ACERCA DEL AUTOR

Con más de 15 años de experiencia, actualmente trabaja como Ingeniero de Software en QuEST Global Engineering España.

Ingeniero Técnico en Informática de Sistemas y Máster en Desarrollo de Videojuegos por la Facultad de Informática de la Universidad Complutense de Madrid.

A lo largo de su carrera, ha compaginado el ejercicio del magisterio privado personalizado para lograr que los alumnos alcancen sus metas con su labor en la empresa privada. Experto en Banca Electrónica y Comercio Electrónico, ha realizado aplicaciones para la inmensa mayoría de los grandes bancos mundiales.

Escritor de artículos:

- En febrero de 2007, en la revista Sólo Programadores, titulado "¿Es viable una aplicación de escritorio con Java?".
- En Agosto de 2011, en la página web de javaHispano, titulado "Multitarea en Swing".
- En Agosto de 2011, en la página web de javaHispano, titulado "Tipos Abstractos de Datos y Diseño por Contrato".

Autor de libros:

- IFCD052PO Programación en Java, Ed. Ra-Ma, 2021.
- Java Curso Práctico, Ed. Ra-Ma, 2020.

INTRODUCCIÓN

El **hardware** es la parte física de un computador, mientras que el **software** es la parte lógica del mismo.

Un **Ingeniero de Software** es aquella persona que construye un sistema software correcto y eficiente para ser ejecutado por el hardware de un computador.

Construir software es divertido y gratificante. En cierto modo, nos convierte en creadores de mundos nuevos. Pero también es difícil, porque los computadores tienen la mala costumbre de hacer lo que les decimos que hagan, no lo que queremos que hagan.

El camino para aprender a construir software correcto y eficiente es duro, pero estoy seguro de que libros como el que estás leyendo ahora mismo pueden ayudarte a hacerlo más fácil.

El libro que estás leyendo está actualizado para Java 17 y JUnit 5. Se dirige a aquellos programadores que quieren poner a trabajar la tecnología Java en proyectos reales, para lo cual se estudian herramientas y técnicas metódicas que permiten el desarrollo de software fiable y eficiente.

El libro está pensado para ser leído secuencialmente, porque cada capítulo se construye sobre los conceptos aprendidos en los capítulos previos. No obstante, el lector que ya conozca determinados contenidos puede saltar directamente a los capítulos que le resulten desconocidos.

Es preciso tener en cuenta que lo normal no es entender todos los conceptos presentados en el libro simplemente leyendo el texto. Es importante estudiar el código, escribirlo, ejecutarlo e incluso puede que depurarlo para llegar a entenderlo.

Para los que deseen profundizar aún más en este lenguaje, el autor tiene publicada otra obra con el título *Java 17 Programación Avanzada* que amplía y complementa los contenidos de ésta.

El código fuente que aparece en el libro está disponible para descargar en:

https://github.com/josemari/JavaCursoPracticoProjects

Y en la web del libro en: *www.ra-ma.com*

Incluye varios proyectos Maven que pueden ser importados en Eclipse. Cualquier versión de este entorno de desarrollo integrado debería servir, siempre y cuando soporte Java 17.

Me encantaría tener noticia de cualquier errata que exista en el libro o en el código. Para informar de una errata, está disponible el siguiente correo electrónico:

jomaveger@gmail.com

Al final de la presente obra, hay una sección dedicada a la bibliografía consultada para la elaboración de este manual. Quiero expresar mi más profundo agradecimiento tanto a los autores de dichas obras de consulta, como a las numerosas fuentes de internet que han ayudado a que este libro sea una realidad, tan amplias que son de difícil enumeración.

1

INTRODUCCIÓN A JAVA

1.1 INSTALACIÓN DE JAVA, MAVEN Y ECLIPSE

El primer paso de nuestra aventura consiste en preparar el entorno de trabajo. Lo primero que necesitamos es tener instalado un JDK compatible en el computador. Instalaremos la última versión que tiene soporte extendido por ser la más estable, Java 17.

El JDK puede ser descargado del sitio web de Oracle en la dirección:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

O bien es posible emplear una implementación de código abierto del JDK denominada OpenJDK, que se puede descargar de:

http://openjdk.java.net/

Una vez instalado el JDK, el siguiente paso es establecer apropiadamente la variable de entorno **JAVA_HOME**.

Así, en Mac OS, en caso de no existir para nuestro usuario el fichero **.profile**, lo primero que hay que hacer es crear uno vacío mediante la siguiente instrucción ejecutada en la ventana del terminal:

```
touch ~/.profile
```

A continuación, utilizamos el editor de textos **TextEdit** para modificar el fichero:

```
open -a TextEdit ~/.profile
```

Y añadimos la siguiente línea al fichero:

```
export JAVAHOME=$(/usr/libexec/javahome)
```

donde **/usr/libexec/java_home** retorna la versión actual de Java instalada en Mac OS.

Una vez hemos guardado los cambios y salido del editor de texto, ejecutamos la siguiente instrucción para aplicar los cambios al fichero **.profile**:

```
source ~/.profile
```

Si ahora ejecutamos en la ventana del terminal la instrucción

```
echo $JAVA_HOME
```

obtendremos algo semejante a lo siguiente

```
/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home
```

También es posible obtener la versión actual de Java instalada en la máquina si ejecutamos la siguiente instrucción en la ventana del terminal

```
java -version
```

de modo tal que obtendremos algo parecido a lo siguiente

```
java version “17” 2021-09-14 LTS
Java(TM) SE Runtime Environment (build 17+35-LTS-2724)
Java HotSpot(TM) 64-Bit Server VM (build 17+35-LTS-2724, mixed mode, sharing)
```

En el caso de Windows, tenemos que pulsar en el botón derecho de *Equipo* y seleccionar *Propiedades*. A continuación, vamos a

```
Configuración avanzada del sistema -> Variables de entorno
```

Después, pulsamos en *Nueva (variable del sistema)*. En el *Nombre de la variable* escribimos **JAVA_HOME** y, en el *Valor de la variable*, escribimos el directorio donde se ha instalado el JDK.

Finalmente, en

```
Configuración avanzada del sistema -> Variables de entorno -> Variables del sis-
tema
```

se modifica la variable **PATH** añadiendo al final **%JAVA_HOME%\bin**. Si abrimos una ventana de la consola y ejecutamos la siguiente instrucción:

```
java -version
```

obtendremos algo parecido a lo siguiente:

```
java version "17" 2021-09-14 LTS
Java(TM) SE Runtime Environment (build 17+35-LTS-2724)
Java HotSpot(TM) 64-Bit Server VM (build 17+35-LTS-2724, mixed mode, sharing)
```

El siguiente paso es la instalación de Maven, una herramienta de gestión de proyectos que se puede descargar de la dirección siguiente:

https://maven.apache.org/download.cgi

Elegimos para descargar Maven en formato *.zip*. Una vez descargada la distribución de Maven, dado que es un archivo *.zip*, se descomprime dicho archivo en una carpeta y ya se ha finalizado todo el proceso de instalación. Es recomendable actualizar la variable de entorno **PATH** para que incluya la ruta del ejecutable de Maven.

En el caso de Mac OS, el archivo .zip se descomprime automáticamente una vez finaliza la descarga, así que después movemos a la ruta **/opt**:

```
sudo mv $HOME/Downloads/apache-maven-3.8.2 /opt
```

Finalmente, modificamos el fichero **.profile** para actualizar la variable de entorno **PATH**:

```
export PATH=$PATH:/opt/apache-maven-3.8.2/bin
```

De nuevo, no nos olvidamos de:

```
source ~/.profile
```

Si ahora ejecutamos en la ventana del terminal la instrucción

```
echo $PATH
```

obtendremos lo siguiente

```
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/apache-maven-3.8.2/bin
```

Asimismo, si ejecutamos en la ventana del terminal la instrucción

```
mvn -version
```

obtendremos lo siguiente

```
Apache Maven 3.8.2 (ea98e05a04480131370aa0c110b8c54cf726c06f)
Maven home: /opt/apache-maven-3.8.2
Java version: 17, vendor: Oracle Corporation, runtime: /Library/Java/JavaVir-
tualMachines/jdk-17.jdk/Contents/Home
Default locale: es_ES, platform encoding: UTF-8
OS name: “mac os x”, version: “11.5.2”, arch: “x86_64”, family: “mac”
```

Las dependencias descargadas por Maven se almacenarán por defecto en la carpeta situada en la siguiente ruta, que constituye el repositorio local de Maven:

```
$USER_HOME/.m2/repository
```

En caso de que queramos almacenar las mencionadas dependencias en otro destino, podemos indicarlo editando el fichero de configuración de Maven, **settings.xml**, que se encuentra en el subdirectorio **/conf** dentro de la carpeta de instalación de Maven; en nuestro caso, sería

```
/opt/apache-maven-3.8.2/conf
```

Lo siguiente que necesitamos es un entorno de desarrollo integrado. Vamos a utilizar **Eclipse IDE for Java Developers**, que podemos descargar de la siguiente dirección:

https://www.eclipse.org/downloads/packages/

Una vez descomprimida la distribución de Eclipse que ha sido descargada en una carpeta, se ha finalizado la instalación. En el caso de Mac OS, al descomprimir el archivo descargado -de tipo **.dmg-**, se obtiene una aplicación que copiamos a la carpeta de aplicaciones del sistema operativo.

En Windows, se ejecuta Eclipse desde el fichero ejecutable *eclipse.exe* que viene con la distribución.

Dentro del menú de Eclipse, buscamos la opción **Preferences -> Maven**. En caso de que la versión de Maven que viene con Eclipse sea inferior a la que hemos instalado en nuestra máquina, podemos añadir y configurar esta última en Eclipse.

También dentro del menú de Eclipse, buscamos la opción **Preferences -> Java -> Installed JREs**, y marcamos por defecto Java 17. A su vez, en la opción **Preferences -> Java -> Compiler**, seleccionamos *17* como **Compiler compliance level**.

1.2 INTRODUCCIÓN A MAVEN

Maven es una herramienta de gestión de proyectos que se ha convertido en un estándar de hecho, debido a que su utilización proporciona los siguientes beneficios:

- Una estructura de proyecto estándar y bien definida.
- Una gestión automática de las dependencias del proyecto.
- Una gestión integral del ciclo de vida del proyecto.
- Una rápida configuración del proyecto por medio de los arquetipos.

