arreglo de caracteres que recibe como argumento.

- **Ejemplo 3**

```
char lista[] = {'P','E','R','U'};
String cadena = new String(lista); // La cadena es "PERU"
```

public String(char[] value, int offset, int count)

Crea un nuevo objeto de tipo **String** a partir de un arreglo de caracteres que recibe como argumento, pero también debe indicarse la posición de inicio y la cantidad de caracteres que debe tomar en cuenta.

- **Ejemplo 4**

```
char lista[] = {'P','E','R','U'};
String cadena = new String(lista,1,2); // La cadena es "ER"
```

public String(int[] codePoints, int offset, int count)

Crea un objeto de tipo **String** desde un arreglo de códigos **Unicode**. Debe indicarse la posición de inicio y la cantidad de códigos para crear el objeto.

- **Ejemplo 5**

```
int lista[] = {65,66,67,68};
String cadena = new String(lista,1,2);   // La cadena es "BC"
```

public String(byte[] bytes)

Crea un nuevo objeto de tipo **String** según la codificación especificada en el arreglo de tipo **byte**, usando el conjunto de caracteres por defecto de la plataforma.

- **Ejemplo 6**

```
byte lista[] = {65,66,67,68};
String cadena = new String(lista); //   La cadena es "ABCD"
```

public String(byte[] bytes, int offset, int length)

Crea un nuevo objeto de tipo **String** según la codificación especificada en el arreglo de tipo byte, usando el conjunto de caracteres por defecto de la plataforma. Se debe especificar la posición inicial y la cantidad de bytes a tomar en cuenta para crear el objeto.

- **Ejemplo 7**

```
byte lista[] = {65,66,67,68};
String cadena = new String(lista,1,2);  // La cadena es "BC"
```

public String(StringBuffer buffer)

Crea un nuevo objeto de tipo **String** con la secuencia de caracteres contenidos en un objeto de tipo **StringBuffer**.

- **Ejemplo 8**

```
StringBuffer sb = new StringBuffer("PERU");
String cadena = new String(sb);      // La cadena es "PERU"
```

public String(StringBuilder builder)

Crea un nuevo objeto de tipo **String** con la secuencia de caracteres contenidos en un objeto de tipo **StringBuilder**.

- **Ejemplo 9**

```
StringBuilder sb = new StringBuilder("Ver Para Creer");
String cadena = new String(sb);      // La cadena es "Ver Para Creer"
```

17.3 OPERACIONES CON CADENAS

Para operar con cadenas debemos utilizar los métodos definidos en la clase **String**. Daremos una descripción de cada uno de ellos.

1. length()

- Prototipo

```
public int length()
```

Retorna la longitud de una cadena.

- **Ejemplo 10**

```
String cadena = "www.perudev.com";
System.out.println(cadena.length());
```

En este ejemplo se imprime el valor 15.

2. charAt()

- Prototipo

```
public char charAt(int index)
```

Retorna el carácter especificado en **index**. El rango de **index** va desde **0** hasta **length()-1**.

- **Ejemplo 11**

```
String cadena = "gcoronel@viabcp.com";
System.out.println(cadena.charAt(3));
```

Aquí se imprime el carácter **r**.

3. codePointAt()

- Prototipo

```
public int codePointAt(int index)
```

Retorna el código del carácter especificado en **index**. El rango de **index** va desde **0** hasta **length()-1**.

- **Ejemplo 12**

```
String cadena = "gcoronel@viabcp.com";
System.out.println(cadena.codePointAt(3));
```

Aquí se imprime **114**, que es el código de la letra **r**.

4. codePointBefore()

- Prototipo

```
public int codePointBefore(int index)
```

Retorna el código del carácter que se encuentra antes de **index**. El rango de **index** va desde **0** hasta **length()-1**.

- **Ejemplo 13**

```
String cadena = "gcoronel@viabcp.com";
System.out.println(cadena.codePointBefore(3));
```

Aquí se imprime **111**, que es el código de la letra **o**.

5. codePointCount()

- Prototipo

```
public int codePointCount(int beginIndex, int endIndex)
```

Retorna el número de caracteres desde **beginIndex** hasta **endIndex -1**. Los valores de **beginIndex** y **endIndex** deden estar en el rango de **0** hasta **length()-1**.

- **Ejemplo 14**

```
String cadena = "gcoronel@viabcp.com";
System.out.println(cadena.codePointBefore(3));
```

En este caso se imprime el valor **3**.

