DESARROLLO DE PROYECTOS INFORMÁTICOS CON TECNOLOGÍA JAVA

Óscar Belmonte Fernández
Carlos Granell Canut
María del Carmen Erdozain Navarro



UNIVERSITAT JAUME·I



Esta obra de Óscar Belmonte et al. está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported

Índice general

1. Introducción **13**
- 1.1. Origen del lenguaje de programación Java . . . 13
- 1.2. Característica de Java . . . 14
- 1.3. El entorno de desarrollo integrado Eclipse . . . 15
  - 1.3.1. Principales características del entorno de desarrollo Eclipse 16
  - 1.3.2. Descarga e instalación de Eclipse . . . 16
  - 1.3.3. Configurando el aspecto de Eclipse: Perspectivas y Vistas 16
  - 1.3.4. El primer ejemplo . . . 18
- 1.4. Herramientas de desarrollo . . . 21
  - 1.4.1. Añadiendo nueva funcionalidad a Eclipse: los *plug-ins* . . 22

2. Clases **23**
- 2.1. Definición de una clase . . . 24
- 2.2. Miembros de una clase . . . 25
  - 2.2.1. Atributos de una clase . . . 25
  - 2.2.2. Métodos de una clase. . . . 26
  - 2.2.3. Constructores. . . . 28
  - 2.2.4. Sobrecarga de métodos y constructores . . . 32
- 2.3. Tipos de datos en Java. . . . 33
  - 2.3.1. Arrays de datos en Java. . . . 34
- 2.4. Estructuras de control. . . . 36
  - 2.4.1. Estructuras de control de repetición. . . . 37
  - 2.4.2. Estructuras de control de selección. . . . 39
- 2.5. Modificadores de acceso. . . . 40
- 2.6. Modificadores `static` y `final`. . . . 42
- 2.7. El recolector de basura. . . . 43
- 2.8. Finalización. . . . 44
- 2.9. Comentarios. Comentarios de documentación. . . . 45

3. Herencia e Interfaces **51**
- 3.1. Herencia. . . . 52
- 3.2. Extensión de una clase. . . . 52
  - 3.2.1. Sobrescribir atributos. . . . 54
  - 3.2.2. Sobrescribir métodos. . . . 56
  - 3.2.3. La palabra reservada `super`. . . . 59
  - 3.2.4. El constructor por defecto y la clase `Object`. . . . 59
  - 3.2.5. El operador `instanceof`. . . . 60
  - 3.2.6. El modificador `final`. . . . 61

3.2.7. Métodos `static` . . . 62
3.3. Clases abstractas . . . 63
3.4. Interfaces . . . 65
3.5. Enumeraciones . . . 68
3.6. Paquetes en Java . . . 69
3.7. Clases e `interface` anidados . . . 71

4. Subversion **75**
4.1. ¿Qué es un sistema de control de versiones? . . . 76
4.2. Principales características de *Subversion* . . . 76
4.3. Creación de un repositorio . . . 77
4.4. Trabajo con repositorios . . . 78
4.4.1. Obteniendo información del repositorio . . . 82
4.5. Integración con Eclipse . . . 84

5. Excepciones **87**
5.1. ¿Qué es una excepción? . . . 87
5.1.1. Tipos de excepciones . . . 88
5.2. Cómo se gestiona una excepción . . . 88
5.3. Creación de excepciones propias . . . 91

6. Pruebas unitarias con JUnit **93**
6.1. ¿Qué son las pruebas unitarias? . . . 94
6.1.1. Principios FIRST para el diseño de pruebas unitarias . . . 94
6.2. Pruebas unitarias con JUnit . . . 95
6.2.1. Creación de clases de prueba . . . 95
6.2.2. La anotación `@Test` . . . 96
6.2.3. Las anotaciones `@Before` y `@After` . . . 98
6.2.4. Las anotaciones `@BeforeClass` y `@AfterClass` . . . 99
6.2.5. Pruebas con batería de datos de entrada . . . 100
6.2.6. Ejecutar varias clases de prueba. Test Suites . . . 101
6.3. Cobertura de las pruebas . . . 102
6.3.1. EclEmma y su plug-in para Eclipse . . . 103

7. Entrada y Salida **105**
7.1. Flujos (*Streams*) . . . 106
7.2. Flujos de bytes . . . 107
7.3. Flujos de caracteres . . . 108
7.4. Conexión entre flujos de bytes y de caracteres . . . 109
7.5. El sistema de ficheros y flujos a ficheros . . . 110
7.5.1. El sistema de ficheros . . . 110
7.5.2. Flujos a ficheros . . . 110
7.6. Serialización . . . 112

8. Algunas clases de utilidad del paquete estándar **117**
8.1. La clase `Scanner` . . . 118
8.2. Trabajo con cadenas de caracteres . . . 120
8.2.1. La clase `String` . . . 120
8.2.2. Las clases `StringBuffer` y `StringBuilder` . . . 121
8.3. Clases recubridoras . . . 122

8.4. Colecciones 124
8.5. Trabajo con fechas 128
8.5.1. La clase `Date` 128
8.5.2. Las clases `Calendar` y `GregorianCalendar` 129
8.6. Matemáticas 129
8.6.1. La clase `Math` 129
8.6.2. La clase `Random` 130

9. Programación con genéricos **133**
9.1. ¿Qué son los tipos de datos genéricos? 133
9.2. Métodos genéricos 134
9.3. Clases genéricas 135
9.4. Ampliación del tipo genérico 138
9.4.1. Tipos genéricos con límite superior 139
9.4.2. Comodines 139
9.5. Borrado de tipo y compatibilidad con código heredado 141

10. Construcción de proyectos con *Ant* **143**
10.1. Qué es *Ant* 144
10.2. Definición del proyecto 144
10.2.1. Objetivos 145
10.2.2. Tareas 145
10.3. Compilar el código fuente de un proyecto 146
10.4. Propiedades 146
10.5. Estructuras `path-like` 147
10.6. Ejecución de las Pruebas Unitarias 148
10.7. Generación de la documentación 150
10.8. Empaquetado de la aplicación 151
10.9. Ejecución y limpieza 151

11. Interfaces gráficas de usuario **153**
11.1. APIs para la programación de interfaces gráficos de usuario en Java: AWT y Swing 154
11.2. Contenedores y Componentes 155
11.3. Gestores de Aspecto (*Layout Managers*) 155
11.4. Detección de eventos: Escuchadores 157
11.5. Algunos componentes Swing 162
11.5.1. `JLabel`, muestra texto o iconos 162
11.5.2. `JButton`, botones que el usuario puede pulsar 162
11.5.3. `JTextField`, campos de introducción de texto 163
11.5.4. `JRadioButton`, botones de opciones 164
11.5.5. `JCheckBox`, botones de selección múltiple 166
11.5.6. `JList`, listas de selección 166
11.6. El patrón de diseño Modelo/Vista/Controlador 168

12. Applets **173**
12.1. ¿Qué son los Applets? 173
12.2. Ciclo de vida de un Applet 174
12.3. Código HTML para contener un Applet 175
12.4. Lectura de parámetros de la página HTML 176

12.5. Convertir una aplicación Swing en un Applet 176
12.6. Comunicación entre Applets 177

13. Control de errores con *MyLyn* y *Bugzilla* 181
13.1. Sistema de control de tareas *MyLyn* 182
13.1.1. Cual es el objetivo de *MyLyn* 182
13.1.2. Trabajar con *MyLyn* 182
13.2. Sistema de gestión de errores *Bugzilla* 188
13.2.1. Cual es el objetivo de *Bugzilla* 188
13.2.2. Instalación de *Bugzilla* 188
13.2.3. Trabajar con *Bugzilla* 195
13.3. Acceso a *Bugzilla* desde *MyLyn* y *Eclipse* 199
13.3.1. Beneficios de la combinación de *Bugzilla* y *MyLyn* desde *Eclipse* 201
13.3.2. Trabajo con *MyLyn* y *Bugzilla* desde *Eclipse* 201

14. Programación concurrente con Hilos 207
14.1. ¿Qué es un hilo? Utilidades. Consideraciones sobre el uso de hilos 208
14.2. Creación de hilos en Java 209
14.2.1. Creación de un Hilo extendiendo a la clase `Thread` 209
14.2.2. Creación de un Hilo mediante una clase interna 210
14.2.3. Creación de un Hilo mediante una clase interna anónima . 211
14.3. Ciclo de vida de un hilo 212
14.4. Control de hilos 213
14.5. Sincronización 215
14.5.1. Sincronizacón utilizando los cerrojos intrínsecos 215
14.5.2. Sincronización utilizando el `interface Lock` 218

15. Programación para la Red 221
15.1. Trabajo con URLs 222
15.1.1. ¿Qué es una URL? 222
15.1.2. Leer desde una URL 223
15.1.3. Escribir a una URL 223
15.2. Trabajo con Sockets 225
15.2.1. ¿Qué es un Socket? 225
15.2.2. Sockets bajo el protocolo TCP 225
15.2.3. Sockets bajo el protocolo UDP 227

16. Patrones de diseño 231
16.1. Principios de POO 232
16.2. ¿Qué son los patrones de diseño? 233
16.3. ¿Qué es el acoplamiento entre clases y por qué hay que evitarlo? 233
16.4. Grupos de patrones de diseño 233
16.5. El patrón de diseño *Singleton* 233
16.5.1. Situación que intenta resolver 234
16.5.2. Ejemplo de implementación 234
16.6. El patrón de diseño *Factory Method* 235
16.6.1. Situación que intenta resolver 235
16.6.2. Ejemplo de implementación 236
16.7. El patrón de diseño *Abstract Factory* 238

16.7.1. Situación que intenta resolver 238
16.7.2. Ejemplo de implementación 238
16.8. El patrón de diseño *Strategy* 244
16.8.1. Situación que intenta resolver 245
16.8.2. Ejemplo de implementación 245
16.9. El patrón de diseño *Observer* 247
16.9.1. Situación que intenta resolver 247
16.9.2. Ejemplo de implementación 248
16.10. El patrón de diseño *Decorator* 249
16.10.1. Situación que intenta resolver 250
16.10.2. Ejemplo de implementación 250

A. build.xml **255**

B. Aplicación Hipoteca **259**

C. Ejemplo sincronización **265**

Prefacio

La escritura de un libro es una tarea ingente. La motivación para abordarla debe ser, al menos, tan grande como la tarea que se desea acometer. Para nosotros, la motivación ha consistido en escribir un libro que se distinguiera del resto de libros que abordan el aprendizaje del lenguaje de programación Java.

