Java 17

Fundamentos prácticos de programación

Desde **www.ra-ma.es** podrá descargar material adicional.

José María Vegas Gertrudix

Ra-Ma®

edU

Java 17

Fundamentos prácticos de programación

José María Vegas Gertrudix

Ra-Ma®
edU
Conocimiento a su alcance
Bogotá - México, D.F.

Vegas Gertrudix, José María

Java 17, fundamentos prácticos de programación / José María Vegas Gertrudix --.
Bogotá: Ediciones de la U, 2022
412 p. ; 24 cm
ISBN 978-958-792-410-7 e-ISBN 978-958-792-411-4
1. Informática 2. Programación 3. Estructuras de datos 4. Java 17 4. I. Tít.
621.39 ed.

Edición original publicada por © Editorial Ra-ma (España)
Edición autorizada a Ediciones de la U para Colombia

Área: Informática
Primera edición: Bogotá, Colombia, septiembre de 2022
ISBN. 978-958-792-410-7

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.
Carátula: Ediciones de la U
Impresión: DGP Editores SAS
Calle 63 #70D-34, Pbx (57+1) 3203510

Impreso y hecho en Colombia
Printed and made in Colombia

A mi madre, María,
por su paciencia, amor, apoyo
y consejo durante toda mi vida
y especialmente en los momentos difíciles.

A mi padre, José María, y a mi abuela, Agustina,
por transmitir su experiencia en medio de la vorágine.

A mi hermanito, Shadi, por su incansable sentido del humor
y su inquebrantable amistad.

A NSJC y su Santa Madre,
para quienes todo es posible.

ÍNDICE

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*.