6. getChars()

- Prototipo

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Copia caracteres hacia un arreglo de caracteres que se especifican en **dst**. Los caracteres copiados están especificados por el rango **srcBegin** hasta **srcEnd** y son copiados en destino a partir de la posición **dstBegin**.

- **Ejemplo 15**

```
char destino[] = new char[20];
String cadena = "www.perudev.com";
cadena.getChars(4, 11, destino, 5);
for(int p = 5, n = 4; n < 11; n++, p++)
    System.out.print(destino[p]);
```

En este caso, se copia **perudev** en **destino** a partir del índice **5**, luego se imprime la cadena **perudev**.

7. getBytes()

- Prototipos

```
public byte[] getBytes()

public byte[] getBytes(String charsetName) throws UnsupportedEncodingException
```

Se trata de un método sobrecargado, que permite almacenar los caracteres de una cena en un arreglo de bytes. Utiliza por defecto el conjunto de caracteres que proporciona la plataforma o en su defecto la que se indique como argumento.

- **Ejemplo 16**

```
String cadena = "www.perudev.com";
byte destino[] = cadena.getBytes();
for(int p = 0; p < destino.length; p++)
    System.out.print((char)destino[p]);
```

En este ejemplo, se copia el contenido de **cadena** en **destino**, luego se imprime cada elemento de **destino**, pero como carácter. Se imprime la cadena **www.perudev.com**.

8. equals()

- Prototipo

```
public boolean equals(Object anObject)
```

Se utiliza para comparar dos cadenas; compara la cadena que pasa a través de **anObject** con la cadena que realiza la llamada.

- **Ejemplo 17**

```
String cadena  = "perudev";
String cadena1 = "PeruDev";
String cadena2 = null;
String cadena3 = "perudev";
boolean rpta1, rpta2, rpta3;

rpta1 = cadena.equals(cadena1);
rpta2 = cadena.equals(cadena2);
rpta3 = cadena.equals(cadena3);

System.out.println(rpta1);
System.out.println(rpta2);
System.out.println(rpta3);
```

En este ejemplo se compara **cadena** con **cadena1**, **cadena2**, y **cadena3**; los resultados se almacenan en **rpta1**, **rpta2** y **rpta3** respectivamente. El resultado que se obtiene es el siguiente:

```
false
false
true
```

En el primer caso es **false** porque considera diferentes las mayúsculas de las minúsculas. Si queremos que no haga esta distinción debemos utilizar el método **equalsIgnoreCase()**. El segundo caso es **false** por que **cadena2** es **null**. El tercer caso es **true** porque realmente las cadenas son iguales.

9. equalsIgnoreCase()

- Prototipo

```
public boolean equalsIgnoreCase(String anotherString)
```

Es similar al método **equals()**, pero ignora las diferencias entre mayúsculas y minúsculas.

- **Ejemplo 18**

```
String cadena  = "perudev";
String cadena1 = "PeruDev";
String cadena2 = null;
String cadena3 = "perudev";
boolean rpta1, rpta2, rpta3;

rpta1 = cadena.equalsIgnoreCase(cadena1);
rpta2 = cadena.equalsIgnoreCase(cadena2);
rpta3 = cadena.equalsIgnoreCase(cadena3);

System.out.println(rpta1);
System.out.println(rpta2);
System.out.println(rpta3);
```

El resultado que se obtiene en este caso es:

```
true
false
true
```

10. compareTo()

- Prototipo

```
public int compareTo(String anotherString)
```

Se utiliza para comparar la cadena actual con **anotherString**. Nos informa si es mayor, menor o igual; para que sea mayor debe estar delante de **anotherString** en orden alfabético.

A continuación, tenemos una descripción de los resultados que se obtienen:

Valor	Significado
Mayor que 0	La cadena que llama al método es mayor que **anotherString**.
Menor que 0	La cadena que llama al método es menor que **anotherString**.
0	Las dos cadenas son iguales.

- **Ejemplo 19**

```
String cadena1 = "perudev.com";
String cadena2 = "perudev";

int p = cadena1.compareTo(cadena2);
System.out.println("Rpta: " + p);
```

La respuesta en este ejemplo es **4** porque **cadena1** es de mayor longitud que **cadena2**.