Por un lado, existen excelentes libros que muestran cómo programar en Java. Por otro lado existen excelentes libros, en número inferior, que muestran cómo utilizar herramientas de ayuda y soporte al desarrollo de proyectos en Java. Pensamos que, entre ellos, existía cabida para escribir un libro que abordase el aprendizaje de Java al mismo tiempo que las herramientas imprescindibles de ayuda al desarrollo.

Dentro de nuestra Universidad, la Jaume I, hemos impartido, y seguimos haciéndolo, cursos sobre el lenguaje de programación Java para todo tipo de alumnado: desde alumnos de las distintas titulaciones de informática, alumnos extranjeros en el Master Europeo Erasmus Mundus sobre tecnologías Geoespaciales, hasta profesionales que quieren mantener al día su conocimiento y mejorar sus expectativas laborales. Esta experiencia nos ha dado la confianza suficiente como para animarnos a escribir el presente libro.

Y, a pesar del contacto casi diario con Java y sus tecnologías, reconocemos que aún nos queda mucho por aprender, que el mundo que brinda el aprendizaje de Java es inmenso y que se renueva constantemente. Esto último es síntoma de que la comunidad alrededor de esta tecnología está viva y posee un gran entusiasmo.

Objetivos del libro

Dos son los objetivos principales del este libro:

- Presentar el lenguaje de programación Java.
- Presentar algunas de las herramientas de desarrollo que ayudan en el desarrollo de proyectos utilizando Java.

Con un poco más de detalle, en el primer objetivo hemos pretendido no sólo presentar el lenguaje de programación, además indicamos unas directrices para crear código de calidad, código que sea fácil leer, fácil mantener y que se puede probar de manera automática.

El segundo de los objetivos es casi una necesidad imperiosa a los equipos de desarrollo que siguen utilizando como herramienta de control de versiones un directorio compartido. O a aquellos equipos de desarrollo que siguen probando

sus aplicaciones de manera manual. O para aquellos equipos de desarrollo que utilizan como sistema de seguimiento de errores el correo electrónico. Y un largo etcétera de prácticas desaconsejadas.

Cómo está organizado este libro

La Figura 1 muestra la organización en capítulos del presente libro. Cada uno de los recuadros representa un capítulo. Los capítulos se han agrupado en dos grandes bloques. En el primero de ellos *Java básico* hemos agrupado los capítulos que consideramos introductorios, y que representan el núcleo de la programación orientada a objetos en Java. En el segundo grupo *Java avanzado* aparecen los capítulos que consideramos aspectos avanzados del lenguaje con respecto a los capítulos del primer grupo.

En ambos grupos hay capítulos que no aparecen en la línea principal del flujo, estos capítulos son los que presentan herramientas que consideramos de gran utilidad en el desarrollo de proyectos informáticos utilizando tecnologías Java. El orden de introducción de estas herramientas ha sido fuente de largas conversaciones: ¿Es conveniente introducir al principio la herramienta JUnit siguiendo una orientación hacia el desarrollo guiado por pruebas? ¿Debemos delegar hasta el segundo bloque de capítulos el dedicado a la construcción de proyectos con Ant? Hemos optado por seguir un orden quizás más conservado y menos arriesgado, intentando presentar las herramientas en el momento en que conceptualmente se entienda cual es la necesidad que vienen a cubrir. Esperamos que esta ordenación haga el tránsito suave entre el aprendizaje de Java como lenguaje de programación y las herramientas de ayuda al desarrollo.

Quien debería leer este libro

El publico objetivo de este libro son los desarrolladores que quieran aprender el lenguaje de programación Java y ya posean conocimientos de programación estructurada y orientación a objetos. Los conceptos del lenguaje son presentados desde la base, suponiendo que es la primera vez que el lector se aproxima al lenguaje de programación Java.

Pero este libro también está pensado para aquellas personas que conocen el lenguaje de programación Java y aún no han descubierto la gran cantidad de herramientas de ayuda que existen en el desarrollo de proyecto.

Agradecimientos

La sección de agradecimientos es posiblemente una de las más complicadas de escribir. Debe tener un equilibrio entre el espacio dedicado a ella y el reconocimiento a todas las personas, que de un modo u otro han contribuido a que un libro tenga su forma final.

Para no dejarnos por citar el nombre de nadie, preferimos ampliar nuestro agradecimiento a colectivos. En primer lugar a nuestro alumnos, por que a fin de cuentas es a ellos a los que va dirigido este libro. Con sus preguntas, apreciaciones, comentarios y dudas nos han ayudado a darnos cuenta de donde estaban los escollos en la lectura de este libro.

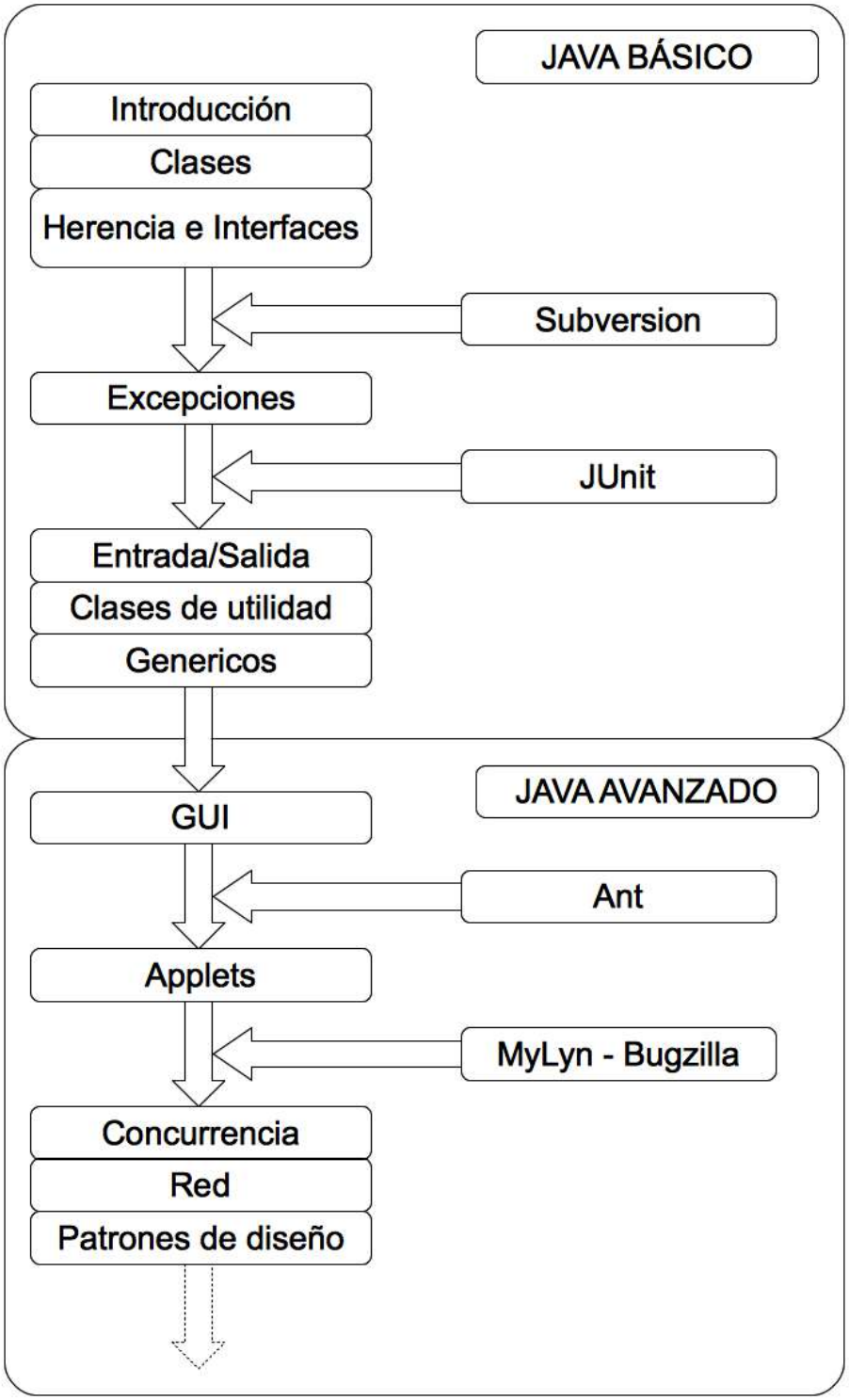

Figura 1: Organización del libro.

También a nuestros compañeros de la Universidad, porque con sus comentarios y rectificaciones nos han ayudado a eliminar errores en los contenidos.

Y finalmente a nuestros amigos por su ánimo constante para que esta labor llegase a buen puerto.