Maven utiliza convención en lugar de configuración. La estructura de proyecto que define Maven es una convención y se ha convertido en un estándar a nivel mundial para el desarrollo de proyectos Java. Por ejemplo, asumiendo que

```
${basedir}
```

indica la ubicación del proyecto en el disco duro, los proyectos Maven suelen tener las carpetas

```
${basedir}/src/main/java
${basedir}/src/main/test
```

En la primera, se coloca el código fuente y, en la segunda, el código de las pruebas. Maven es suficientemente inteligente para incluir en los archivos binarios que se van a distribuir solo las clases compiladas a partir de la carpeta *java* y no las que provienen de las pruebas. Otras carpetas habituales de los proyectos Maven son

```
${basedir}/src/main/resources
${basedir}/src/test/resources
```

En la primera, se almacenan ficheros de recursos necesarios para la ejecución de la aplicación y, en la segunda, se almacenan ficheros de recursos necesarios únicamente para la ejecución de las pruebas. También tenemos que las clases compiladas se almacenan en

```
${basedir}/target/classes
```

mientras que el artefacto generado se almacena en

```
${basedir}/target
```

En definitiva, Maven reconoce su estructura de proyecto al compilar y empaquetar los proyectos. Por ejemplo, coloca todos los ficheros de recursos en la raíz del archivo JAR resultante de la operación de empaquetado. De esta manera, se facilita muchísimo el desarrollo de un proyecto en múltiples entornos de desarrollo integrados.

Todo proyecto Maven tiene un fichero XML descriptor **pom.xml** en la raíz del proyecto que especifica todas sus características, también llamado simplemente fichero **POM**, donde **POM** significa *Modelo de Objetos del Proyecto* (del inglés *Project Object Model*).

Un proyecto genera un único artefacto, sea un archivo JAR, WAR o EAR. El artefacto o biblioteca generada necesita ser identificado para poder ser utilizado por otros proyectos. De ahí que se definan las coordenadas del proyecto como los

parámetros cuyos valores definen al proyecto de manera unívoca. Estas coordenadas del proyecto son el **groupId**, el **artifactId**, la **version** y el **packaging** y se definen en el **pom.xml**.

- La coordenada **groupId** indica el identificador único de la organización o grupo que ha creado el proyecto. Generalmente, se suele utilizar el nombre del dominio en internet de la organización, pero invirtiendo el orden de escritura de las palabras; es decir, si el dominio en internet fuera **maven.apache.org**, el **groupId** del proyecto creado sería **org.apache.maven**.

- La coordenada **artifactId** indica el prefijo único del nombre del artefacto que genera el proyecto. Generalmente, el nombre final del artefacto será de la forma:

```
<artifactId>-<version>.<extension>
```

- La coordenada **version** indica la versión del artefacto que genera el proyecto. Cuando una versión tiene el apéndice de **SNAPSHOT**, significa que el proyecto está en desarrollo.

- La coordenada **packaging** indica qué tipo de artefacto va a generar Maven a partir del proyecto. Si se omite esta coordenada, el empaquetado por defecto es **jar**. Otras opciones posibles son **ejb**, **war**, **ear** y **pom**. Lógicamente, en función del tipo de proyecto, el valor del empaquetado será uno u otro.

Un repositorio de Maven es un almacén de bibliotecas JAR y de ficheros descriptores POM. Cuando un proyecto define una dependencia en su fichero POM, Maven busca primero en el repositorio local y, si no la encuentra, la descarga del repositorio central. Como ya sabemos, el repositorio local por defecto de Maven es:

```
$USER_HOME/.m2/repository
```

Y el repositorio central de Maven es:

```
https://repo1.maven.org/maven2/
```

En el fichero descriptor POM, también se indican las dependencias, es decir, las bibliotecas necesarias para compilar o probar el proyecto. Una

biblioteca tiene distintas versiones de modo que la dependencia, a través de sus coordenadas, especifica qué versión nos interesa. Maven gestiona automáticamente las dependencias transitivas, de modo que descarga también las dependencias de las bibliotecas descargadas. Cuando se define una dependencia en el POM, existe también la posibilidad de definir su ámbito (en inglés *scope*). Existen varios ámbitos diferentes posibles:

- El ámbito **compile** es el ámbito por defecto, de modo que será el utilizado si no se especifica ámbito alguno. Toda dependencia con ámbito de compilación será incluida en la ruta de clases del proyecto y también será incluida en el artefacto final.
- El ámbito **test** indica que la dependencia sólo es necesaria para compilar y ejecutar las pruebas del proyecto, de modo que no será incluida en el artefacto final.
- El ámbito **provided** indica que la dependencia se utiliza durante las fases de compilación y pruebas, pero que no se incluyen en el artefacto final. Se utiliza a menudo para incluir los archivos JAR de JavaEE (como, por ejemplo, **servlet-api.jar**), ya que son necesarios para compilar pero, como ya están en el servidor de aplicaciones Java, no es necesario volver a incluirlos dentro del artefacto final.
- El ámbito **runtime** indica que la dependencia será necesaria durante la ejecución de la aplicación pero no al compilar.

Existen tres ciclos de vida de construcción en Maven: **default**, **clean** y **site**.

- El ciclo de vida **clean** controla la limpieza del proyecto, ya que se encarga de eliminar las clases compiladas y archivos binarios del proyecto.
- El ciclo de vida **default** controla el despliegue del proyecto, ya que genera las clases compiladas y archivos binarios del proyecto.
- El ciclo de vida **site** controla la generación de la página web de documentación del proyecto, ya que genera los ficheros .html que describen el proyecto.

Para ejecutar estos ciclos de vida, excepto en el caso de **default**, se debe poner **mvn ciclo**. Así, por ejemplo:

```
mvn clean
```

O bien:

```
mvn site
```

Cada uno de estos ciclos de vida está definido por una lista diferente de fases, donde cada fase representa un estado en el ciclo de vida. Las fases del ciclo de vida **default** son 23, pero no es necesario aprenderlas todas; entre las más relevantes, y en el orden en el que se ejecutan, están:

- La fase **validate**, que valida el proyecto.
- La fase **initialize** que configura propiedades y crea directorios.
- La fase **compile** que compila el código fuente del proyecto.
- La fase **test** que ejecuta las pruebas.
- La fase **package** que genera el artefacto del proyecto.
- La fase **verify** que verifica el artefacto generado.
- La fase **install** que instala el artefacto en el repositorio local.
- La fase **deploy** que sube el artefacto a un repositorio Maven en la red.

Para ejecutar una fase, basta escribir **mvn fase**. Una fase contiene a todas las fases anteriores. Es decir, cada una de estas fases se ejecuta secuencialmente para completar el ciclo de vida. Si, por ejemplo, invocamos la ejecución de la fase **deploy** del ciclo de vida **default**:

```
mvn deploy
```

Esta invocación también provocará la ejecución de todas las fases que la preceden.

Aunque parece complicado, en la práctica la mayoría de las veces se utiliza para compilar un proyecto la orden:

```
mvn clean install
```

Sin embargo, aunque una fase de construcción es responsable de un paso específico en el ciclo de vida de construcción, el modo en que éste se lleva a cabo puede

variar. Esto se consigue declarando objetivos (en inglés *goals*) de complementos (en inglés *plugins*) asociados a fases de construcción.

Un objetivo de un complemento es una tarea específica que se puede unir a alguna de las fases, aunque podría darse el caso de que un objetivo no estuviera unido a fase alguna y se ejecutara fuera del ciclo de vida; por ejemplo, mediante una llamada directa.

Pongamos como ejemplo la orden siguiente:

```
mvn clean dependency:copy-dependencies package
```

Los argumentos **clean** y **package** son fases de construcción, mientras que **dependency:copy-dependencies** es un objetivo de un complemento. Si la ejecutásemos, la fase **clean** se ejecutaría primero (lo cual quiere decir que ejecutará todas las fases precedentes del ciclo de vida **clean**, además de la propia fase **clean**), y luego el objetivo **dependency:copy-dependencies** (lo cual quiere decir que ejecuta el objetivo **copy-dependencies** del complemento de Maven **dependency**), antes de ejecutar finalmente la fase **package** (y todas sus fases precedentes del ciclo de vida **default**).

En función del tipo de empaquetado del proyecto, cambian los objetivos que Maven asocia por defecto a las diferentes fases del ciclo de vida.

Podemos desarrollar más aún estos conceptos.

Un objetivo es una tarea unitaria aislada que realiza algún tipo de trabajo real. Por ejemplo, el objetivo **compile** compila el código fuente de Java y se ejecuta de la forma siguiente

```
mvn compiler:compile
```

Todos los objetivos son proporcionados por complementos, sean complementos por defecto o bien complementos definidos por el usuario y configurados en el fichero POM.

Una fase es un grupo de objetivos ordenados o, en otras palabras, cero o más objetivos de complementos que están vinculados a una fase, bien por defecto o bien por el usuario. Por ejemplo, la fase **compile** de compilación, que se puede ejecutar como

```
mvn compile
```

consiste sólo en el objetivo del complemento siguiente:

```
compiler:compile
```

Ejecutar una fase consiste básicamente en ejecutar todos los complementos vinculados a ella.

El ciclo de vida de construcción es un grupo de fases ordenadas. Si ejecutamos una fase, todas las fases anteriores incluyendo dicha fase serán ejecutadas. Una fase con cero objetivos de complementos asociados no realiza tarea alguna.

Como hemos comentado, en función del empaquetado del proyecto, diferentes objetivos serán asociados a diferentes fases del ciclo de vida por Maven.

La diferencia entre objetivo y complemento es clara:

- Un complemento puede tener múltiples objetivos.
- Un complemento puede no estar necesariamente vinculado a una fase.
- Cuando un objetivo de un complemento está asociado a una fase de un ciclo de vida, es básicamente para extender funcionalidad no encontrada todavía en ese ciclo de vida.
- Una fase de un ciclo de vida puede estar asociada a uno o más objetivos de un complemento.
- Un objetivo es a veces llamado *Mojo*, del inglés *Maven plain Old Java Object*.

Cada versión generada de un artefacto software debe estar etiquetada a través de un número de versión. Dicho número de versión debe seguir un formato específico para que así cada versión del artefacto sea identificada de manera única, y además pueda aportar información de su naturaleza y objetivo. En general, se expresa que a una versión de un artefacto software se le asigna una versión X.Y.Z, de modo que:

- X representa el número de versión mayor.
- Y representa el número de versión menor.
- Z representa el número de parche de la versión.

Cuando se genera una nueva versión del artefacto software, se decide si se incrementa el número de versión mayor, menor o el número de parche en función de

la naturaleza del cambio. Así, dada una versión con número de versión X.Y.Z y una nueva versión a partir de ella:

- Se incrementa X si la nueva versión incluye cambios de API incompatibles.
- Se incrementa Y si se trata de evolutivos o cambios retrocompatibles con la versión X.Y.Z.
- Se incrementa Z cuando se trata de correcciones de errores en la versión X.Y.Z.