- **Ejemplo 20**

```
String cadena1 = "peruDev.com";
String cadena2 = "perudev";

int p = cadena1.compareTo(cadena2);
System.out.println("Rpta: " + p);
```

La respuesta en este ejemplo es **-32** porque **cadena1** es menor que **cadena2**; la diferencia está en la letra **D** que es alfabéticamente menor que **d**.

11. compareToIgnoreCase()

- Prototipo

```
public int compareToIgnoreCase(String str)
```

Este método es similar a **compareTo()**, la diferencia está en que este método no hace diferencia entre mayúsculas y minúsculas.

- **Ejemplo 21**

```
String cadena1 = "peruDev";
String cadena2 = "perudev";

int p = cadena1.compareToIgnoreCase(cadena2);
System.out.println("Rpta: " + p);
```

La respuesta es **0** puesto que este método no hace diferencias entre mayúsculas y minúsculas.

12. regionMatches()

- Prototipos

```
public boolean regionMatches(int toffset, String other, int ooffset, int len)

public boolean regionMatches(boolean ignoreCase, int toffset, String other,
int ooffset, int len)
```

Se trata de una versión especializada del método **equals()**. Se utiliza para comparar una región especifica que parte de una cadena con otra región de otra cadena. Hay dos variantes de **regionMatches()**, una le permite controlar si es importante la diferenciación entre mayúsculas y minúsculas; la otra asume que sí lo es.

En estas dos versiones de **regionMatches()**, el parámetro **toffset** indica el desplazamiento en caracteres en el objeto **String** sobre el que estamos llamando el método. La cadena con la que se compara se llama **other**, y el desplazamiento dentro de esa cadena se llama **ooffset**. Se comparan **len** caracteres de las dos cadenas comenzando a partir de los dos desplazamientos.

- **Ejemplo 22**

```
String cadena1 = "www.perudev.com";
String cadena2 = "El sitio peruDev es lo maximo.";

boolean rpta = cadena1.regionMatches(4, cadena2, 9, 7);
System.out.println("Rpta: " + rpta);
```

Aquí se está comparando exactamente la palabra **perudev** de **cadena1** contra **peruDev** de **cadena2**; la respuesta es **false**, la diferencia está en la letra **D**.

- **Ejemplo 23**

```
String cadena1 = "www.perudev.com";
String cadena2 = "El sitio peruDev es lo maximo.";

boolean rpta = cadena1.regionMatches(true,4, cadena2, 9, 7);
System.out.println("Rpta: " + rpta);
```

En este otro ejemplo, se está utilizando la segunda versión del método **regionMatches()**, en el primer argumento se esta ingresando el valor **true**, esto indica que ignore la diferencia entre mayúsculas y minúsculas, por lo tanto, el resultado en este caso es **true**.

13. startsWith()

- Prototipos

```
public boolean startsWith(String prefix)

public boolean startsWith(String prefix, int toffset)
```

Este método permite comprobar si un objeto de tipo **String** empieza con una cadena especificada en **prefix**. En la segunda versión de este método, se puede especificar la posición (**toffset**) a partir de la cual se inició la comprobación, esto puede comprobarse en los siguientes los ejemplos:

- **Ejemplo 24**

```
String cadena1 = "www.perudev.com";

boolean rpta = cadena1.startsWith("www");
System.out.println("Rpta: " + rpta);
```

El resultado es **true**, porque **cadena1** empieza con **www**.

- **Ejemplo 25**

```
String cadena1 = "www.perudev.com";

boolean rpta = cadena1.startsWith("perudev"5);
System.out.println("Rpta: " + rpta);
```

El resultado es **false** porque **cadena1** no empieza con **perudev**.

- **Ejemplo 26**

```
String cadena1 = "www.perudev.com";

boolean rpta = cadena1.startsWith("perudev", 4);
System.out.println("Rpta: " + rpta);
```

La respuesta es **true** porque a partir de la posición **4** (inicia en 0) **cadena1** si empieza con **perudev**.

14. endsWith()

- Prototipo

```
public boolean endsWith(String suffix)
```

Este método permite comprobar si un objeto de tipo **String** termina con una cadena específica en **suffix**.

- **Ejemplo 27**

```
String cadena1 = "www.perudev.com";

boolean rpta = cadena1.endsWith("com");
System.out.println("Rpta: " + rpta);
```

La respuesta es **true** porque **cadena1** finaliza en **com**.