A todos ellos gracias.

Capítulo 1

Introducción

Contenidos

1.1. Origen del lenguaje de programación Java 13
1.2. Característica de Java 14
1.3. El entorno de desarrollo integrado Eclipse 15
1.3.1. Principales características del entorno de desarrollo Eclipse 16
1.3.2. Descarga e instalación de Eclipse 16
1.3.3. Configurando el aspecto de Eclipse: Perspectivas y Vistas 16
1.3.4. El primer ejemplo 18
1.4. Herramientas de desarrollo 21
1.4.1. Añadiendo nueva funcionalidad a Eclipse: los *plug-ins* 22

Introducción

En este capítulo de introducción se muestran los orígenes del lenguaje de programación Java, sus principales características y las diferencias con respecto a C++, lenguaje del que hereda gran parte de la sintaxis.

En la segunda parte del capítulo se presenta en Entorno de Desarrollo Integrado *Eclipse*, los fundamentos para crear proyectos Java utilizando este Entorno y cómo se puede enriquecer su funcionalidad mediante la instalación de *plug-ins*.

1.1. Origen del lenguaje de programación Java

El lenguaje de programación Java tiene sus orígenes en un lenguaje de programación anterior, llamado *Oak* (roble en inglés), que nació de un proyecto interno en *Sun Microsystems* en el año 1991 llamado *Green project*.

Oak fue creado con el objetivo de ser el lenguaje de programación con el que programar dispositivos electrónicos domésticos, en particular aparatos de televisión inteligentes e interactivos. *Oak* tenía, entre otras, las siguientes características de interés:

- Orientado a objetos y de propósito general.
- Robusto.
- Sintaxis parecida a C++.
- Independiente del hardware.

El proyecto de televisión inteligente e interactiva nunca se materializó. De modo simultáneo, a principios de la década de los 90 surgió *Internet* y con ella, la aparición de los primeros *navegadores web*. Los líderes del *Green project* fueron conscientes de la importancia que iba a tener *Internet* y orientaron su lenguaje de programación *Oak* para que programas escritos en este lenguaje de programación se pudiesen ejecutar dentro del navegador web *Mozilla*. Y este fue el inicio de *Java*, así llamado porque cuando se intentó registrar el nombre *Oak* este ya estaba registrado.

Las nuevas características generales que se añadieron son:

- Seguridad, ya que los programas que se ejecutan en un navegador se descargan desde *Internet*.
- Potencia, ya no se tenía la restricción de la ejecución en dispositivos de electrónica de consumo.

Java se ha consolidado como lenguaje de programación gracias a que su curva de aprendizaje es relativamente suave para programadores provenientes de C++. Además, la ventaja de que un programa escrito en Java se puede ejecutar en una gran cantidad de plataformas ha hecho de él un interesante lenguaje de programación por su «universalidad».

1.2. Característica de Java

Java es un lenguaje de programación orientado a objetos y de propósito general que toma de otros lenguajes de programación algunas ideas fundamentales, en particular toma de *Smalltalk* el hecho de que los programas Java se ejecutan sobre una *máquina virtual*. Y del lenguaje de programación *C++* toma su sintaxis.

El uso de la máquina virtual garantiza la independencia de la plataforma en Java. Si disponemos de una máquina virtual para nuestra plataforma, podremos ejecutar el mismo programa escrito en Java sin necesidad de volverlo a compilar.

En el proceso de compilación de un programa en Java, se genera un código intermedio, llamado *bytecode*, que la máquina virtual interpreta y traduce a llamadas nativas del sistema sobre el que se ejecuta la máquina virtual. Así, una máquina virtual para una plataforma Windows 7 de 64 bits, traducirá los *bytecodes* a código nativo para esta plataforma, y otra máquina virtual para una plataforma Linux de 64 bits traducirá los mismos *bytecodes* a código nativo para esta otra plataforma. Los *bytecodes* son los mismos en ambos casos, las máquinas virtuales sobre las que se ejecutan son nativas de la plataforma correspondiente.

Puede parecer que este paso de traducir los *bytecodes* a código nativo de la plataforma suponga una pérdida de rendimiento en la ejecución de los programas en Java, pero esto no es así gracias a la introducción de la tecnología *JIT (Just*

In Time compilation). La idea básica de esta tecnología es que la primera vez que se llama a un método, este se interpreta generando código nativo de la plataforma sobre la que se ejecuta la máquina virtual, pero una vez generado este código nativo, se almacena, de tal modo que la siguiente vez que se llama al mismo método no es necesaria su interpretación ya que el código nativo para ese método se almacenó previamente.

Otras características generales de Java son:

- Seguridad desde el punto de vista del programador:
  - Comprobación estricta de tipos.
  - Gestión de excepciones.
  - No existen punteros.
  - Recolector de basura.
- Seguridad desde el punto de vista del usuario de aplicaciones:
  - Los programas se ejecutan sobre una máquina virtual.
  - Espacio de nombre.
- Soporta programación concurrente de modo nativo.
- Los tipos de datos están estandarizados.
- Sólo se admite herencia simple.

1.3. El entorno de desarrollo integrado Eclipse

Un entorno integrado de desarrollo o IDE de sus siglas en inglés (emphIntegrated Develop Environment) nos permite escribir código de un modo cómodo. La comodidad reside en que los entornos de desarrollo integrados son mucho más que un simple editor de textos. Algunas características comunes a los IDE son:

- Coloreado de la sintaxis.
- Herramientas de búsqueda.
- Asistentes para la escritura de código.
- Ejecución de aplicaciones sin abandonar el entorno.
- Herramientas de depuración de código.

Junto a estas características, los modernos IDE poseen algunas otras realmente espectaculares como por ejemplo:

- Conexión con sistemas de control de versiones.
- Conexión con sistema de seguimiento de errores.
- Facilidades para la creación de tareas.
- Herramientas avanzadas para el análisis de código.

- Herramientas de análisis de rendimiento.
- Conexión a gestores de bases de datos.

Eclipse es un IDE orientado al desarrollo de proyectos con tecnología Java, aunque no es el único lenguaje de programación al que da soporte. Eclipse es una herramienta de software libre, mantenido por la Eclipse Foundation.

1.3.1. Principales características del entorno de desarrollo Eclipse

Eclipse reúne todas las características comunes a los modernos IDE enumeradas más arriba. Además posee un sistema de *plug-ins* con los que se pueden añadir nuevas funcionalidades. Por ejemplo, mediante un plug-in nos podemos conectar al sistema de control de versiones *Subversion*

1.3.2. Descarga e instalación de Eclipse

Eclipse se puede descargar desde el sitio web `http://www.eclipse.org`. Existen versiones para las principales plataformas y sistemas operativos.

Una particularidad de Eclipse es que no necesita instalación. Una vez descargado el fichero comprimido, lo único que debemos hacer para empezar a utilizarlo es descomprimirlo en algún directorio y seguidamente ejecutar el binario correspondiente a nuestra plataforma. La Figura 1.1 muestra la página de inicio de Eclipse. Los iconos que muestra esta pantalla son enlaces a sitios de información sobre Eclipse. La pantalla de inicio se puede cerrar pulsando el aspa que se encuentra a la derecha de la leyenda *Welcome*.

1.3.3. Configurando el aspecto de Eclipse: Perspectivas y Vistas

El interface gráfico de usuario de Eclipse cuenta con dos conceptos fundamentales: las *Perspectivas* y las *Vistas*.

Una *Perspectiva* es un contenedor de *Vistas*. En una misma *Perspectiva* podemos agrupar más de una *Vista*.

Las *Vistas* por su lado, son componentes visual que pueden mostrar desde un editor de código, hasta un árbol de jerarquía de clases en Java. En la figura 1.2 se muestra el aspecto de Eclipse al mostrar la *Perspectiva* por defecto orientada al desarrollo de aplicaciones Java.

Esta perspectiva contiene varias vistas, por ejemplo la vista *Package Explorer* donde se muestra información de la configuración de nuestros proyectos. La vista *Problems* muestra un listado con los errores y avisos presentes en el código de nuestro proyecto. Cada una de estas vistas está orientada a presentar un tipo de información de nuestro proyecto o tareas relacionadas con él.

El aspecto de las perspectivas se puede configurar. Te habrás dado cuenta que cada una de estas vistas está etiquetada con un nombre dentro de una solapa, si haces *click* sobre una de estas solapas sin soltar el botón del ratón, puedes trasladar la vista a cualquier otra posición dentro de la perspectiva. Esto te permite organizar las vistas dentro de las perspectivas según tus preferencias.

Existe una gran cantidad de vistas, puedes acceder a cualquiera de ellas a través de la opción *Show view* del menú *Window*.