Es importante tener en cuenta lo siguiente:

- El incremento de los números de versión mayor, menor y número de parche es secuencial.
- Si se incrementa la versión mayor, se ponen a cero los valores de la versión menor y del número de parche de la versión. Es decir, la nueva versión mayor de la versión 3.2.1 es la 4.0.0.
- Si se incrementa la versión menor, se pone a cero el número de parche de la versión. Es decir, la nueva versión menor de la versión 3.2.1 es la 3.3.0.
- Si se incrementa el número de parche de la versión, los números de versión mayor y de versión menor no se modifican. Es decir, un parche sobre la versión 3.2.1 genera una nueva versión con número de versión 3.2.2.
- Pueden incluirse etiquetas en las versiones del artefacto que actualmente están en desarrollo. Es decir, 3.0.0-alpha o también 3.0.0-SNAPSHOT.

Un concepto particularmente útil que presenta Maven es el de arquetipo. En esencia, un arquetipo es una plantilla para la creación de una clase de proyectos. Existen multitud de arquetipos de modo que, empleando Eclipse, podemos seleccionar el arquetipo a partir del cual queremos que nos cree el proyecto.

Del menú **File**, se selecciona **New**, y luego **Project**. En la ventana que aparece de **New Project**, seleccionamos **Maven Project** y pulsamos **Next**. En la siguiente ventana de configuración, si seleccionamos la opción **Create a simple project**, Eclipse indicará a Maven que utilice el arquetipo por defecto -que crea un proyecto Java simple- y, por tanto, no nos permitirá escoger qué arquetipo queremos usar para crear el proyecto.

La ventana de configuración también nos permite decidir si se utiliza o no la ruta por defecto del espacio de trabajo actual para este nuevo proyecto. El concepto

de espacio de trabajo es esencial en Eclipse. Cada vez que se ejecuta el entorno de desarrollo integrado Eclipse, lo hace sobre un determinado espacio de trabajo, que no es más que un directorio o carpeta de la máquina en la que estamos trabajando. Este espacio de trabajo se selecciona cuando se inicia Eclipse. No obstante, se puede cambiar mientras se está ejecutando el entorno de desarrollo; ahora bien, en caso de modificarlo, habría que reiniciar Eclipse.

En caso de no seleccionar la opción anteriormente indicada, sí podremos seleccionar el arquetipo que nos interese en la siguiente ventana de selección. Después, pulsamos **Next**.

En la siguiente ventana de configuración, debemos especificar las coordenadas de Maven que se van a añadir al fichero **pom.xml** del nuevo proyecto:

- El paquete base para el nuevo proyecto, por defecto, recibe automáticamente el valor de **groupId**.
- El nombre del nuevo proyecto, **artifactId**.
- La versión inicial del nuevo proyecto, **version** que, por defecto, recibe un valor automáticamente aunque puede ser modificado a nuestra conveniencia.

Después, pulsamos **Finish** y ya estaría.

Hay que hacer notar que el arquetipo por defecto no está actualizado, de forma que los complementos que tiene configurados por defecto no son compatibles con versiones actuales de Java. No obstante, se soluciona fácilmente: Sólo es necesario decirle a Maven, a través del POM, que utilice versiones más actuales de los complementos indicándoles, allí donde sea necesario, qué versión de Java queremos emplear. Así, por ejemplo, para compilar con una versión actualizada de Java, podemos incluir en el POM lo siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.
org/xsd/maven-4.0.0.xsd">

   <modelVersion>...</modelVersion>
   <groupId>...</groupId>
   <artifactId>...</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <name>...</name>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
        <java.version>17</java.version>
    </properties>

    <dependencies>
        ...
    </dependencies>

    <build>
        <finalName>...</finalName>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.8.1</version>
                    <configuration>
                        <source>${java.version}</source>
                        <target>${java.version}</target>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

Cuando se realiza algún cambio en el POM, es necesario pulsar el botón derecho del ratón sobre el nombre del proyecto en Eclipse y seleccionar:

```
Maven -> Update Project
```

De esta manera, la configuración de Eclipse se sincroniza con la configuración del POM.

En ocasiones, Eclipse se queda atascado usando Maven, mostrando errores persistentes. Si estamos seguros de que el proyecto se encuentra bien configurado, podemos seguir la siguiente secuencia de acciones para que Eclipse se actualice correctamente y deje de mostrar dichos errores:

- Pulsamos botón derecho del ratón sobre el proyecto y seleccionamos -> Refresh.
- En la barra de menú de Eclipse, seleccionamos Project -> Clean.
- Pulsamos botón derecho del ratón sobre el proyecto y seleccionamos Maven -> Update Project.
- Pulsamos botón derecho del ratón sobre el proyecto y seleccionamos Run As... -> Maven Clean.

- Pulsamos botón derecho del ratón sobre el proyecto y seleccionamos Run As... -> Maven Install.

En el peor de los casos, podemos borrar todo el repositorio local de Maven y volver a ejecutar la secuencia anterior de acciones. Los artefactos que necesita el proyecto volverán a descargarse del repositorio central.

También podemos importar en Eclipse un proyecto ya creado con Maven. Así, por ejemplo, podemos descargar los proyectos situados en la siguiente dirección de GitHub:

https://github.com/josemari/JavaCursoPracticoProjects

Después, los proyectos descargados se pueden importar uno por uno en Eclipse siguiendo la siguiente secuencia de movimientos:

```
File -> Import -> Maven -> Existing Maven Projects
```

y seleccionar el directorio donde se encuentre el proyecto descargado y descomprimido.

Dado que el proyecto *BookExamples* tiene como dependencia al proyecto *BookLibrary*, es mejor compilar primero el proyecto *BookLibrary*, por lo que primero nos situamos sobre el proyecto *BookLibrary* y, sobre el nombre del mismo, pulsamos el botón derecho y seleccionamos

```
Run As... -> Maven Clean
```

Y después

```
Run As... -> Maven Install
```

Después, hacemos lo mismo con *BookExamples*.

Finalmente, para ejecutar cualquiera de los ejemplos que vienen con el proyecto *BookExamples* y que explicaremos en los sucesivos capítulos, lo único que hay que hacer es situarse en el fichero que contiene la clase que posee el método **main()** (del que hablaremos en su momento), pulsar el botón derecho y seleccionar

```
Run As... -> Java Application
```

1.3 DISECCIÓN DE UN PROGRAMA SENCILLO EN JAVA

Un programa sencillo en Java sería el siguiente:

```
package org.jomaveger.bookexamples.chapter1;
public class FirstExample {
   public static void main(String[] args) {
      System.out.println(“Hello World!”);
   }
}
```

Este programa se limita a imprimir un mensaje en la ventana de la consola.

Java respeta las mayúsculas y las minúsculas por lo que, si se cometen errores en este sentido (como, por ejemplo, escribir **Main** en lugar de **main**), el programa no funcionará.

La palabra reservada **public** es un modificador de acceso. Estos modificadores de acceso controlan el nivel de acceso que tienen a este código otras partes del programa.

La palabra reservada **class** recuerda que toda entidad que existe en un programa en Java vive dentro de una clase.

Después de la palabra reservada **class**, está el nombre de la clase. Los nombres de las clases tienen que empezar por una letra, y después pueden contener cualquier combinación de letras y dígitos. La longitud del nombre no está limitada. No se pueden utilizar palabras reservadas de Java como nombre de una clase.

La convención estándar consiste en que los nombres de las clases son sustantivos que comienzan por una letra mayúscula. Si un nombre está formado por múltiples palabras, se escribe en mayúsculas la inicial de cada una de estas palabras.

Es necesario hacer que el nombre de fichero correspondiente al código fuente sea el nombre de la clase pública, añadiéndole la extensión *.java*. Por tanto, es preciso almacenar este código en un fichero llamado *FirstExample.java*. De nuevo, las mayúsculas y minúsculas son importantes. No obstante, normalmente es el IDE Eclipse el que se encarga automáticamente de esta gestión.

Para ejecutar un programa compilado, la máquina virtual de Java siempre empieza la ejecución en el código que se encuentra en el método **main()** de la clase indicada. Por tanto, para que el código se pueda ejecutar, es preciso tener un método **main()** en el código fuente declarado como en el ejemplo. Es posible, por supuesto,

añadir nuestros propios métodos a una clase e invocar esos métodos desde el método **main()**, como veremos más adelante.

Las llaves delimitan los bloques del programa. Un método es un bloque de programa, por lo que el código de los métodos tiene que empezar por una llave abierta y tiene que terminar con una llave cerrada. Para un método, estas llaves marcan el principio y el fin del cuerpo del método. Dentro del método, se encuentran las sentencias. Toda sentencia en Java debe acabar con un punto y coma.

El cuerpo del método **main()** contiene una sentencia que escribe una única línea de texto en la consola. En este caso, estamos empleando el objeto **System.out** e invocamos al método **println()**. La sintaxis habitual de Java

```
object.method(params)
```

es, en cierto modo, equivalente a llamar a una función.

En este caso, estamos llamando al método **println()** y le pasamos una cadena de caracteres como parámetro real. El método escribe en la consola la cadena que recibe como argumento. Después, pone fin a la línea de resultado de tal modo que cada nueva llamada a **println()** muestra su resultado en una línea nueva.

Las cadenas de caracteres se delimitan mediante comillas dobles.

Los métodos en Java pueden carecer de parámetros, o bien tener uno o más parámetros. Ahora bien, aun cuando un método no tenga parámetros, sigue siendo necesario emplear los paréntesis cuando se le invoca. Por ejemplo, una variante del método **println()** sin parámetros se limita a imprimir una línea en blanco, y se invoca mediante la siguiente llamada:

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

El objeto **System.out** también tiene un método **print()** que no añade el caracter de nueva línea al resultado. Por ejemplo:

```
System.out.print(“Hello”);
```

imprime **Hello** sin un salto de línea al final. El próximo resultado aparecerá inmediatamente después de la *o*.

1.4 COMENTARIOS

Los comentarios en el código fuente en Java, al igual que en el resto de lenguajes de programación, no aparecen en el programa ejecutable. Java posee tres tipos de comentarios:

- El comentario que empieza por // se extiende desde ese caracter hasta el final de la línea.
- Se pueden emplear los delimitadores de comentarios /* y */ para crear un comentario de varias líneas.
- Finalmente, hay un tercer tipo de comentario que permite generar automáticamente la documentación del código. Este comentario emplea un /** para empezar y un */ para terminar.

Java tiene una herramienta muy útil, llamada **javadoc**, que genera documentación en formato **HTML** a partir de los ficheros de código fuente. La utilidad **javadoc** extrae información relativa a los elementos siguientes:

- Paquetes
- Clases e interfaces públicas.
- Métodos públicos y protegidos.
- Atributos públicos y protegidos.