15. indexOf()

- Prototipos

```
public int indexOf(int ch)

public int indexOf(int ch, int fromIndex)

public int indexOf(String str)

public int indexOf(String str, int fromIndex)
```

Este método permite buscar la primera ocurrencia de un carácter o subcadena dentro de una cadena. El parámetro **fromIndex** en la segunda versión de este método para cada uno de los casos respectivamente, permite indicar la posición desde donde se inicia la búsqueda.

- **Ejemplo 28**

```
String cadena1 = "www.perudev.com";

int rpta = cadena1.indexOf('e');
System.out.println("Rpta: " + rpta);
```

Se busca el carácter **e** en **cadena1**. El resultado es **5**.

- **Ejemplo 29**

```
String cadena1 = "www.perudev.com";

int rpta = cadena1.indexOf('e', 6);
System.out.println("Rpta: " + rpta);
```

También se busca el carácter **e**, pero a partir de la posición **6**. El resultado es **9**.

- **Ejemplo 30**

```
String cadena1 = "si no sabes sonreír, es que no sabes vivir";

int rpta = cadena1.indexOf("sabes");
System.out.println("Rpta: " + rpta);
```

Se busca la cadena "**sabes**" en **cadena1**. El resultado es **6**.

- **Ejemplo 31**

```
String cadena1 = "si no sabes sonreír, es que no sabes vivir";

int rpta = cadena1.indexOf("sabes", 10);
System.out.println("Rpta: " + rpta);
```

Se busca la cadena "**sabes**" en **cadena1**, pero a partir de la posición **10**. El resultado es **31**.

16. lastIndexOf()

- Prototipos

```
public int lastIndexOf(int ch)

public int lastIndexOf(int ch, int fromIndex)

public int lastIndexOf(String str)

public int lastIndexOf(String str, int fromIndex)
```

Busca la última aparición de un carácter o una subcadena dentro de una cadena. El parámetro **fromIndex** en la segunda versión de este método para cada uno de los casos respectivamente, permite indicar la posición desde donde se inicia la búsqueda. La búsqueda en este caso es desde el final hacia el inicio de la cadena.

- **Ejemplo 32**

```
String cadena1 = "www.perudev.com";

int rpta = cadena1.lastIndexOf('e');
System.out.println("Rpta: " + rpta);
```

Se busca la última aparición del carácter **e** en **cadena1**. La respuesta es **9**.

- **Ejemplo 33**

```
String cadena1 = "www.perudev.com";

int rpta = cadena1.lastIndexOf('e', 8);
System.out.println("Rpta: " + rpta);
```

Se busca la última aparición del carácter e en **cadena1**, pero la búsqueda se inicia en la posición **8**, y continúa hacia la posición **0**. El resultado es **5**.

- **Ejemplo 34**

```
String cadena1 = "si no sabes sonreír, es que no sabes vivir";

int rpta = cadena1.lastIndexOf("sabes");
System.out.println("Rpta: " + rpta);
```

Se busca la última aparición de la cadena "**sabes**" en **cadena1**. La respuesta es **31**.

- **Ejemplo 35**

```
String cadena1 = "si no sabes sonreir, es que no sabes vivir";

int rpta = cadena1.lastIndexOf("sabes",20);
System.out.println("Rpta: " + rpta);
```

También se busca la última aparición de la cadena "**sabes**", pero la búsqueda se inicia en la posición **20**, y continúa hacia la posición **0**. La respuesta es **6**.

17. substring()

- Prototipos

```
public String substring(int beginIndex)

public String substring(int beginIndex, int endIndex)
```

Este método permite extraer una subcadena de la cadena que llama al método. El parámetro **beginIndex** permite especificar donde comienza la subcadena a extraer, y el parámetro **endIndex** permite especificar donde finaliza la subcadena a extraer. Cuando no se especifica **endIndex** la subcadena finaliza en el último carácter de la cadena que llama al método.

- **Ejemplo 36**

```
String cadena1 = "Nunca se esta solo cuando se tiene un amigo.";

String subcad = cadena1.substring(19);
System.out.println(subcad);
```

La subcadena a extraer empieza en la posición **19**, hasta el final de la frase. El resultado es "**cuando se tiene un amigo.**".

- **Ejemplo 37**

```
String cadena1 = "Nunca se esta solo cuando se tiene un amigo.";

String subcad = cadena1.substring(19, 34);
System.out.println(subcad);
```

La subcadena a extraer empieza en la posición **19**, y finaliza en la posición **34**. El resultado es "**cuando se tiene**".