Figura 1.1: Pantalla inicial de *Eclipse*

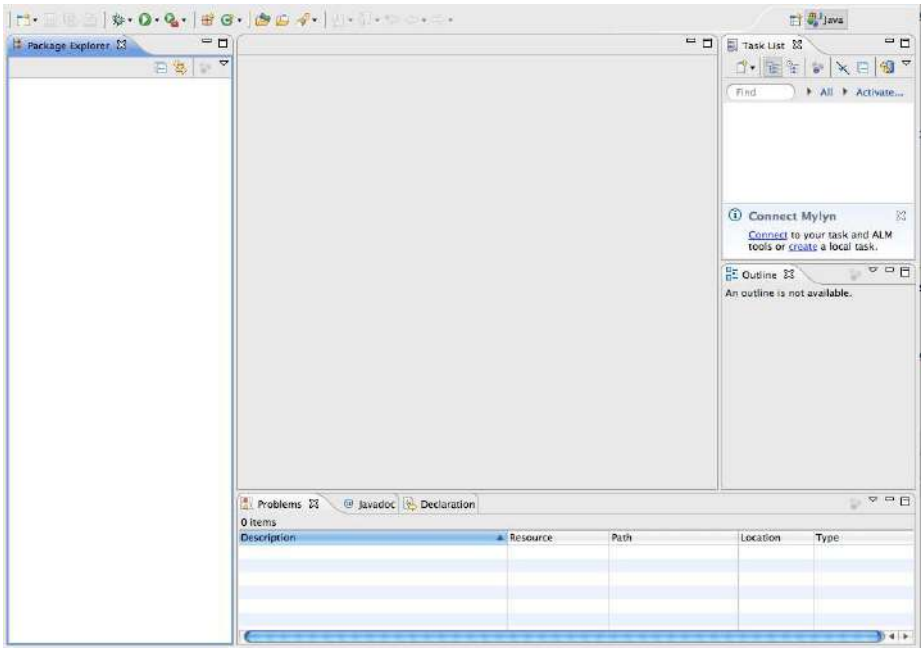

Figura 1.2: Perspectiva inicial orientada al desarrollo de proyectos Java2 SE.

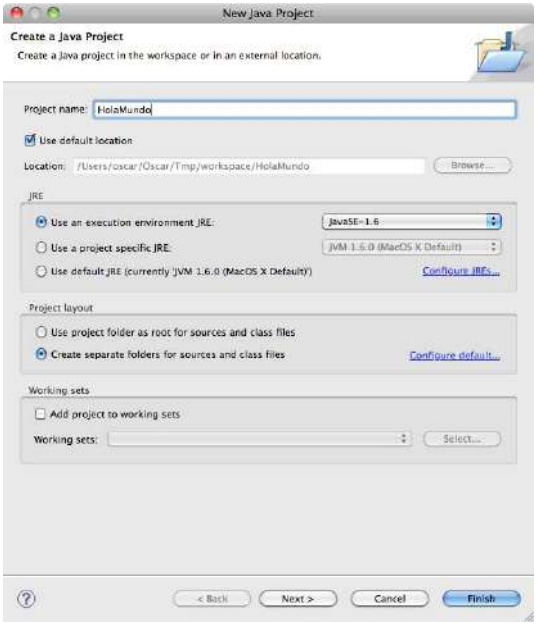

Figura 1.3: Ventana para la creación de un proyecto Java utilizando Eclipse.

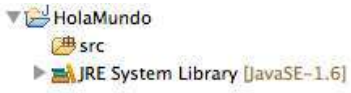

Figura 1.4: Estructura mínima de un proyecto en Eclipse.

1.3.4. El primer ejemplo

Vamos a crear un primer proyecto Java con Eclipse. Para ello, simplemente haz *click* con el botón derecho en la vista *Package Explorer*, y sobre el menú emergente que aparecerá selecciona *New* → *Project*, se abrirá una ventana como la mostrada en la Figura 1.3. En esta ventana lo único que vamos a introducir es el nombre del proyecto.

Una vez introducido el nombre del proyecto, pulsa el botón *Finish*, verás que el aspecto de Eclipse se actualiza para mostrar el nuevo proyecto recién creado. En la vista *Package Explorer* aparecerá el nombre del nuevo proyecto recién creado. La vista de proyecto sigue una estructura de árbol, que puedes desplegar, el resultado se muestra en la Figura 1.4

El siguiente paso que vamos a dar es crear una nueva clase en nuestro proyecto. Esta clase va a ser muy sencilla, y únicamente nos va a servir para conocer cual es el procedimiento de creación, edición, compilación y ejecución utilizando Eclipse. Para crear una nueva clase, haz *click* con el botón derecho del ratón sobre el nombre del proyecto recién creado, se abrirá un menú emergente, selecciona la opción *New* → *Class*, presta atención al icono que se dibuja a la izquierda de esta opción, y verás que ese mismo icono la encuentras en la barrar de herramientas en la parte superior de la ventana de Eclipse. Otro procedimiento, más rápido, de crear una nueva clase en Eclipse es pulsar directamente

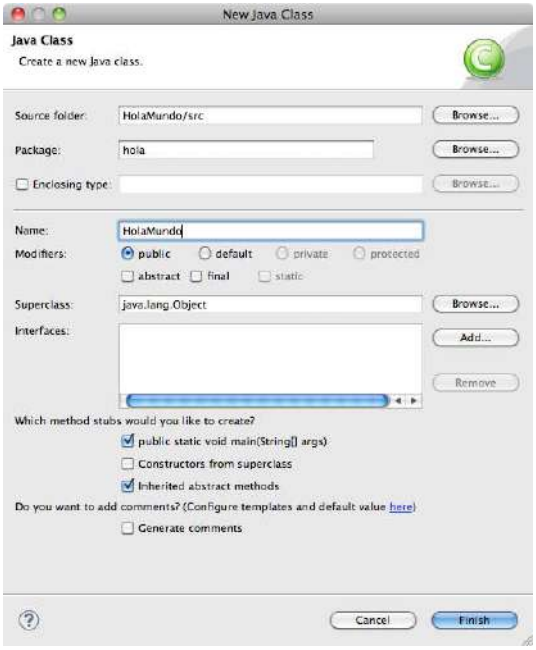

Figura 1.5: Creación de una nueva clase Java en Eclipse.

ese icono en la barra de herramientas.

Al seleccionar esta opción, se abrirá la nueva ventana mostrada en la Figura 1.5. En esta ventana vamos a introducir tres piezas de información:

- Un nombre de paquete en minúsculas (en el Capítulo 3 conocerás con detalle el significado de los paquetes en Java.
- Un nombre de clase con la primera letra de cada palabra en mayúsculas y sin espacios entre ellas.
- Selecciona la opción **public static void main(String[] args)**

Esta tres piezas de información aparecen en la Figura 1.5. Recuerda introducir el nombre del paquete y de la clase utilizando mayúsculas y minúsculas tal y como se muestra en la Figura 1.5, en el Capítulo 2 conoceremos algunas de estas *convenciones de codificación Java*. Finalmente pulsa el botón *Finish*. Verás que de nuevo se actualiza la estructura del proyecto, ahora podrás ver que se ha creado, bajo el nodo del árbol *src* un nuevo nodo con nombre *hola* y bajo él el nodo *HolaMundo.java*. Además, se ha abierto una nueva vista del editor de código tal y como muestra la Figura 1.6.

Hagamos una pequeña modificación sobre este código. Añade la siguiente línea tal y como se muestra en la Figura 1.7. Esta instrucción sirve para mostrar una cadena de texto por consola.

Una vez escrita la nueva línea de código graba el fichero, para ello pulsa la combinación de teclas *Ctrl + S*. El siguiente paso va a ser ejecutar el programa, para ello haz *click* con el botón derecho del ratón sobre el editor de código y en el menú emergente que aparecerá selecciona la opción *Run As* $\rightarrow$ *Java Application*.

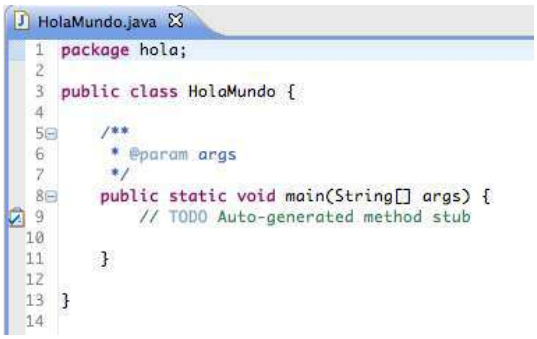

Figura 1.6: Aspecto del editor de código Java.

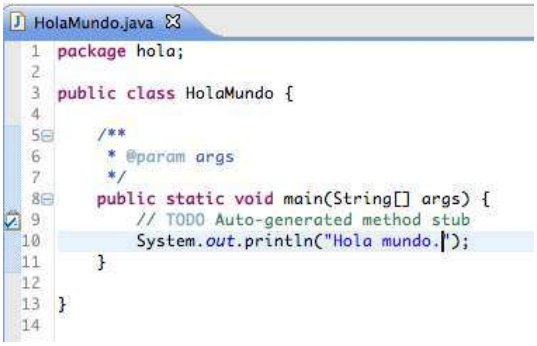

Figura 1.7: El primer programa en Java *Hola mundo.*

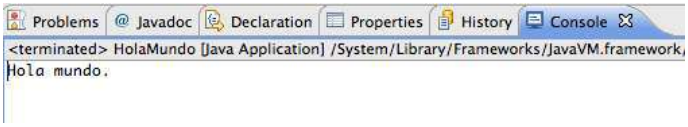

Figura 1.8: Resultado de le ejecución del primer programa en Java *Hola mundo.*

En la vista *Console* podrás ver el resultado de la ejecución del programa, tal y como muestra la Figura 1.8

Por defecto, cada vez que haces modificaciones en el código de definición de una clase y grabas el fichero, Eclipse compila automáticamente el código modificado. De este modo la compilación se realiza de modo transparente a medida que vamos trabajando en nuestro proyecto.

Con estos sencillos pasos hemos creado nuestro primer proyecto en Java, y una única clase en la que hemos introducido una línea de código que muestra un mensaje de texto por consola. El trabajo con Eclipse es realmente sencillo.

1.4. Herramientas de desarrollo

Cuando nos planteamos desarrollar un proyecto informático será de gran ayuda elegir una serie de herramientas que nos faciliten actividades tales como el control de versiones de nuestro código o la prueba automática de nuestros métodos.