Por tanto, la utilidad **javadoc** genera automáticamente la documentación del código fuente a partir de los comentarios que empiezan por /** y terminan por */.

Estos comentarios de documentación contienen texto de formato libre seguido en ocasiones de marcadores. La primera frase del texto de formato libre debería ser una frase que aporte un resumen. Los marcadores más habituales son los siguientes:

- El marcador

```
@param descripción de variable
```

  añade una entrada a la sección de parámetros del método actual. La descripción puede abarcar múltiples líneas. Todos los marcadores de este tipo de un mismo método deben estar juntos.

- El marcador

```
@return descripción
```

añade una sección al método actual, cuya longitud puede ser de varias líneas, que describe el contenido de lo que retorna el método.

- El marcador

```
@throws descripción de clase
```

añade una nota indicativa de que el método actual puede lanzar una excepción.

- El marcador

```
@author nombre
```

es un comentario de documentación de clase que indica el autor de la misma.

- El marcador

```
@version texto
```

es también un comentario de documentación de clase que describe la versión actual de la misma.

- El marcador

```
@since texto
```

se puede utilizar con cualquier elemento susceptible de ser documentado, de modo que el texto puede ser cualquier descripción de la versión en que se introdujo esta característica.

- El marcador

```
@deprecated texto
```

añade un comentario que indica que esta clase, método o variable ya no debería utilizarse; ahora bien, el texto debería sugerir un sustituto.

1.5 TIPOS DE DATOS ENTEROS

Los tipos de datos enteros se utilizan para números que carecen de parte fraccionaria. Se permiten valores negativos. Java ofrece cuatro tipos de datos enteros:

- El tipo **byte**, cuya anchura es de 8 bits o bien 1 byte, y su rango inclusivo de valores va de -128 a 127.
- El tipo **short**, cuya anchura es de 16 bits o bien 2 bytes, y su rango inclusivo de valores va de -32.768 hasta 32.767.
- El tipo **int**, cuya anchura es de 32 bits o bien 4 bytes, y su rango inclusivo de valores va de –2.147.483.648 a 2.147.483.647.
- El tipo **long**, cuya anchura es de 64 bits o bien 8 bytes, y su rango inclusivo de valores va de –9.223.372.036.854.775.808 a 9.223.372.036.854.775.807.

Es interesante observar que Java carece de tipos sin signo. Asimismo, en Java los tamaños de todos los tipos numéricos son independientes de la plataforma.

1.6 TIPOS DE DATOS DE PUNTO FLOTANTE

Los tipos de datos de punto flotante denotan números con parte fraccionaria. Java ofrece dos tipos de punto flotante:

- El tipo **float**, cuya anchura es de 32 bits o bien 4 bytes, y su rango aproximado de valores va de 1,4e–045 hasta 3,4e+038.
- El tipo **double**, cuya anchura es de 64 bits o bien 8 bytes, y su rango aproximado de valores va de 4,9e–324 hasta 1,8e+308.

Hay tres valores especiales en punto flotante: El infinito positivo, el infinito negativo y *NaN* (no es un número, del inglés *Not a Number*). Estos tres valores sirven para denotar los desbordamientos y errores. Por ejemplo, el resultado de dividir un número positivo entre cero es el infinito positivo. Al calcular la división de cero entre cero o la raíz cuadrada de un número negativo se obtiene *NaN*.

Las constantes **Double.POSITIVE_INFINITY**, **Double.NEGATIVE_INFINITY** y **Double.NaN** (así como las constantes **Float** correspondientes) representan estos valores especiales. No obstante, no se puede utilizar la comprobación:

```
if (x == Double.NaN)
```

porque nunca es verdadero, en el caso de querer determinar si un determinado resultado no es un número. Esto es debido a que todos los valores que no son un número se consideran diferentes entre sí. Sin embargo, se puede emplear para la misma finalidad el siguiente método:

```
if (Double.isNaN(x))
```

1.7 EL TIPO DE DATOS DE LOS CARACTERES

En Java, el tipo de datos empleado para almacenar caracteres es **char** y se considera también un tipo numérico. Java utiliza Unicode para representar los caracteres, que es un conjunto de caracteres completamente internacional que requiere 16 bits. Por tanto, el tipo **char** en Java tiene una anchura de 16 bits o bien 2 bytes y, por tanto, su rango inclusivo de valores va de 0 a 65.536. El conjunto estándar de caracteres conocido como ASCII abarca el rango de valores que va de 0 a 127; a su vez, el conjunto de caracteres ISO-Latin-1 comprende el rango de valores que va de 0 a 255.

1.8 EL TIPO DE DATOS LÓGICO

El tipo de datos en Java que gobierna los valores lógicos es el tipo de datos **boolean**, que sólo tiene dos posibles valores literales, **false** y **true**. Se utiliza para evaluar condiciones lógicas. A diferencia de otros lenguajes de programación, no se pueden hacer conversiones entre valores enteros y el tipo de datos **boolean**.

1.9 LITERALES

Un valor constante en Java se crea utilizando una representación literal del mismo.

Cualquier número entero escrito en base decimal, como *1*, *2*, *3* y *42*, se considera de tipo **int**. Un número entero puede expresarse en hexadecimal o base 16, mediante el prefijo **0x** o bien **0X** (por ejemplo, 0xCAFE). Un número entero también puede expresarse en octal o base 8, mediante el prefijo **0**; así, por ejemplo, *010* en octal es el *8* en decimal.

Para especificar un literal entero largo de tipo **long**, es necesario anexar al literal el sufijo **L** o **l** (por ejemplo, *4000L*).

A partir de Java 7, se pueden especificar literales enteros utilizando notación binaria, para lo cual se utiliza el prefijo **0b** o bien **0B**. Por ejemplo, el valor decimal *10* puede especificarse mediante el siguiente literal entero en forma binaria:

```
0b1010
```

También, a partir de Java 7, es posible utilizar uno o más guiones bajos en un literal entero con el fin de separar dígitos, de modo que los guiones bajos no pueden aparecer ni al principio ni al final del literal. Cuando el literal es compilado, los guiones bajos son descartados. Así, si tenemos el siguiente literal:

```
123_456_789
```

quedará el siguiente valor una vez haya sido compilado:

```
123456789
```

Los literales de punto flotante representan números decimales con una componente fraccionaria. Se pueden expresar en notación estándar o científica.

La notación estándar consiste en escribir el número como la parte entera seguida del punto decimal seguido de la parte fraccionaria; así, por ejemplo, *2.0*, *3.14159*, y *0.6667* son literales de punto flotante válidos escritos en notación estándar.

La notación científica utiliza un número en notación estándar más un sufijo que especifica la potencia de 10 por la que el número anterior va a ser multiplicado. El exponente se indica mediante una **E** o bien una **e** seguida de un número decimal que puede ser positivo o negativo; así, por ejemplo, *6.022E23*, *314159E–05*, y *2e+100* son literales de punto flotante válidos escritos en notación científica.

Los literales de punto flotante en Java por defecto son de tipo **double**. Para especificar un literal de tipo **float**, es necesario anexar el sufijo **F** o **f** a la constante.

Al igual que ocurría con los literales enteros, es posible utilizar uno o más guiones bajos en un literal de punto flotante con el fin de separar dígitos.

Los caracteres en Java son índices en la tabla de caracteres de Unicode. Un literal caracter se representa dentro de un par de comillas simples, como *'a'*, *'z'* y *'@'*. Para caracteres que no tienen una representación directa, existen varias

secuencias de escape que permiten introducir el caracter que se necesita como, por ejemplo:

- La secuencia de escape `\'` para el caracter de comilla simple.
- La secuencia de escape `\"` para el caracter de comillas dobles.
- La secuencia de escape \n para el caracter de salto de línea.
- La secuencia de escape \t para el caracter tabulador.
- La secuencia de escape \r para el caracter retorno de carro.
- La secuencia de escape \b para el caracter retroceso.
- La secuencia de escape `\\` para el caracter barra atrás.

1.10 VARIABLES

La variable es la unidad básica de almacenamiento de un programa Java. En Java, todas las variables tienen asociado un tipo de datos. Las variables se declaran poniendo primero el tipo, que va seguido por el nombre de la variable. Por ejemplo:

```
double sueldo;
int diasDeVacaciones;
long poblacionTerrestre;
boolean done;
```

Nótese el punto y coma que hay al final de cada declaración, necesario porque una declaración de variable es una sentencia completa en Java.

Los nombres de variable tienen que empezar por una letra y deben ser una sucesión formada por letras y dígitos. La longitud del nombre de una variable es, en esencia, ilimitada.

La convención estándar consiste en que los nombres de las variables son sustantivos que comienzan por una letra minúscula. Si un nombre está formado por múltiples palabras, se escribe en mayúsculas la inicial de cada una de estas palabras.

Tampoco se pueden utilizar las palabras reservadas de Java como nombres de variable.

Se pueden tener varias declaraciones en la misma línea:

```
int i, j;
```

Tras declarar una variable, es preciso darle valor inicial explícitamente mediante una sentencia de asignación; nunca se pueden utilizar los valores de variables que no hayan recibido un valor inicial.

Para asignar un valor a una variable declarada previamente, se pone el nombre de la variable a la izquierda, después un signo igual (que se conoce como el operador de asignación) y, a continuación, alguna expresión de Java a su derecha que tenga el valor del tipo adecuado. Por ejemplo:

```
int diasDeVacaciones;
diasDeVacaciones = 23;
```

Se puede declarar y dar valor inicial a una variable en la misma línea:

```
int diasDeVacaciones = 23;
```

Se pueden escribir declaraciones de variables en cualquier lugar del código.

Se puede dar valor inicial a una variable de forma dinámica; es decir, en lugar de utilizar un valor constante expresado mediante un literal, se puede iniciar una variable mediante una expresión válida cuyo valor se calculará en tiempo de ejecución. Así, por ejemplo sea el siguiente programa que calcula la longitud de la hipotenusa de un triángulo rectángulo dada la longitud de sus dos catetos:

```
package org.jomaveger.bookexamples.chapter1;
public class SecondExample {

   public static void main(String args[]) {
      double a = 3.0, b = 4.0;
      double c = Math.sqrt(a * a + b * b);
      System.out.println(“Hypotenuse is “ + c);
   }
}
```

Hemos visto en el ejemplo anterior que las variables han sido declaradas al comienzo del método **main()**. Ahora bien, como dijimos antes, las llaves delimitan los bloques del programa. Así, Java permite que se declaren variables dentro de

cualquier bloque de código. Un bloque de código define un ámbito. Por tanto, cada vez que se empieza un nuevo bloque abriendo una llave, se está creando un nuevo ámbito. Un ámbito determina qué objetos son visibles a otras partes del programa y también determina el tiempo de vida de dichos objetos.