18. concat()

- Prototipo

```
public String concat(String str)
```

Este método permite concatenar la cadena que llama al método con otra cadena, que se pasa como a través del parámetro **str**.

- **Ejemplo 38**

```
String cadena1 = "www.perudev";
String cadena2 = ".com";
String cadena3 = cadena1.concat(cadena2);

System.out.println(cadena3);
```

En **cadena3** se está guardando la concatenación de **cadena1** y **cadena2**. El resultado es "**www.perudev.com**".

- **Ejemplo 39**

```
String cadena = "www.".concat("perudev").concat(".com");

System.out.println(cadena);
```

Se están concatenando tres cadenas, el resultado se almacena en la variable cadena. El resultado es "**www.perudev.com**".

19. replace()

- **Prototipo**

```
public String replace(char oldChar, char newChar)

public String replace(CharSequence target, CharSequence replacement)
```

Reemplaza todas las apariciones de un carácter o subcadena en la cadena que realiza la llamada al método por otro carácter o subcadena.

- **Ejemplo 40**

```
String cadena1 = "www-perudev-com";
String cadena2 = cadena1.replace('-', '.');

System.out.println(cadena2);
```

Aquí estamos reemplazando el carácter guion (–) por el punto (.). El resultado es "**www.perudev.com**".

- **Ejemplo 41**

```
String cadena1 = "La triste noche del triste día.";
String cadena2 = cadena1.replace( "triste", "alegre" );

System.out.println(cadena2);
```

Reemplaza la palabra **triste** por **alegre** en la frase contenida en **cadena1**, el resultado es asignado a **cadena2**. La cadena resultante es "**La alegre noche del alegre día**".

20. toLowerCase()

- Prototipos

```
public String toLowerCase()

public String toLowerCase(Locale locale)
```

Este método permite convertir los caracteres de una cadena a minúsculas.

- **Ejemplo 42**

```
String cadena1 = "WWW.PERUDEV.COM";
String cadena2 = cadena1.toLowerCase();

System.out.println(cadena2);
```

Se convierte la cadena "**WWW.PERUDEV.COM**" a minúsculas, el resultado es "**www.perudev.com**".

21. toUpperCase()

- Prototipos

```
public String toUpperCase()

toUpperCase()
```

Este método permite convertir los caracteres de una cadena a mayúsculas.

- **Ejemplo 43**

```
String cadena1 = "www.perudev.com";
String cadena2 = cadena1.toUpperCase();

System.out.println(cadena2);
```

Se convierte la cadena "**www.perudev.com**" a mayúsculas, el resultado es "**WWW.PERUDEV.COM**".

22. trim()

- Prototipo

```
public String trim()
```

Este método retorna una copia de la cadena que realiza la llamada, pero eliminando los espacios en blanco que se encuentran al inicio y al final.

- **Ejemplo 44**

```
String cadena1 = "   Bienvenido   a  Java   ";
String cadena2 = cadena1.trim();

System.out.println(cadena2);
```

Al aplicar el método **trim()** a **cadena1** se eliminan los espacios en blanco que se encuentran al inicio y al final. El resultado que se obtiene es "**Bienvenido a Java**".

23. toString()

- Prototipo

```
public String toString()
```

Este es un método que lo heredan todas las clases de la clase **Object**, y normalmente lo sobreescriben para que retorne una descripción más adecuada del objeto. En el caso de la clase String, retorna la misma cadena. Cuando utilizamos un objeto, por ejemplo, directamente en **print()** o **println()**, así como en una expresión de concatenación; implícitamente se está llamando al método **toString()**.

- **Ejemplo 45**
  Se ha creado la clase **Box**, se está sobreescribiendo el método **toString()**. Cualquier instancia que utilizemos de esta clase, directamente en **print()** o **println()**, llamará automáticamente al método **toString()**.

```
public class Box {

    private int ancho = 0;
    private int altura = 0;
    private int area = 0;

    public Box(int ancho, int altura) {
            this.ancho = ancho;
            this.altura = altura;
            this.area = this.ancho * this.altura;
    }

    // Sobreescribiendo el método toString
    public String toString(){
            return( "Ancho: " + ancho + " Altura: " + altura + " Area: " +
area);
    }
}
```

En el siguiente código, se está creando una instancia de **Box** y la envía directamente al método **println()**.

```
String cadena = "Bienvenido a Java";
Box obj = new Box(5, 8);

System.out.println(cadena);
System.out.println(obj);
```

El resultado de ejecutar este código es:

```
Bienvenido a Java
Ancho: 5 Altura: 8 Area: 40
```

Como puede comprobarse, se ha ejecutado llamado al método **toString()** de **obj**, sin haberlo llamado explícitamente.