En el estado actual de madurez de las tecnologías de desarrollo de proyectos informáticos, resulta impensable iniciar el desarrollo de un proyecto sin planificar el control de versiones, la gestión y seguimiento de errores, las herramientas de despliegue de la aplicación, y un largo etcétera.

Afortunadamente, en la actualidad, contamos con excelentes herramientas de software libre que cubren este tipo de tareas. E incluso, en algunos casos, existe más de una solución de software libre, con lo que podemos evaluar varias de ellas y seleccionar la que mejor se adapte a nuestra forma de trabajar antes de empezar a utilizar.

En este libro vamos a presentar algunas de estar herramientas. Las hemos elegido porque, a través de nuestra propia experiencia, nos han parecido las más adecuadas a nuestro caso, pero como lector no debes seguir ciegamente nuestra elección. Mejor aún, tómala como punto de partida que le permita evaluar otras alternativas. La rápida evolución de la tecnología informática permite vaticinar que seguirán apareciendo cada vez más y mejores alternativas.

En particular vamos a presentar las siguientes herramientas:

Subvesion Es una herramienta para la gestión de versiones.

JUnit Es un *framework* de pruebas automáticas de código.

Ant Es una herramienta de construcción de proyectos.

MyLyn Es una herramienta de gestión de tareas.

Bugzilla Es una herramienta de gestión y seguimiento de errores.

Cada una de las anteriores herramientas cuentan con una gran popularidad dentro de la comunidad de desarrollo de proyectos informáticos. Todas ellas

cuentan con otras excelentes alternativas. Todas ellas se pueden utilizar en proyectos que utilicen un lenguaje de programación alternativo a Java, o existen versiones de ellas para otros lenguajes de programación.

1.4.1. Añadiendo nueva funcionalidad a Eclipse: los *plug-ins*

Afortunadamente, desde Eclipse se puede interaccionar con todas las herramientas expuestas en la sección anterior.

Eclipse cuenta con un sistema de *plug-ins* de tal modo que podemos aumentar sus ya de por sí numerosas y potentes funcionalidades con otras nuevas.

Así, por ejemplo, podemos instalar un *plug-in* de Eclipse para poder realizar el control de versiones de nuestro código sin necesidad de abandonar Eclipse. En el Capítulo 4 se mostrará cómo instalar el *plug-in* para Eclipse y cómo trabajar con él.

Lecturas recomendadas

- Un escueto resumen sobre lo que significa el lenguaje de programación Java se puede encontrar en [11].
- Una referencia completa sobre el entorno de desarrollo Eclipse se puede encontrar en la página web `http://www.eclipse.org`.

Capítulo 2

Clases en Java

Contenidos

2.1. Definición de una clase 24
2.2. Miembros de una clase 25
2.2.1. Atributos de una clase 25
2.2.2. Métodos de una clase. 26
2.2.3. Constructores. 28
2.2.4. Sobrecarga de métodos y constructores 32
2.3. Tipos de datos en Java. 33
2.3.1. Arrays de datos en Java. 34
2.4. Estructuras de control. 36
2.4.1. Estructuras de control de repetición. 37
2.4.1.1. El bucle `for` 37
2.4.1.2. El bucle `while` 38
2.4.1.3. El bucle `do...while` 38
2.4.2. Estructuras de control de selección. 39
2.4.2.1. Bifurcaciones con la sentencia `if...else`. . 39
2.4.2.2. Múltiples caminos con la sentencia `switch` 39
2.5. Modificadores de acceso. 40
2.6. Modificadores `static` y `final`. 42
2.7. El recolector de basura. 43
2.8. Finalización. 44
2.9. Comentarios. Comentarios de documentación. . . 45

Introducción

Las clases son la piedra angular de los lenguaje de programación orientados a objetos (POO). Las clases son abstracciones de entidades de la realidad que sirven como plantillas para la creación de ejemplares de la clase. A estos ejemplares en POO se les llama objetos o instancias de la clase. El proceso de abstracción depende del contexto en el que se utilizarán los ejemplares, es decir, no es lo mismo abstraer la entidad del mundo real «Persona» para utilizarla en una aplicación

de gestión de los clientes de una clínica, que para utilizarla en una aplicación de seguros o de banca. En general, las características de la entidad real que nos interese utilizar en la aplicación y las operaciones que podamos realizar sobre estas abstracciones serán diferentes.

Java es un lenguaje de programación orientado a objetos, y aunque como veremos posee tipos básicos para poder manejar enteros o caracteres, todo en Java es un objeto. De hecho en Java existen dos grandes tipos de datos: tipos de datos primitivos y tipos de datos referencia.

En este capítulo vamos a ver cómo, a partir de una abstracción, podemos transcribirla a código Java para crear una clase, y cómo a partir de una clase podemos crear instancias de esa clase.

Finalmente avanzaremos la idea de reutilización de una clase. En POO la reutilización implica escribir una nueva clase sin partir de cero, sino tomando como base otra clase cuyo comportamiento ampliamos o modificamos. Los detalles de cómo ampliar o extender el comportamiento de una clase base se introducirán en el Capítulo 3.

2.1. Definición de una clase

Supongamos que queremos programar una aplicación de agenda telefónica. El objetivo de nuestra agenda telefónica es gestionar una serie de contactos. Cada uno de estos contactos representa a una `Persona`. Dicho de otro modo cada uno de los contactos de la agenda está creado a partir de la misma plantilla `Persona`, que es la abstracción de una persona del mundo real en el contexto de la aplicación de la agenda telefónica.

¿Qué necesitamos especificar para crear un objeto o ejemplar de la clase `Persona`? Cada uno de los objetos creados a partir de esta clase contendrá una serie de valores que lo identifican, como el nombre y los apellidos del contacto y su número de teléfono. El conjunto de todos los valores de un objeto va a determinar su estado en un momento concreto. Por otro lado, sobre cada uno de los objetos vamos a poder llevar a cabo un conjunto de operaciones definidas en la clase. Volviendo al ejemplo de la agenda telefónica, cada una de las «Persona» de la agenda va a tener una serie de datos de interés, que pueden o no variar a lo largo del tiempo (un contacto de mi agenda puede cambiar de número de teléfono, pero no es probable que cambie de apellidos), y me va a ofrecer una serie de operaciones que puedo realizar sobre ella, como por ejemplo consultar su nombre.

Definición

Al conjunto de valores definidos en la clase se le llama atributos de la clase. Al conjunto de operaciones que define una clase se le llama métodos de la clase. Cuando hablamos de miembros de una clase hacemos referencia tanto a los atributos como a los métodos de la clase.

La definición de una clase en Java empieza con la palabra reservada `class`, y el conjunto de atributos y métodos de la clase se define en un bloque delimitado por llaves, del siguiente modo

```
1 class Persona {
2  // Declaración de atributos
3  // Definición de métodos
4 }
```

2.2. Miembros de una clase

2.2.1. Atributos de una clase

Ahora que ya sabemos que debemos abstraer una «Persona» del mundo real en el contexto de nuestra aplicación la siguiente pregunta es: ¿Cuales son las características, o datos, de una persona relevantes en el contexto de una agenda telefónica? Sin duda uno de estos datos es el número de teléfono de la persona; cada contacto de mi agenda tiene, de manera simplificada, un número de teléfono. ¿Qué otros datos pueden ser de interés almacenar en una agenda telefónica?, parece evidente que, al menos, el nombre y los apellidos de cada uno de los contactos.

Representemos gráficamente lo que tenemos hasta ahora

Persona posee:
→Nombre;
→Apellidos;
→Teléfono;

¿Cómo se definen estos atributos en la clase? De cada uno de los atributos debemos especificar su tipo, por ejemplo, en el caso del Nombre, utilizaremos una cadena de caracteres; en el caso del Telefono podemos optar entre representarlo como un número entero o una cadena de caracteres; si queremos almacenar los números de teléfono en formato internacional (Ej: (+34) 555 555 555) optaremos por representarlos como cadenas de caracteres.

Los atributos los declararemos de este modo:

```
1 class Persona {
2  String nombre;
3  String apellidos;
4  String telefono;
5  // Definición de métodos
6 }
```

Fíjate que, al escribir el nombre de la clase hemos empezado la palabra por una letra mayúscula, y que al empezar el nombre de un atributo lo hemos empezado por minúscula. Esta es una convención de codificación en Java que conviene seguir puesto que está ampliamente extendida entre los desarrolladores Java. Veremos más reglas de convención en la Sección 2.9. Fíjate también que hemos definido cada atributo en un línea distinta y que cada línea acaba con el caracter `;`.

Reglas de convención

Según las reglas de convención más extendidas en Java, al definir una clase, el nombre de la clase se debe escribir con la primera letra en mayúscula y los nombres de los atributos y métodos deben empezar por una letra en minúscula. Si estos nombres están formados por más de una palabra, la segunda y siguientes palabras que constituyen el nombre se escriben con su primera letra en mayúscula. Por ejemplo: numeroTelefono.

Veamos ahora cómo definir las operaciones que podremos realizar sobre las instancias de la clase `Persona`.

2.2.2. Métodos de una clase.

Una vez hemos creado una instancia de la clase `Persona`, ¿Cómo podemos recuperar a partir de ella su nombre?, ¿Cómo podemos recuperar el nombre que almacenamos en un contacto de nuestra agenda?.

Una posibilidad es simplemente leer el valor del atributo, pero como veremos en la sección 2.5 el acceso directo a los atributos de una clase está desaconsejado.