En Java, existen principalmente dos ámbitos: El ámbito de clase y el ámbito de método. Las clases serán estudiadas con detenimiento en el próximo capítulo. Vamos a comentar ahora el ámbito definido por un método, que comienza con su llave abierta. No obstante, si el método tiene parámetros, están incluidos también en el ámbito del método.

Como regla general, las variables declaradas dentro de un ámbito no son visibles (y, por tanto, no son accesibles) al código definido fuera de ese ámbito. Ahora bien, los ámbitos pueden estar anidados. Cuando el anidamiento ocurre, el ámbito externo engloba al ámbito interno, lo que significa que los objetos declarados en el ámbito externo serán visibles al código situado en el ámbito interno. No obstante, lo contrario no es cierto, por lo que los objetos declarados dentro del ámbito interno no serán visibles fuera del mismo.

Sea el siguiente ejemplo:

```
package org.jomaveger.bookexamples.chapter1;
public class Scope {

   public static void main(String args[]) {
      int x;
      x = 10;
      if (x == 10) {
         int y = 20;

         System.out.println("x and y: " + x + " " + y);
         x = y * 2;
      }
      // y = 100; // Error!

      System.out.println("x is " + x);
   }
}
```

La variable *x* está declarada al comienzo del ámbito del método **main()** y es accesible a todo el código existente dentro de **main()**. Dentro del bloque *if*, la variable *y* es declarada. Dado que un bloque define un ámbito, *y* sólo es visible al resto del código situado en ese bloque. Ésta es la razón por la que, fuera de este bloque, la línea

```
y = 100;
```

está comentada. Si se descomenta la línea, tendrá lugar un error en tiempo de compilación, dado que *y* no es visible fuera de su bloque. Dentro del bloque *if*, *x* se puede utilizar porque el código que existe dentro de un bloque (es decir, el código que pertenece a un ámbito anidado) tiene acceso a las variables declaradas por el ámbito que lo engloba.

Un dato importante a tener en cuenta es que, aunque los bloques se pueden anidar, no es posible declarar una variable con el mismo nombre que otra situada en un ámbito externo.

Dentro de un bloque, las variables pueden ser declaradas en cualquier lugar, pero sólo son válidas después de haber sido declaradas.

Es importante recordar que las variables son creadas cuando se entra en su ámbito, y son destruidas cuando se abandona dicho ámbito. Esto significa que una variable no continuará almacenando su valor una vez hayamos abandonado su ámbito. Por tanto, las variables declaradas dentro de un método no conservarán sus valores entre distintas invocaciones a ese método. De forma semejante, una variable declarada dentro de un bloque perderá su valor cuando la ejecución del programa abandone ese bloque. Por tanto, el tiempo de vida de una variable está determinado por su ámbito. Si la declaración de una variable incluye un iniciador, entonces esa variable será reiniciada cada vez que la ejecución del programa entre en el bloque en el que dicha variable está declarada.

1.11 CONVERSIONES DE TIPO

Es muy común asignar un valor de un tipo de datos a una variable de otro tipo. Si los dos tipos son compatibles, entonces Java llevará a cabo la conversión automáticamente. Por ejemplo, siempre es posible asignar un valor entero de tipo **int** a una variable de tipo **long**. No obstante, no todos los tipos son compatibles y, por tanto, no todas las conversiones de tipo se permiten de manera implícita; así, por ejemplo, no existe una conversión automática definida del tipo **double** al tipo de datos **byte**. Afortunadamente, aún es posible realizar una conversión entre tipos incompatibles; para ello, es necesario emplear un moldeado de tipo, que realiza una conversión explícita entre tipos de datos incompatibles.

En resumidas cuentas, cuando un valor de un tipo de datos es asignado a una variable de otro tipo, una conversión de tipo automática tendrá lugar si se cumplen las dos condiciones siguientes:

- Los dos tipos son compatibles.
- El tipo de destino es más grande que el tipo origen.

Cuando se cumplen las dos condiciones, tiene lugar una conversión *creciente*. Por ejemplo, el tipo **int** es siempre suficientemente grande para almacenar todos los valores del tipo **byte**, por lo que no es necesario emplear una sentencia explícita de moldeado de tipo.

Para conversiones crecientes, los tipos numéricos, que incluyen los tipos de datos enteros y de punto flotante, son compatibles entre sí. No obstante, no existe conversión automática alguna de los tipos numéricos a **char** o **boolean**; además, los tipos de datos **char** y **boolean** no son compatibles entre sí.

Java también realiza una conversión de tipo automática cuando almacena una constante literal entera, de tipo **int**, en una variable de tipo **byte**, **short**, **long** o **char**.

Si, por el contrario, queremos asignar un valor de tipo **int** a una variable de tipo **byte**, entonces esta conversión no será realizada automáticamente porque el tipo de datos **byte** es más pequeño que el tipo de datos **int**. Se trata de una conversión *decreciente*, debido a que estamos explícitamente haciendo que el valor decrezca para que encaje en el tipo de destino.

Para crear una conversión entre dos tipos incompatibles, es preciso usar un moldeado de tipo, que no es más que una conversión explícita de tipo. La forma general de un moldeado de tipo es la siguiente:

```
(targetType) value
```

donde *targetType* especifica el tipo de datos deseado al que queremos convertir el valor especificado *value*. Por ejemplo, el siguiente fragmento de código moldea un entero **int** a un **byte**. Si el valor entero es mayor que el rango de valores admitido por el tipo **byte**, entonces se reducirá dicho valor almacenando en su lugar el resto de la división entera (también llamado módulo) del valor entre el rango del tipo **byte**:

```
int a;
byte b;
// …
b = (byte) a;
```

donde el rango del tipo **byte** es *256*.

Un tipo diferente de conversión tendrá lugar cuando un valor en punto flotante es asignado a una variable de tipo entero: El truncamiento. Dado que los números enteros no tienen parte fraccionaria, cuando un valor en punto flotante es

asignado a una variable de tipo entero, la parte fraccionaria se pierde. Por ejemplo, si el valor *1.23* se asigna a una variable de tipo **int**, el valor que realmente se almacenará en la variable entera será *1*. El *0.23* quedará truncado. Por supuesto, si el tamaño del número completo es demasiado grande para caber en el tipo entero de destino, entonces el valor final será reducido calculando el módulo del valor con el rango del tipo de datos destino.

Por ejemplo, sea el siguiente fragmento de código:

```
byte b;
double d = 323.142;
b = (byte) d;
```

Cuando **d** es convertido a un **byte**, la parte fraccionaria se trunca, y el valor se reduce módulo *256*, lo que en este caso produce un valor final de *67*.

Además de en las asignaciones, es muy común que ocurran conversiones de tipo en expresiones. En una expresión, la precisión requerida para un valor intermedio puede a veces exceder el rango de los operandos involucrados. Por ejemplo, sea la siguiente expresión:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

El resultado del término intermedio *a * b* excede el rango de sus operandos, que son de tipo **byte**. Para manejar este tipo de situaciones, Java automáticamente promociona cada operando de tipo **byte** o **short** al tipo **int** cuando evalúa una expresión. Esta comodidad tiene una desventaja. Sea, por ejemplo, el siguiente fragmento de código:

```
byte b = 50;
b = b * 2; // ERROR!
```

Se está intentando asignar *50 * 2* en una variable de tipo **byte**, lo que en principio parece admisible. Ahora bien, como los operandos fueron automáticamente promocionados al tipo **int** cuando la expresión fue evaluada, el resultado también ha sido promocionado al tipo **int**. Por tanto, el resultado de la expresión es ahora de tipo **int** y no puede asignarse a una variable de tipo **byte** sin emplear un moldeado de tipo. Y esto es cierto incluso aunque, como en este caso particular, el valor del resultado

se encuentre dentro del rango del tipo de datos destino. En este caso, se podría hacer lo siguiente:

```
byte b = 50;
b = (byte)(b * 2);
```

lo que conduce al valor correcto de *100*.

Java define varias reglas de promoción de tipo para aplicar en expresiones, de la forma siguiente: En primer lugar, todos los valores de tipo **byte**, **short** y **char** son promocionados al tipo **int**. Después, si un operando es de tipo **long**, toda la expresión es promocionada a **long**. Si un operando es de tipo **float**, la expresión completa es promocionada a **float**. Finalmente, si cualquiera de los operandos es de tipo **double**, el resultado será de tipo **double**.

Sea el siguiente fragmento de código:

```
byte b = 42;
byte c = 12;
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
```

En la primera subexpresión, **f * b**, **b** es promocionado al tipo **float** y el resultado de la subexpresión es de tipo **float**. Después, en la subexpresión **i / c**, **c** es promocionado al tipo **int**, de modo que el resultado es de tipo **int**. Luego, en la subexpresión **d * s**, el valor de **s** es promocionado al tipo **double** y el resultado de la subexpresión es de tipo **double**.

Finalmente, estos tres valores intermedios son considerados. El resultado del valor de tipo **float** más el valor de tipo **int** hace que éste último sea promocionado al tipo **float** y dé lugar a un resultado de tipo **float**. Posteriormente, este valor resultante de tipo **float** es promocionado al tipo **double** para hacer la diferencia menos el valor de tipo **double**, resultando el tipo de datos **double** el tipo del resultado final de la expresión global.

1.12 OPERADORES ARITMÉTICOS

Los operadores aritméticos habituales + - * / se emplean en Java para denotar suma, resta, multiplicación y división. El operador / denota la división entera

si ambos argumentos son enteros, y la división de punto flotante en caso contrario. El operador aritmético *%* denota el módulo o resto de la división entera. Así, por ejemplo, *15 / 2* es *7*, *15 % 2* es *1* y *15.0 / 2* es *7.5*.

La división entera entre cero da lugar a una excepción, mientras que la división de punto flotante entre cero produce un resultado infinito o *NaN*.

Existe una abreviatura cómoda para emplear operadores de aritmética binaria en la asignación. Por ejemplo:

```
x += 4;
```

es equivalente a

```
x = x + 4;
```

En general, se coloca el operador a la izquierda del signo igual, como en *= o en %=.