24. toCharArray()

- Prototipo

```
public char[] toCharArray()
```

Convierte un objeto **String** a un arreglo de tipo **char**.

- **Ejemplo 46**

```
String cadena = "www.perudev.com";
char lista[] = cadena.toCharArray();

for( int k = lista.length - 1; k >= 0; k-- )
    System.out.print(lista[k]);
```

Imprime la cadena "**www.perudev.com**" invertida. El resultado es "**moc.vedurep.www**".

25. valueOf()

- Prototipos

```
public static String valueOf(Object obj)

public static String valueOf(char[] data)

public static String valueOf(char[] data, int offset, int count)

public static String valueOf(boolean b)

public static String valueOf(char c)

public static String valueOf(int i)

public static String valueOf(long l)

public static String valueOf(float f)

public static String valueOf(double d)
```

Este método pasa los datos de su formato interno a un formato legible. Es un método estático que se sobrecarga dentro de la clase **String** para todos los tipos que incorpora Java; de modo que cada tipo puede convertirse en una cadena de la forma más adecuada. También existe una sobrecarga para el tipo **Object**, de manera que un objeto de cualquier tipo de clase también pueda usarse como argumento.

- **Ejemplo 47**

```
int n1 = 15;
double n2 = 45.34;
String cad = "Datos: " + String.valueOf(n1) + ", " + String.valueOf(n2);

System.out.print(cad);
```

Aquí se ilustra el uso del método **valueOf()**. El resultado es la siguiente cadena: **"Datos:15, 45.34"**.

- **Ejemplo 48**

```
char lista[] = {'G','u','s','t','a','v','o'};
String cad = String.valueOf(lista);

System.out.print(cad);
```

Se está pasando un arreglo de caracteres a una cadena. El resultado es "**Gustavo**".

26. copyValueOf()

- Prototipos

```
public static String copyValueOf(char[] data)

public static String copyValueOf(char[] data, int offset, int count)
```

Este método pasa un arreglo de caracteres a una cadena. Se puede especificar la posición a través del parámetro **offset** y la cantidad de caracteres en el parámetro **count**.

- **Ejemplo 49**

```
char lista[] = {'p','e','r','u','d','e','v'};
String cad = String.copyValueOf(lista, 2, 2);

System.out.print(cad);
```

Aquí está pasando de la posición **2** del arreglo **lista**, **2** caracteres a una cadena. La respuesta es "**ru**".

Enlaces web

- **http://www.comunidadjava.com.ar/forums.html**
  Comunidad Java de Argentina.

- **http://www.kickjava.com/freeBooks.html**
  Sitio web donde encontrará manuales referentes a Java.

- **http://www.javahispano.org/**
  Sitio web dedicado a Java, sobre todo en español.

- **http://www.theserverside.com/tss**
  Sitio web dedicado a Java.

- **http://javahispano.net/**
  Sitio web desde donde podrá bajar código libremente.

- **http://www.elistas.net/lista/todojava**
  Lista de interés relacionado a Java.

- **http://es.groups.yahoo.com/group/javaSpain/**
  Lista de interés sobre programación con Java.

- **http://gcoronelc.blogspot.com**
  Blog personal de Gustavo Coronel Castillo.

- **http://java.sun.com/docs/index.html**
  Desde este enlace tiene acceso a descargar la documentación del producto.

- **http://java.sun.com/j2se/1.5.0/docs/**
  Desde este enlace tiene acceso a la documentación en línea del producto.

Bibliografía

- Bruce, E. (2007). *Piensa en Java*. 4.ª edición. Madrid: Pearson Educación S. A.
- Flanagan, D. & McLaughlin, B. (2004). *Java 1.5 Tiger: A Developer's Notebook*. California: Editorial O'Reilly.
- Schildt, H. (2005). *La biblia de Java 2 v5.0*. Salamanca: Anaya.

Impreso en los talleres gráficos de

Surquillo