La respuesta es: a través de una llamada a un método que devuelva el nombre del contacto. En el caso de la recuperación del nombre, el tipo de dato de retorno es una cadena `class String`. Un método que cumple este objetivo es el siguiente:

```
1 String getPersona() {
2  return nombre;
3 }
```

Sintaxis

La sintaxis de declaración de un método es:
{modificadores} tipoRetorno nombre(tipo argumento1, tipo argumento2, ...) {
    Bloque de definición del método;
}

En estas pocas líneas de código hay varias novedades, veámoslas:

1. Un método tiene un nombre que lo identifica, en este caso `getNombre`.
2. Delante del nombre del método escribimos el tipo del valor de retorno, en nuestro caso, como lo que el método devuelve es el nombre cuyo tipo es un `String`, este será el tipo del valor retorno.
3. Detrás del nombre del método aparecen unos paréntesis sin nada en su interior. Dentro de los paréntesis se define la lista de argumentos del método. Si estás familiarizado con las matemáticas, los argumentos de los métodos tienen el mismo significado que los argumentos de las funciones matemáticas, por ejemplo seno(45º) significa que queremos utilizar el cálculo del seno sobre el argumento 45 grados. En nuestro caso la lista está vacía, lo que indica que no necesito especificar ningún argumento para poder hacer uso del método.

4. La definición del método va dentro de las llaves .

5. Para devolver un valor utilizamos la palabra reservada `return`.

Como ves, muchos conceptos nuevos en tan sólo tres líneas de código. Y una nueva convención de codificación, si un método devuelve el valor de un atributo empieza por la palabra inglesa `get`, de ahí que hayamos escrito `getNombre()`.

Con lo que ya hemos visto, es sencillo escribir dos nuevos método que devuelvan los apellidos y el número de teléfono de una `Persona`. Aquí tienes el código de la clase:

```
1 class Persona {
2  String nombre;
3  String apellidos;
4  String telefono;
5
6  String getPersona() {
7   return nombre;
8  }
9
10  String getApellidos() {
11   return apellidos;
12  }
13
14  String getTelefono() {
15   return telefono;
16  }
17 }
```

Listado 2.1: Código de la clase `Persona`

De nuevo, fíjate que si un método no recibe argumentos su lista de argumentos está vacía. Pero si un método no devuelve ningún parámetro, hay que indicarlo explícitamente utilizando la palabra reservada `void`. Por ejemplo, el siguiente método no devuelve ningún valor:

```
1 void nada() {
2  // Código del método
3 }
```

En muchas ocasiones resulta interesante poder modificar el valor de los atributos. Como ya hemos comentado anteriormente, un contacto de mi agenda podría cambiar de número de teléfono, luego parece buena idea que la clase `Persona` me proporcione un método que permita modificar el valor del atributo `telefono`, como el que se muestra en el siguiente ejemplo:

```
1 void setTelefono(String nuevoTelefono) {
2  telefono = nuevoTelefono;
3 }
```

Listado 2.2: Método para modificar el valor del teléfono.

De nuevo, hemos seguido una convención de codificación:

Regla de convención

Los métodos que modifican el valor de los atributos de una clase se nombran empezando con la palabra inglesa *set* seguida por el nombre del atributo, cuya primera letra se escribe en mayúsculas.

De modo análogo, podemos añadir a la clase `Persona` dos nuevos métodos para poder modificar el valor de los atributos `nombre` y `apellidos`, tal y como se muestra a continuación:

```
1 void setNombre(String nuevoNombre) {
2  nombre = nuevoNombre;
3 }
4
5 void setApellidos(String nuevosApellidos) {
6  apellidos = nuevosApellidos;
7 }
```

Listado 2.3: Métodos para modificar el nombre y los apellidos de una `Persona`

Ya tenemos escritos métodos que nos permiten leer y modificar los atributos de la clase `Persona`. Ahora queremos crear ejemplares de esta clase, para ello necesitamos escribir métodos especiales que nos sirvan para crear instancias de la clase, a estos métodos especiales se les llama constructores de la clase.

2.2.3. Constructores.

Para crear un ejemplar de una clase utilizamos métodos especiales llamados constructores de la clase. En las siguientes líneas de código se muestra cómo se define un constructor de la clase `Persona`:

```
1 Persona(String nombre, String apellidos, String telefono) {
2  this.nombre = nombre;
3  this.apellidos = apellidos;
4  this.telefono = telefono;
5 }
```

Listado 2.4: Constructor con parámetros de la clase `Persona`

Volvemos a tener nuevos conceptos en estas líneas de código, veámoslo:

1. Un constructor es un método cuyo nombre coincide con el de la clase, en nuestro caso el nombre del método es `Persona` que es precisamente el nombre de la clase.

2. Como cualquier otro método, tiene un lista de argumentos que en este caso no está vacía, si no que indica que va a recibir tres argumentos y los tres de tipo `String`.

3. Fíjate que los nombres de los tres argumentos coinciden con los nombres de los atributos; la clase tiene declarado un atributo de tipo `String` llamado `nombre` y el primer argumento del constructor también se llama `nombre` y es de tipo `String`. ¿Cómo resolvemos la ambigüedad entre el nombre del atributo y el nombre del argumento?, utilizando la palabra reservada `this`; si escribimos `this.nombre` estamos haciendo referencia al atributo, si sólo escribimos `nombre`, estamos haciendo referencia al argumento del método. Veremos con más detalle el significado de la palabra reservada `this` en la sección 2.3.

4. Un constructor no devuelve ningún valor de retorno, ya que estos métodos especiales nos sirven para crear objetos.

Escribamos otro constructor para la clase `Persona`:

```
1 Persona() { }
```

Más novedades conceptuales en estas líneas de código:

1. La lista de argumentos de este constructor está vacía.
2. No hemos escrito ninguna línea de código entre las llaves.

A este constructor tan particular se le llama *Constructor por defecto* y hablaremos más sobre él en el capítulo 3 dedicado a la herencia en Java. De momento quédate con la idea de que es importante que tus clases definan el constructor por defecto, de hecho, todas tus clases deberían definirlo. Si tu clase no proporciona ningún constructor, como en el caso del Listado 2.5, el compilador de Java crea el constructor por defecto para la clase, de modo que puedas crear instancias a partir de ella.

```
1 class SinConstructores {
2  private int a;
3
4  int getA() {
5   return a;
6  }
7 }
```

Listado 2.5: Una clase sin ningún constructor. El compilador de Java creará el constructor por defecto por nosotros.

Veamos todo el código que hemos escrito para la clase `Persona`:

```
1 package agenda;
2
3 public class Persona {
4  String nombre;
5  String apellidos;
6  String telefono;
7
8  Persona() { }
9
10  Persona(String nombre, String apellidos, String telefono) {
11   this.nombre = nombre;
12   this.apellidos = apellidos;
13   this.telefono = telefono;
14  }
15
16  String getNombre() {
17   return nombre;
18  }
19
20  String getApellidos() {
21   return apellidos;
22  }
23
24  String getTelefono() {
25   return telefono;
26  }
27 }
```

Listado 2.6: Código de la clase Persona

En el Listado 2.6 hemos agrupado los métodos get/set para cada uno de los atributos, además hemos modificado la definición de los métodos set para

deshacer la ambigüedad entre el nombre de los atributos y de los argumentos, tal y como hemos hecho en el caso del constructor con argumentos.

Antes de pasar adelante, escribamos nuestra primera pequeña aplicación en Java para probar todo lo que hemos visto hasta ahora. Vamos a utilizar para ello el entorno integrado de desarrollo *Eclipse*, inicia pues esta aplicación. Hay varias opciones para crear un nuevo proyecto en *Eclipse*, a través del menú puedes elegir *File* → *New* → *Java Project*, o bien puedes pulsar el botón de creación de proyectos. Eclipse te solicitará un nombre para el proyecto, introduce uno adecuado (por ejemplo «AgendaTelefonica»), y ya puedes pulsar directamente la tecla *Finish*. Verás que en la columna izquierda de Eclipse, donde se muestra la vista *Package Explorer* te aparece una carpeta con el mismo nombre que el proyecto recién creado. *Eclipse* organiza los proyectos en carpetas, el código de tu proyecto, ficheros de bibliotecas y recursos necesarios estarán en la carpeta del proyecto.

Para crear un nueva clase en *Eclipse* puedes hacerlo a través del menú *File* → *New* → *Class*, o bien pulsando directamente el botón de creación de una nueva clase. Se abrirá una ventana de diálogo solicitándote un nombre para la nueva clase y el paquete donde se incluirá. Es muy recomendable que cada clase esté dentro de un paquete (veremos con más detalle el significado de los paquetes en Java en la Sección 3.6). Según las convenciones de Java, los nombres de paquetes se escriben en minúscula. Escribe, por ejemplo, para el nombre del paquete `agenda`, y para el nombre de la clase `Persona`. Verás que se abre la vista del editor de código en Eclipse y que si despliegas la carpeta de proyecto te aparece el fichero de clase `Persona.java`. Escribe la definición de la clase según el Listado 2.6.