El operador de incremento incrementa el valor de su operando en uno. El operador de decremento decrementa el valor de su operando en uno. Por ejemplo, la siguiente sentencia:

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

puede ser reescrita de la siguiente forma utilizando el operador incremento:

```
x++;
```

De forma similar, la siguiente sentencia:

```
x = x - 1;
```

es equivalente a:

```
x--;
```

Estos dos operadores son únicos en el sentido de que pueden aparecer en forma sufija, cuando van después del operando, como es el caso mostrado anteriormente, y también pueden aparecer en forma prefija, cuando preceden al

operando. Ciertamente, ambos operadores modifican en uno el valor de la variable; ahora bien, la diferencia entre utilizar la forma sufija y la forma prefija aparece únicamente cuando se utilizan dentro de expresiones. En la forma prefija, el operando es incrementado o decrementado antes de que el valor sea obtenido para ser utilizado en la expresión. En la forma sufija, el valor previo es obtenido para ser usado en la expresión, y sólo entonces es modificado el operando.

Por ejemplo, sea el siguiente fragmento de código:

```
x = 42;
y = ++x;
```

En este caso, el valor de *y* es *43* como se podría esperar, dado que el incremento tiene lugar antes de que *x* se asigne a *y*. Por tanto, la línea

```
y = ++x;
```

es equivalente a estas dos sentencias

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

No obstante, si escribimos lo siguiente:

```
x = 42;
y = x++;
```

En este caso, la situación es bien distinta, porque el valor de *x* se obtiene antes de que el operador de incremento sea ejecutado, de modo que el valor de *y* es *42*. Por supuesto, en ambos casos el valor final de *x* se establece en *43*. Aquí, la línea

```
y = x++;
```

es equivalente a las siguientes dos sentencias

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

1.13 OPERADORES RELACIONALES Y LÓGICOS

Para comprobar la igualdad, se emplea un doble signo igual, ==. Para comprobar la desigualdad, se emplea el signo *!*=. Por último, se dispone de los operadores habituales < (menor), > (mayor), <= (menor o igual), y >= (mayor o igual).

Java utiliza *&&* para el operador lógico AND y emplea || para el operador lógico OR. Asimismo, utiliza *!* para el operador lógico NOT y ^ para el operador lógico XOR. Los operadores *&&* y || se evaluán en cortocircuito, de modo que el segundo argumento no se calcula si el primero ya ha determinado el resultado.

Así, por ejemplo, si se combinan dos expresiones mediante el operador *&&*:

```
expresion1 && expresion2
```

y se determina que el valor lógico de la primera expresión es **false**, entonces es imposible que el resultado sea **true**. Por tanto, no se calcula el valor de la segunda expresión.

Análogamente, el valor de la expresión:

```
expresion1 || expresion2
```

es automáticamente **true** si la primera expresión es **true**, sin evaluar la segunda expresión.

Por último, Java admite el operador ternario *?:*, que tiene la siguiente forma general:

```
condicion ? expresion1 : expresion2
```

Esta expresión produce como resultado el valor de *expresion1* si el valor de *condicion* es **true**, y el valor de *expresion2* en caso contrario. Por ejemplo, la siguiente expresión:

```
int x = 10;
int y = 7;
int z = x < y ? x : y
```

devuelve el menor de los valores *x* e *y*.

1.14 OPERADORES DE BITS

Cuando se trabaja con cualquiera de los tipos de datos enteros, se dispone de operadores que pueden operar directamente con los bits que forman los enteros. Los operadores de bits son los siguientes: *&* (AND), | (OR), ^ (XOR), ~ (NOT). Estos operadores actúan sobre tramos de bits. Por ejemplo, si *n* es una variable de tipo **int**:

```
int cuartoBitPorLaDerecha = (n & 8) / 8;
```

proporciona el valor *1* si el cuarto bit por la derecha de la representación en binario de *n* es un *1*, y *0* en caso contrario. El uso de *&* con la potencia de *2* adecuada nos permite enmascarar todos los bits menos uno.

Cuando se aplican a valores lógicos, los operadores *&* y | producen valores lógicos. Estos operadores son parecidos a los operadores *&&* y || , salvo que los operadores *&* y | no se evalúan en modo de cortocircuito; esto es, se evalúan los dos argumentos antes de calcular el resultado.

El operador de desplazamiento a la izquierda, <<, desplaza a la izquierda todos los bits de un valor un número especificado de veces. La forma general es la siguiente:

```
value << num
```

Por cada desplazamiento, el bit de orden superior es desplazado a la izquierda y se pierde, a la vez que un cero entra por la derecha.

Dado que cada desplazamiento a la izquierda tiene el efecto de doblar el valor original, es frecuente utilizar este hecho como una alternativa eficiente a la multiplicación por dos.

El operador de desplazamiento a la derecha, >>, desplaza a la derecha todos los bits de un valor un número especificado de veces. La forma general es la siguiente:

```
value >> num
```

Por cada desplazamiento, el bit de orden inferior es desplazado a la derecha y se pierde, a la vez que por la izquierda entra un bit igual al bit de orden superior. A este hecho se le denomina extensión de signo y se utiliza para preservar el signo de

los números negativos cuando se desplazan a la derecha. Por ejemplo, *–8 >> 1* es *–4* que, en binario, es

```
11111000  -8
>> 1
11111100  -4
```

Podemos observar que, cada vez que se desplaza un valor a la derecha, se divide ese valor entre dos, y se descarta el resto, por lo que tenemos una alternativa eficiente a la división entre dos.

En caso de no querer preservar el signo en un desplazamiento a la derecha, se puede emplear el operador de desplazamiento a la derecha sin signo, >>>, que desplaza a la derecha todos los bits de un valor un número especificado de veces de forma que rellena los bits de orden superior con ceros.

El siguiente fragmento de código demuestra el operador >>>. La variable *a* se establece al valor *-1*, lo que implica que sus *32* bits tienen valor *1* en binario. Este valor es desplazado a la derecha sin signo *24* bits, rellenando los *24* bits de orden superior con ceros, lo que conduce al valor de *255*:

```
int a = -1;
a = a >>> 24;
```

La misma operación en forma binaria sería la siguiente:

```
11111111 11111111 11111111 11111111    -1
>>>24
00000000 00000000 00000000 11111111    255
```

1.15 PRECEDENCIA DE OPERADORES Y PARÉNTESIS

La siguiente tabla muestra el orden de precedencia de los operadores de Java, del más al menos prioritario.

```
| Operadores                                                | Asociatividad
|  ------------------------------------------------------   | ---------------------- |
| [] . () (llamada a método)                                | De izquierda a derecha |
| ! ~ ++ -- +(unario) -(unario) () (moldeado) new           | De derecha a izquierda |
| * / %                                                     | De izquierda a derecha |
| + -                                                       | De izquierda a derecha |
| << >> >>>                                                 | De izquierda a derecha |
| < <= > >= instanceof                                      | De izquierda a derecha |
| == !=                                                     | De izquierda a derecha |
| &                                                         | De izquierda a derecha |
| ^                                                         | De izquierda a derecha |
| |                                                         | De izquierda a derecha |
| &&                                                        | De izquierda a derecha |
| ||                                                        | De izquierda a derecha |
| ?:                                                        | De derecha a izquierda |
| = += -= *= /= %= &= |= ^= <<= >>= >>>=                    | De derecha a izquierda |
```

Si no se emplean paréntesis, las operaciones se efectúan en el orden jerárquico que se indica. Los operadores del mismo nivel se procesan de izquierda a derecha, salvo aquellos que son asociativos por la derecha, según indica la tabla. Por ejemplo, como *&&* tiene una prioridad mayor que la de ||, la expresión

```
a && b || c
```

significa realmente

```
(a && b) || c
```

Como += se asocia de derecha a izquierda, la expresión

```
a += b += c
```

significa realmente

```
a += (b += c)
```

Esto es, el valor de *b* += *c* (que es el valor de *b* tras la suma) se le añade a *a*.

1.16 SENTENCIAS DE CONTROL: IF

La sentencia condicional de Java tiene la forma general siguiente:

```
if (condition) statement1;
else statement2;
```

Tanto *statement1* como *statement2* pueden ser una sentencia simple o una sentencia compuesta encerrada entre llaves (es decir, un bloque). La condición, que siempre tiene que ir encerrada entre paréntesis, es cualquier expresión que retorne un valor booleano. La cláusula **else** es opcional.

La sentencia **if** funciona de la manera siguiente: Si la condición es verdadera, entonces se ejecuta la sentencia *statement1*; en caso contrario, se ejecuta la sentencia *statement2* si existe. En ningún caso se ejecutarán ambas sentencias.

Por ejemplo, en el siguiente fragmento de código:

```
int a, b;
//...
if (a < b) a = 0;
else b = 0;
```

Si *a* es menor que *b*, entonces el valor de *a* se asigna a cero. En caso contrario, el valor de *b* se asigna a cero. Ahora bien, en ningún caso ambos son asignados a cero.

Un **if** anidado es una sentencia **if** que es el objetivo de otro **if** o **else**. Cuando se anida un **if**, el tema principal a recordar es que una sentencia **else** siempre se refiere a la sentencia **if** más cercana que está situada dentro del mismo bloque que el **else** y que no está ya asociada a un **else**. Por ejemplo, sea el siguiente fragmento de código:

```
if (i == 10) {
   if (j < 20) a = b;
   if (k > 100) c = d; // este if esta
   else a = c; // asociado con este else
}
else a = d; // este else se refiere al if(i == 10)
```

Como indican los comentarios, la sentencia **else** final no está asociada con la sentencia

if (j < 20)

dado que no está en el mismo bloque (aun cuando es la sentencia condicional más cercana sin un **else** asociado). En su lugar, el **else** final está asociado con

if (i == 10)

A su vez, el **else** interno se refiere al **if (k > 100)** porque es el más cercano dentro del mismo bloque.

Una construcción común en programación es la escalera **if-else-if**, que tiene la forma general siguiente:

```
if (condition1)
   statement1;
else if (condition2)
   statement2;
else if (condition3)
   statement3;
...
else
   statement;
```

La forma de ejecutarse de esta construcción no tiene mucho misterio. Las sentencias **if** se ejecutan de arriba abajo. Tan pronto como una de las condiciones que controlan uno de los **if** es verdadera, la sentencia asociada con ese **if** es ejecutada, y el resto de la escalera es ignorada. Si ninguna de las condiciones es verdadera, entonces la sentencia asociada al **else** final será ejecutada. El **else** final actúa como una condición por defecto; es decir, si fallan el resto de pruebas condicionales, entonces se ejecuta la última sentencia **else**. En caso de no existir **else** final y además que el resto de condiciones sean falsas, entonces no se ejecuta acción alguna.