Lo siguiente que vamos a hacer es escribir una clase para probar nuestra clase `Persona`, para ello crea en el proyecto una nueva clase y llámala `PruebaPersona` y como nombre de paquete introduce `agenda`, y en el cuadro de diálogo de creación de la clase marca la casilla `public static void main(String[] args)`, con ello Eclipse creará de manera automática el método principal `main`. Escribe el resto de código que aparece en el Listado 2.7.

```
1 package agenda;
2
3 public class PruebaPersona {
4
5  /**
6   * @param args
7   */
8  public static void main(String[] args) {
9   // TODO Auto-generated method stub
10  Persona unaPersona = new Persona("Óscar", "Belmonte", "1234");
11  System.out.println("Muestra información accediendo directamente a los
        campos.");
12  System.out.println("Nombre: " + unaPersona.nombre);
13  System.out.println("Apellidos:" + unaPersona.apellidos);
14  System.out.println("Teléfono: " + unaPersona.telefono);
15
16  System.out.println("Muestra información llamando a los métodos de la
        clase.");
17  System.out.println("Nombre: " + unaPersona.getNombre());
18  System.out.println("Apellidos:" + unaPersona.getApellidos());
19  System.out.println("Teléfono: " + unaPersona.getTelefono());
20 }
21
22 }
```

Listado 2.7: Código de la clase Principal

La clase *Principal* está repleta de novedades. Esta clase tiene un único método `public static void main(String[] args)`, este método es el punto de entrada a la ejecución de un programa Java. En las siguientes secciones veremos el significado de todos los modificadores que tiene este método delante de su nombre que es `main`. En la línea número 10, vemos cómo se usa el operador `new` para crear una instancia de la clase, escribimos tras `new` un constructor de la clase, en este caso `Persona("Óscar", "Belmonte", "1234")`,`new` utilizará el constructor con tres argumentos de la clase `Persona` para crear una nueva instancia. Fíjate que a la izquierda de `new` tenemos `Persona unaPersona =`, esto indica que nos guardamos lo que el operador `new` devuelve en la variable de tipo referencia a `Persona` que llamamos `unaPersona`, en las siguientes secciones veremos con más detalle qué significa el concepto *variable de tipo referencia*, de momento la idea es que, para poder usar la instancia a la `Persona` recién creada utilizaremos la variable de referencia `unaPersona`.

Reglas de convención

Los nombre de los paquetes y subpaquetes se escriben en minúsculas.

En las líneas 12-14 recuperamos la información a partir de la variable de tipo referencia a `Persona` accediendo directamente a sus atributos (`nombre, apellidos, telefono`); mientras que en las líneas 17-19 accedemos a la misma información haciendo uso de los métodos definidos en la clase (`getNombre(), getApellidos(), getTelefono()`).

Finalmente, para mostrar información en forma de texto por consola utilizamos `System.out.println("Texto")`.

Ejecutemos este primer programa para ver cual es el resultado, para ello haz click con el botón derecho sobre el nombre de la clase `Principal.java` que tienes en la columna de la derecha en Eclipse (*Package Explorer*) y en el menú emergente que te aparecerá selecciona *Run as* → *Java Application*; en la parte inferior de Eclipse se abrirá una nueva solapa (*Console*) donde se mostrará el resultado de la ejecución que debe ser:

```
Muestra información accediendo directamente a los campos.
Nombre: Óscar
Apellidos:Belmonte
Teléfono: 1234
Muestra información llamando a los métodos de la clase.
Nombre: Óscar
Apellidos:Belmonte
Teléfono: 1234
```

Como has comprobado, el trabajo de edición en Eclipse es realmente sencillo y te irás dando cuenta que este entorno de programación Java es muy potente.

Pregunta

En el ejemplo anterior estamos recuperando la información almacenada en una instancia de la clase `Persona` de dos modos: accediendo directamente a sus atributos, o llamando a los métodos de la clase. ¿Qué sentido tiene declarar métodos de acceso a los atributos de una clase si puedo acceder directamente a ellos?.

2.2.4. Sobrecarga de métodos y constructores

Dos o más métodos pueden tener el mismo nombre siempre que su número de argumentos sea distinto. En caso de que los dos métodos tengan el mismo número de argumentos, serán distintos si al menos un tipo de sus argumentos es distinto. Por ejemplo en el siguiente Listado los dos métodos `unMetodo` están sobrecargados y son distintos.

```
1 public void unMetodo(int entero) {
2   // Definición del método
3 }
4
5 public void unMetodo(float real) {
6   // Definición del método
7 }
```

De modo análogo, los constructores también pueden estar sobrecargados, de hecho hemos sobrecargado el constructor de la clase `Persona` en el Listado 2.6, esta clase tiene dos constructores `Persona()` y `Persona(String nombre, String apellidos, String telefono)`.

Un detalle muy importante en la sobrecarga de métodos es que el tipo de retorno no sirve para distinguir dos métodos. Si dos métodos tienen el mismo número de argumentos y sus tipos son los mismos, no los podremos sobrecargar haciendo que el tipo de sus valores de retorno sean distintos.

Definición

El nombre de un método junto con su lista de argumentos forman la signatura del método. El tipo del valor de retorno no forma parte de la signatura de un método.

En el siguiente Listado se muestra un error al intentar sobrecargar dos métodos que se distinguen únicamente por su tipo de retorno.

```
1 // ESTE LISTADO CONTIENE UN ERROR.
2 // LOS MÉTODOS NO SE PUEDEN SOBRECARGAR POR EL TIPO DE RETORNO.
3 public void unMetodo() {
4   // Definición del método
5 }
6
7 public int unMetodo() {
8   // Definición del método
9 }
```

Los dos métodos tienen el mismo nombre y ningún argumento, el primero de ellos no retorna nada `void`, y el segundo de ellos retorna un `int`. El compilador es

Tipo	Tamaño(bits)	Definición
boolean	1	true o false
char	16	Carácter Unicode
byte	8	Entero en complemento a dos con signo
short	16	Entero en complemento a dos con signo
int	32	Entero en complemento a dos con signo
long	64	Entero en complemento a dos con signo
float	32	Real en punto flotante según la norma IEEE 754
double	64	Real en punto flotante según la norma IEEE 754

Tabla 2.1: Tipos de datos primitivos en Java y sus tamaños en memoria.

incapaz de distinguirlos y devuelve un error que indica que estamos intentando definir el mismo método dos veces.

Pero volvamos a Java y vemos qué significa el término *tipo de dato referencia*.

2.3. Tipos de datos en Java.

En Java existen dos grandes grupos de tipos de datos, los tipos de datos primitivos y los tipos de datos referencia.

Los tipos de datos primitivos sirven para representar tipos de datos tales como números enteros, caracteres, números reales, booleanos, etcétera. Se les llama primitivos porque nos permiten manejar elementos de información básicos como letras y números. Una variable de tipo primitivo nos permite almacenar en ella un tipo primitivo como por ejemplo un valor numérico.

Por otro lado, los tipos de datos referencia nos permiten indicar que vamos a trabajar con instancias de clases, no con tipos primitivos. Una variable de tipo referencia establece una conexión hacia un objeto, y a través de esta conexión podremos acceder a sus atributos y métodos.

Cuando hablamos de variables, es muy importante asimilar la diferencia entre variables de tipo primitivo y variables de tipo referencia. En una variable de tipo primitivo podemos *almacenar* valores de tipo primitivo (números, caracteres); pero el las variables de tipo referencia *no almacenamos valores* son la puerta de entrada hacia los objetos. Son los objetos, las instancias de clases, las que almacenan información y me permiten trabajar con ellos a través de llamadas a sus métodos.

Concepto

Las variables de tipo primitivo nos permiten almacenar valores de tipo primitivo como números y caracteres. Las variables de tipo referencia no almacenan valores, sino que nos permiten acceder a los atributos y métodos de los objetos.

En Java, el tamaño en memoria de los tipos de datos primitivos está estandarizado. Los tipos de datos primitivos y sus tamaños son los que aparecen en la Tabla 2.1.

Sintaxis

Las variables de tipo primitivo se declaran de este modo:
tipo nombre [= valor inicial];
Ejemplo 1: `int hojas;`
Ejemplo 2: `float pi = 3.14f; //fíjate en la f al final del número`

Como ya hemos visto, las referencias en Java son la puerta de entrada a los objetos, las referencias me permiten acceder a los atributos y métodos de los objetos, el tipo de una referencia debe ser compatible con el tipo del objeto al que se refiere. En el capítulo 3 dedicado a la herencia veremos qué quiere decir «compatible».

Sintaxis

Las variables de tipo referencia se declaran de este modo:
tipoReferencia nombre [= valor referencia inicial];
Ejemplo 1: `Persona persona;`
Ejemplo 2: `Persona persona = new Persona("Óscar", "Pérez", "123");`

En múltiples ocasiones, nos interesa trabajar con más de un único valor de un determinado tipo, en vez de trabajar con una única `Persona` queremos trabajar con un grupo de personas. Veamos cómo podemos declarar conjuntos de elementos del mismo tipo en Java.

2.3.1. Arrays de datos en Java.

Hasta el momento, hemos aprendido cómo declarar variables de tipos de datos primitivos y de tipos de datos referencia. Esto nos sirve para crear una única variable que contendrá bien un tipo de datos primitivo a una referencia a un objeto, pero a veces nos interesa poder manejar conjuntos de elementos del mismo tipo, por ejemplo, en alguna circunstancia nos puede interesar declarar una variable con la que poder acceder a un grupo de 10 enteros o 100 objetos de la clase `Persona`.

En Java utilizaremos arrays de elementos cuando necesitemos manejar más de un elemento del mismo tipo. Para declarar un array en Java utilizamos los corchetes según la siguiente sintaxis:

Sintaxis

Los arrays de tipos primitivos se declaran:
Ejemplo 1: `int array[]; // Array declarado`
Ejemplo 2: `int arrayEnteros[] = new int[10]; // Array iniciado`
Los arrays de tipos referencia se declaran:
Ejemplo 3: `Persona grupo[]; // Array declarado`
Ejemplo 4: `Persona grupo = new Persona[10]; // Array iniciado`

Aunque la sintaxis de la declaración de arrays de tipo primitivo y de tipo referencia es la misma, el resultado es radicalmente distinto en los dos casos.