1.17 SENTENCIAS DE CONTROL: SWITCH

La escalaera **if-else-if** puede llegar a ser engorrosa si se extiende con demasiadas alternativas. Java posee una sentencia de selección múltiple **switch** para tal fin. La forma general de la sentencia **switch** es la siguiente:

```
switch (expression) {
   case value1:
      // statement sequence
      break;
   case value2:
      // statement sequence
      break;
```

```
    ...
    case valueN :
       // statement sequence
       break;
    default:
       // default statement sequence
}
```

La expresión de una sentencia **switch** puede ser de tipo **byte**, **short**, **int**, **char**, **String** o bien de un tipo enumerado.

Cada valor especificado en una sentencia **case** debe ser una expresión constante única (como, por ejemplo, un valor literal); por tanto, valores duplicados no se permiten y, como es lógico, el tipo de cada valor debe ser compatible con el tipo de la expresión.

La sentencia **switch** funciona de la siguiente manera: El valor de la expresión es comparado con cada uno de los valores de las sentencias **case**. Si alguno coincide, la secuencia de código que sigue a ese **case** se ejecuta. Si ninguna de las constantes coincide con el valor de la expresión, entonces se ejecuta la sentencia **default**. No obstante, la sentencia **default** es opcional. Si no existe sentencia **default** y ningún **case** coincide con el valor de la expresión, entonces no se lleva a cabo ninguna acción.

La sentencia **break** se utiliza dentro de la sentencia **switch** para terminar una secuencia de sentencias. Cuando la ejecución se encuentra con una sentencia **break**, la ejecución se bifurca a la primera línea de código que continúa después de la sentencia **switch** completa, lo que produce el efecto de saltar fuera del **switch**.

Sea el siguiente ejemplo que utiliza una sentencia **switch**:

```
package org.jomaveger.bookexamples.chapter1;
public class SampleSwitch {

   public static void main(String args[]) {
      for (int i = 0; i < 6; i++)
         switch (i) {
         case 0:
            System.out.println("i is zero.");
            break;
         case 1:
            System.out.println("i is one.");
            break;
         case 2:
            System.out.println("i is two.");
            break;
         case 3:
            System.out.println("i is three.");
```

```
            break;
         default:
            System.out.println("i is greater than 3.");
         }
      }
   }
```

La salida producida por el programa sería la siguiente:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Como se puede ver, cada vez que se ejecuta una iteración del bucle, se ejecutan las sentencias asociadas con la constante del **case** que se corresponde con el valor de *i*. Todas las demás ramas de la sentencia **switch** son ignoradas. Una vez que *i* es mayor que 3, no hay sentencia **case** que se corresponda con el valor de *i*, por lo que la sentencia **default** se ejecuta.

La sentencia **break** es opcional. Si se omite el **break**, la ejecución continuará en el siguiente caso, es decir, en la siguiente sentencia **case**. No obstante, a veces la lógica de la aplicación hace deseable tener múltiples sentencias **case** una detrás de otra sin sentencias **break** entre ellas.

1.18 SENTENCIAS DE CONTROL: WHILE

Las sentencias de iteración de Java (más conocidas como bucles) son **while**, **for** y **do-while**. El bucle **while** ejecuta una sentencia (que puede ser una sentencia de bloque) mientras sea cierta una determinada condición. Su forma general es la siguiente:

```
while (condition) {
   statement;
}
```

El bucle **while** no llegará a ejecutarse si la condición *condition* es falsa desde el principio.

El bucle **while** es la sentencia de iteración más importante de Java. En esencia, repite una sentencia o bloque de sentencias mientras su condición de entrada es verdadera. La condición de entrada al bucle **while** puede ser cualquier expresión de tipo **boolean**. Cuando dicha condición *condition* se vuelve falsa, la ejecución continúa en la línea de código inmediatamente posterior al bucle. Las llaves son innecesarias si el bucle se limita a repetir una única sentencia.

La sentencia o sentencias que se ejecutan repetidamente por un bucle reciben el nombre de cuerpo del bucle.

He aquí un bucle muy sencillo que cuenta en sentido descendente desde diez:

```
package org.jomaveger.bookexamples.chapter1;
public class While {

   public static void main(String args[]) {
      int n = 10;
      while (n > 0) {
         System.out.println("tick " + n);
         n--;
      }
   }
}
```

La salida de este programa será la siguiente:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

1.19 SENTENCIAS DE CONTROL: DO-WHILE

Si la expresión condicional que controla el bucle **while** es inicialmente falsa, entonces el cuerpo del bucle no llegará a ejecutarse ni una sola vez. No obstante, en ocasiones es deseable ejecutar el cuerpo del bucle al menos una vez, incluso si la expresión condicional es falsa la primera vez que se evalúa. En otras palabras,

hay veces en las que nos gustaría comprobar la expresión de terminación al final del bucle en lugar de al principio. Afortunadamente, Java proporciona un bucle que hace justo eso: El bucle **do-while**, que siempre ejecuta su cuerpo al menos una vez, porque su expresión condicional de terminación se sitúa al final del bucle. Su forma general es la siguiente:

```
do {
   statement;
} while (condition);
```

Cada iteración del bucle **do-while** ejecuta en primer lugar el cuerpo del bucle y después evalúa la condición de terminación. Si la expresión es verdadera, el bucle se volverá a ejecutar; en caso contrario, el bucle finaliza. Como es habitual en los bucles de Java, la condición de terminación debe ser una expresión de tipo **boolean**.

Aquí tenemos una reescritura del programa anterior que muestra el bucle **do-while** y genera la misma salida que antes:

```
package org.jomaveger.bookexamples.chapter1;
public class DoWhile {
   public static void main(String args[]) {
      int n = 10;
      do {
         System.out.println("tick " + n);
         n--;
      } while (n > 0);
   }
}
```

1.20 SENTENCIAS DE CONTROL: FOR

Vamos a estudiar el bucle **for** tradicional. Otras versiones serán estudiadas en próximos capítulos. La forma general de un bucle **for** tradicional es la siguiente:

```
for (initialization; condition; iteration) {
   statement;
}
```

Si sólo se va a repetir una única sentencia, entonces las llaves no son necesarias.

La forma de trabajar del bucle **for** es la siguiente: Cuando el bucle comienza su ejecución, se ejecuta la sección de inicialización del bucle; normalmente, se trata de una expresión que establece el valor inicial de la variable de control del bucle, que actúa como un contador que controla el bucle. Es importante entender que la expresión de inicialización se ejecuta sólo una vez.

A continuación, se evalúa la condición, que debe ser una expresión de tipo **boolean**. Generalmente, se comprueba el valor de la variable de control del bucle respecto a un cierto valor objetivo. Si esta expresión es verdadera, entonces el cuerpo del bucle se ejecuta; en caso contrario, el bucle finaliza su ejecución.

Finalmente, se ejecuta la expresión de iteración del bucle, que usualmente se trata de una expresión que incrementa o decrementa la variable de control del bucle.

El bucle entonces itera, primero evaluando la condición, después ejecutando el cuerpo del bucle, a continuación ejecutando la expresión de iteración, y así este proceso se repite hasta que la condición de terminación es falsa.

Aquí tenemos una reescritura del programa anterior que muestra el bucle **for** y genera la misma salida que antes:

```
package org.jomaveger.bookexamples.chapter1;
public class For {
   public static void main(String args[]) {
      for (int n = 10; n > 0; n--)
         System.out.println("tick " + n);
   }
}
```

Cuando se declara una variable en la primera cláusula del bucle **for**, el ámbito de la variable llega hasta el final de dicho bucle **for**. En particular, si se define una variable en el cuerpo de un bucle **for**, no es posible utilizar el valor de esa variable fuera del bucle. Por tanto, si se desea utilizar el valor final del contador de un bucle fuera del bucle, hay que asegurarse de declararlo fuera del encabezado del bucle, como en el ejemplo siguiente:

```
package org.jomaveger.bookexamples.chapter1;
public class For2 {
   public static void main(String args[]) {
      int n;
      for (n = 10; n > 0; n--)
         System.out.println("tick " + n);

      System.out.println("Final tick " + n);
   }
}
```

La salida generada por este programa sería:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
Final tick 0
```

1.21 SENTENCIAS DE CONTROL: BREAK Y CONTINUE

Las sentencias **break** y **continue** se denominan sentencias de salto y, como norma general, es mejor no usarlas porque suelen denotar un estilo pobre de programación. No obstante, vamos a aprender lo mínimo imprescindible sobre ellas.

La sentencia **break** se puede utilizar para abandonar un bucle. Al utilizar **break** dentro de un bucle, se fuerza la inmediata terminación del bucle, evitando la evaluación de la condición e ignorando la ejecución de cualquier código pendiente de ejecutar en el cuerpo del bucle. Cuando la máquina virtual de Java encuentra una sentencia **break** dentro de un bucle, el bucle es finalizado y el programa continúa ejecutándose en la siguiente sentencia situada a continuación del bucle.

La sentencia **continue** transfiere el control al encabezado del bucle que la contenga. Es decir, se utiliza para forzar antes de tiempo una iteración del bucle, de modo que continuamos ejecutando el bucle pero detenemos la ejecución del resto del código del cuerpo del bucle para esta iteración particular. En un bucle **while** y **do-while**, la sentencia **continue** provoca que el control se transfiera directamente a la expresión condicional que controla el bucle. En un bucle **for**, la sentencia **continue** provoca que el control se transfiera primero a la expresión de iteración y después a la expresión condicional. Y, para los tres tipos de bucles, cualquier código restante situado en el cuerpo del bucle es ignorado.

1.22 FUNCIONES Y CONSTANTES MATEMÁTICAS

La clase **Math** de Java contiene un amplio conjunto de funciones matemáticas que quizá se necesiten ocasionalmente, dependiendo del tipo de aplicaciones que se desarrollen.

Para calcular la raíz cuadrada de un número, se utiliza el método **sqrt()**. Por ejemplo, sea el siguiente fragmento de código:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); //imprime 2.0
```

Hay una sutil diferencia entre el método **println()** y el método **sqrt()**. El método **println()** actúa sobre un objeto, **System.out**, que está definido en la clase **System**. Pero el método **sqrt()** de la clase **Math** no actúa sobre objeto alguno: Se trata de un método estático. En el próximo capítulo estudiaremos los métodos estáticos.

El lenguaje de programación Java no tiene un operador que sirva para elevar un número a una potencia; para realizar esta operación, es preciso utilizar el método **pow()** de la clase **Math**. Así, la sentencia:

```
double y = Math.pow(x, a);
```

hace que *y* tome el valor de *x* elevado a la potencia de *a*. El método **pow()** tiene dos parámetros, ambos de tipo **double**, y también retorna un **double**.

La clase **Math** ofrece las funciones trigonométricas habituales:

```
Math.sin();
Math.cos();
Math.tan();
Math.atan();
Math.atan2();
```

y también la función exponencial y su inversa, el logaritmo natural:

```
Math.exp();
Math.log();
```

La clase **Math** también tiene dos constantes que representan aproximaciones de las constantes matemáticas *pi* y *e*:

```
Math.PI;
Math.E;
```

Por último, la clase **Math** ofrece dos métodos **max()** y **min()** que reciben dos argumentos del mismo tipo y calculan, respectivamente, el mayor o el menor valor de dichos argumentos. Cada uno de los métodos tiene cuatro versiones del mismo en la clase **Math**, cambiando sólo el tipo de los dos argumentos, que pueden ser **float**, **double**, **int** y **long**.

1.23 NÚMEROS GRANDES

Los números de punto flotante no son adecuados para efectuar aritmética financiera, en la cual no se admiten errores de redondeo. Por ejemplo, la sentencia

```
System.out.println(2.0 - 1.1);
```

produce 0.89999999999, y no 0.9 como se podía esperar. Estos errores de redondeo están causados por el hecho de que los números de punto flotante se representan en sistema binario. La fracción 1/10 no tiene una representación finita en binario, del mismo modo que no existe una representación finita de 1/3 en el sistema decimal. Si se necesitan unos cálculos numéricos precisos sin errores de redondeo, hay que utilizar la clase **BigDecimal**.

En general, si no es suficiente la precisión de los tipos básicos enteros y de punto flotante, podemos recurrir a un par de clases que se encuentran en el paquete **java.math**: **BigInteger** y **BigDecimal**. Se trata de clases para manipular números que constan de una secuencia de cifras de longitud arbitraria. La clase **BigInteger** implementa la aritmética entera de precisión arbitraria, y la clase **BigDecimal** implementa la aritmética de punto flotante de precisión arbitraria.

El método estático **valueOf()** se utiliza para convertir un número ordinario en un número grande:

```
BigInteger a = BigInteger.valueOf(100);
```

Para operar con números grandes, hay que utilizar métodos como **add()** y **multiply()**, pertenecientes a las clases dedicadas a los números grandes:

```
BigInteger b = BigInteger.valueOf(20);
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```

Los métodos más importantes de la clase **BigInteger** son los siguientes:

- Los métodos **add()**, **substract()**, **multiply()**, **divide()** y **mod()** retornan, respectivamente, la suma, resta, producto, cociente y resto de este entero grande por otro pasado como argumento.
- El método **compareTo()** retorna, respectivamente, cero, un resultado negativo o un resultado positivo si este entero grande es igual, menor o bien mayor que otro que recibe como argumento.
- El método **valueOf()** devuelve un número entero grande cuyo valor es igual al del argumento de tipo **long** que recibe el método.

Los métodos más importantes de la clase **BigDecimal** son los siguientes:

- Los métodos **add()**, **substract()**, **multiply()** y **divide()** retornan, respectivamente, la suma, resta, producto y cociente de este número real grande por otro pasado como argumento. Para calcular el cociente, es preciso proporcionar un modo de redondeo. El modo **RoundingMode.HALF_UP** es el método de redondeo que aprendimos en la escuela; esto es, redondear hacia abajo las cifras del 0 al 4, y hacia arriba las cifras del 5 al 9.
- El método **compareTo()** retorna, respectivamente, cero, un resultado negativo o un resultado positivo si este número real grande es igual, menor o bien mayor que otro que recibe como argumento.
- El método **valueOf()** devuelve un número real grande cuyo valor es igual al del argumento de tipo **double** que recibe el método.

1.24 CADENAS DE CARACTERES

Conceptualmente, las cadenas de caracteres en Java son secuencias de caracteres en Unicode. Java no posee un tipo predefinido para las cadenas de caracteres; en su lugar, la biblioteca estándar de Java provee de una clase predefinida que se denomina, lógicamente, **String**. Toda cadena que vaya entre comillas dobles es un ejemplar de la clase **String**:

```
String e = “”; // una cadena vacía
String greeting = “Hello”;
```

El método **length()** devuelve la longitud de la cadena de caracteres:

```
int l = greeting.length(); // es 5
```

El método **charAt()** proporciona el caracter que se encuentra en la posición de la cadena indicada por el argumento del método, y tal que los valores válidos de esta posición están entre 0 y **length() - 1**, ambos inclusive:

```
char first = greeting.charAt(0);  // la primera es 'H'
char last = greeting.charAt(4);  // la última es 'o'
```

Para extraer una subcadena de una cadena más larga, se emplea el método **substring()** de la clase **String**. Por ejemplo, el siguiente fragmento de código:

```
String s = greeting.substring(0, 3);
```

crea una cadena formada por los caracteres "Hel".

El segundo parámetro de **substring()** es el primer caracter que no se quiere copiar. En nuestro caso, deseamos copiar los caracteres que se encuentran en las posiciones 0, 1 y 2 (de la posición 0 a la posición 2, ambas inclusive). Tal y como se cuentan en el método **substring()**, se trata de ir desde la posición 0 inclusive hasta la posición 3 exclusive.

La clase **String** no posee métodos que sirvan para modificar un caracter de una cadena ya existente. Como no es posible modificar los caracteres individuales de una cadena de Java, se dice que los objetos de la clase **String** son inmutables. Del mismo modo que el número 3 siempre es 3, la cadena de caracteres "Hello" siempre contendrá los mismos caracteres. Estos valores no se pueden cambiar. Lo que sí se puede hacer es modificar el contenido de la variable de tipo **String** llamada *greeting*, y hacer que pase a referirse a otra cadena de caracteres diferente, del mismo modo que se puede hacer que una variable numérica que contenga el valor 3 pase a tener el valor 4. Así, si se desea cambiar el valor de *greeting* para que pase a ser "Hell-Oh!", habría que concatenar la subcadena que se quiere mantener con los caracteres que se quieran sustituir:

```
greeting = greeting.substring(0, 4) + "-Oh!";
```

Como vemos, Java permite emplear el signo + para unir (concatenar) dos cadenas. Es más, cuando se concatena una cadena con un valor que no es una cadena, este último se transforma en una cadena.

Para comprobar si dos cadenas son iguales, se utiliza el método **equals()**. La expresión

```
s.equals(t)
```

devuelve **true** si las cadenas **s** y **t** son iguales, y **false** en caso contrario. Tanto **s** como **t** pueden ser variables de cadena como cadenas constantes. Por ejemplo, la expresión

```
"Hello".equals(greeting)
```

es perfectamente válida. Para determinar si dos cadenas son idénticas salvo por la distinción entre mayúsculas y minúsculas, se emplea el método **equalsIgnoreCase()**:

```
"Hello".equalsIgnoreCase("hello");
```

En caso de necesitar manipular directamente las cadenas, Java proporciona dos clases para ello: **StringBuffer** y **StringBuilder**. La diferencia entre ambas radica en que, mientras **StringBuffer** está sincronizada y proporciona acceso concurrente seguro por parte de múltiples hilos de ejecución a una cadena de caracteres mutable, la clase **StringBuilder** no lo está. No obstante, ésta última manifiesta mejor rendimiento debido a dicha ausencia de mecanismos de sincronización.

Los métodos de las clases **StringBuffer** y **StringBuilder** son idénticos. Los más importantes son:

- Métodos constructores que nos permiten crear ejemplares de las respectivas clases. Los más importantes son los constructores vacíos y los que reciben una cadena de caracteres **String** como argumento.

```
StringBuffer sb1 = new StringBuffer();  // no hay caracteres
StringBuffer sb2 = new StringBuffer("Hello");
```

- El método **toString()** que permite obtener una cadena de caracteres **String** con los mismos datos.

```
String s = sb2.toString();
```

- El método **append()** que añade una cadena de caracteres al objeto actual y, a su vez, devuelve el objeto actual modificado.

```
sb2 = sb2.append(“ world”).append(“ Java”);
```

1.25 ENTRADA Y SALIDA

Ya se ha visto que resulta sencillo imprimir el resultado en la ventana de la consola con sólo llamar a **System.out.println**. Para leer la entrada de la consola, lo primero que se hace es construir un objeto de la clase **Scanner** que esté asociado al flujo de entrada estándar, **System.in**:

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

Ahora ya se pueden utilizar los distintos métodos de la clase **Scanner** para leer la entrada. Por ejemplo, el método **nextLine()** lee una línea de entrada:

```
System.out.println(“¿Cuál es tu nombre?”);
String nombre = in.nextLine();
```

Se utiliza el método **nextLine()** porque la entrada podría contener espacios en blanco de separación. Para leer una sola palabra, que lógicamente puede estar separada de la siguiente por espacios de separación, se utiliza la llamada:

```
String nombreDePila = in.next();
```

Para leer un número entero, se emplea el método **nextInt()**:

```
System.out.println(“¿Cuál es tu edad?”);
int edad = in.nextInt();
```

Análogamente, el método **nextDouble()** lee el próximo número de punto flotante.

Por último, para utilizar la clase **Scanner** es preciso añadir la línea

```
import java.util.Scanner;
```

al principio del programa. La clase **Scanner** está definida en el paquete **java.util**. Siempre que se utiliza una clase que no está definida en el paquete básico **java.lang**,

es preciso utilizar una sentencia **import**. En el próximo capítulo estudiaremos más en profundidad sobre paquetes e importación de los mismos.

Sea el siguiente fragmento de código:

```
double x = 10000.0 / 3.0;
System.out.println(x);
```

Como ya sabemos, así podemos escribir el número *x* en la consola; ahora bien, el número *x* se imprimirá con el número máximo de dígitos distintos de cero correspondientes a su tipo. En este caso, lo que aparece por la consola es lo siguiente:

```
3333.3333333333335
```

No obstante, existe el método **printf()** que nos permite dar formato a la salida. Por ejemplo, la llamada

```
System.out.printf("%8.2f", x);
```

imprime *x* con una anchura de campo de 8 caracteres y una precisión de 2 cifras. Esto es, el resultado contiene un espacio en blanco precedente y los siguientes siete caracteres:

```
 3333,33
```

Al método **printf()** se le pueden pasar múltiples parámetros, por ejemplo:

```
System.out.printf("Hola, %s. Tienes %d años.", nombre, edad);
```

Todos los especificadores de formato que comienzan por el caracter *%* se reemplazan por el argumento correspondiente. El caracter de conversión que hay al final del especificador de formato indica el tipo del valor al que se desea dar formato: *f* es un número de punto flotante, *s* es una cadena de caracteres y *d* es un número entero decimal.

Además, se pueden especificar modificadores que controlan el aspecto del resultado impreso. Por ejemplo, el modificador coma añade separadores de grupo:

```
System.out.printf("%,.2f", 10000.0 / 3.0);
```

lo que nos daría como resultado

```
3.333,33
```