Analicémoslo. En el Ejemplo 2 del recuadro de sintaxis anterior se está definiendo un array capaz de albergar 10 enteros (con índice 0 para el primer elemento e índice 9 para el último), dentro de cada una de las posiciones del array podemos almacenar un entero.

En el caso del Ejemplo 4, estamos definiendo un array capaz de albergar 10 «referencias» de tipo `Persona`. En este caso, lo que tenemos en cada una de las posiciones del array no es un objeto de tipo `Persona`, si no una referencia a un objeto de tipo `Persona`. Dicho de otro modo **No se ha creado ningún objeto de la clase `Persona`, sólo referencias a objetos de ese tipo**.

La diferencia entre arrays de tipo primitivo y tipo referencia es muy importante. Mientras que en el caso de los arrays de tipo primitivo, una vez creados ya tenemos disponible en cada una de sus posiciones espacio para albergar un elemento del tipo correspondiente, en los arrays de tipo referencia no se ha creado ninguna instancia de la clase correspondiente, lo único que se ha creado es un conjunto de referencias que podremos conectar a objetos de la clase correspondiente, y estos objetos los habremos creado en otro lugar de nuestro programa.

Veamos esta diferencia con el siguiente ejemplo

```
1 package agenda;
2
3 public class Arrays {
4
5  /**
6   * @param args
7   */
8  public static void main(String[] args) {
9   // TODO Auto-generated method stub
10  int arrayEnteros[] = new int[10];
11  Persona grupoPersonas[] = new Persona[10];
12  // La siguiente sentencia es valida
13  System.out.println("Valor en arrayEnteros[5]: " + arrayEnteros[5]);
14  // Se produce un error, no hay nada en la posición [5]
15  System.out.println("Nombre en posición grupoPersonas[5]: " +
        grupoPersonas[5].nombre);
16
17 }
18
19 }
```

Listado 2.8: Diferencia entre arrays de tipos primitivos y arrays de tipos referencia

Si creas una nueva clase con el código del Listado 2.8 y lo ejecutas (recuerda: botón derecho sobre el nombre de la clase en el *Package Explorer*, y luego *Run as* → *Java Applications*), obtendrás el siguiente error:

```
Valor en arrayEnteros[5]: 0
Exception in thread "main" java.lang.NullPointerException
at hola.Arrays.main(Arrays.java:15)
```

En la posición 5 del array de enteros tenemos un valor por defecto, pero la referencia que tenemos en la posición 5 del array de tipo `Persona` es el valor por defecto *null* que en Java tiene el significado de *Referencia no asignada*.

Sintaxis

A los elementos de un array se accede mediante el operador []. Dentro de este operador indicamos la posición del elemento a la que deseamos acceder.

¿Cómo podemos resolver el error anterior?. Simplemente asignando a la referencia en la posición 5 del array `grupoPersonas` una referencia a un objeto que haya sido creado:

```
1 package agenda;
2
3 public class Arrays2 {
4
5  /**
6   * @param args
7   */
8  public static void main(String[] args) {
9   // TODO Auto-generated method stub
10  int arrayEnteros[] = new int[10];
11  Persona grupoPersonas[] = new Persona[10];
12  grupoPersonas[5] = new  Persona("James", "Gossling", "555 123 456");
13  // La siguiente sentencia es valida
14  System.out.println("Valor en arrayEnteros[5]: " + arrayEnteros[5]);
15  // Se produce un error, no hay nada en la posición [5]
16  System.out.println("Nombre en posición grupoPersonas[5]: " +
        grupoPersonas[5].nombre);
17
18 }
19
20 }
```

Si ejecutas el código con la modificación obtendrás el siguiente resultado:

```
Valor en arrayEnteros[5]: 0
Nombre en posición grupoPersonas[5]: James
```

En este último caso la referencia en la posición 5 del array `grupoPersonas` sí que hace referencia a un objeto, luego no hay problema al usarla para acceder a su atributo `nombre`.

Ya sabemos cómo acceder a los elementos de un array, la pregunta que nos surge es ¿Cómo puedo recorrer todos los elementos de un array?. La respuesta es: «Usando estructuras de control de repetición»

2.4. Estructuras de control.

Java es un lenguaje de programación estructurado, esto significa que Java proporciona estructuras de control para decidir el flujo de ejecución de nuestros programas.

Existen dos grandes grupos de estructuras de control:

Estructuras de control de repetición: Nos permiten indicar si un determinado bloque de código se debe ejecutar mas de una vez.

Estructuras de control de selección: Nos permiten especificar mas de una dirección de flujo dependiendo de alguna condición.

2.4.1. Estructuras de control de repetición.

En Java existen tres estructuras de control de repetición:

- Bucle for.
- Bucle while.
- Bucle do...while.

Las estructuras de repetición sirven para repetir una determinada tarea mientras se cumpla cierta condición. En el caso de un array nos sirven para recorrer los elementos almacenados en el array secuencialmente, para, por ejemplo, mostrar sus valores. Veamos como se usa cada una de estas estructuras de repetición.

2.4.1.1. El bucle `for`

Si conocemos cual es el primer elementos y el último sobre los que queremos iterar el bucle `for` es la manera más cómoda de recorrerlos todos. Su sintaxis es la siguiente:

Sintaxis

La sintaxis del bucle `for` es:
for(inicio; condición: incremento)
Ejemplo 1: `for(int i = 0; i <10; i += 2)`
La variable «i» se declara en el bucle y sólo tiene existencia dentro del bucle, al salir del bucle desaparece la variable de control «i». Para el bucle for...each:
for(Tipo variable: Colección)
Ejemplo 2:

```
int arrayEnteros [] = new int[10];
for(int i: arrayEnteros)
```

En el primer ejemplo del recuadro de sintaxis se utiliza una variable de control que se inicia a 0, la condición de parada es que el valor de la variable sea menor que 10 y el incremento en cada paso del bucle es 2, luego la variable toma los valores 0, 2, 4, 6 y 8.

En el segundo ejemplo se utiliza el bucle `for...each` introducido en la versión 5 de Java. En este caso utilizamos una variable que va recibiendo los valores de los elementos que hay dentro del conjunto de manera incremental, uno con cada iteración. El bucle `for...each` es especialmente útil cuando se itera sobre los elementos de una colección, tal y como veremos en el Capítulo 8.

Veamos un ejemplo con un poco más de detalle:

```
1 package repeticion;
2
3 public class BucleFor {
4
5  public static void main(String[] args) {
6   // Declaramos el array
7   int arrayEnteros[] = new int[5];
8   // Almacenamos datos en sus elementos
9   for(int i = 0; i < 5; i++)
```

```
10    arrayEnteros[i] = i;
11   // Lo recorremos y extraemos la información almacenada
12   for(int i: arrayEnteros)
13    System.out.println("arrayEnteros[" + i + "] = " + arrayEnteros[i]);
14  }
15
16 }
```

El resultado de la ejecución de este código es el siguiente:

```
arrayEnteros[0] = 0
arrayEnteros[1] = 1
arrayEnteros[2] = 2
arrayEnteros[3] = 3
arrayEnteros[4] = 4
```

El primer bubcle `for` itera sobre las posiciones del array almacenando los números 0 a 4 en las posiciones 0 a 4 del array. El segundo bucle itera sobre las posiciones del array y muestra el valor almacenado en cada una de ellas.

2.4.1.2. El bucle `while`

En el caso del bucle `while`, la condición de parada se comprueba antes de cada iteración y, si la condición se cumple, se ejecuta el bloque del bucle `while`.

Sintaxis

La sintaxis del bucle `while` es:
while(condición) {
Bloque de código
}

2.4.1.3. El bucle `do...while`

En el caso del bucle `do...while` la condición se comprueba después de haberse ejecutado al menos una vez el cuerpo del bucle. La condición se comprueba al final del bucle.

Sintaxis

La sintaxis del bucle `do...while` es:
do {
Bloque de código
} while(condición);

Estas tres estructuras de control de repetición son intercambiables, se puede sustituir una por otra con pequeñas modificaciones. Elegir una u otra depende de cada caso: si conocemos el intervalo sobre el que queremos iterar el bucle `for` es el mas comodo de utilizar; si la condición de parada no involucra el valor de una posicion podemos utilizar el bucle `while` si necesitamos comprobar la condición antes, o bien el bucle `do...while` si queremos ejecutar al menos una vez el bloque de código que encierra el bucle.

2.4.2. Estructuras de control de selección.

Las estructuras de control de selección nos permiten especificar mas de un posible camino a seguir por el flujo de ejecución de nuestro programa. La dirección final que seguirá el flujo dependerá de si se cumple o no cierta condición.

2.4.2.1. Bifurcaciones con la sentencia `if...else`.

La sentencia `if` nos permite ejecutar un bloque de código, o no, dependiendo de cierta condición. La condición debe evaluarse a un valor booleano, es decir, a `true` o `false`, como en el siguiente ejemplo de código:

```
1  int entero;
2  if(entero %2 == 0) System.out.println("El número es par.");
3  else System.out.println("El número es impar.");
```

Dentro del bloque de código correspondiente al `else` podemos añadir una nueva sentencia `if`, se dice entonces que las sentencias `if` están encadenadas, como en el siguiente ejemplo:

```
1  if(primeraCondicion) {
2    \\ Bloque de código
3  } else if(segundaCondicion) {
4    \\ Bloque de código
5  } else {
6    \\ Bloque de código
7  }
```

Esto nos permite especificar más de un posible camino a seguir por el flujo de ejecución de nuestro código.

2.4.2.2. Múltiples caminos con la sentencia `switch`

Existe una construcción del lenguaje que nos permite especificar múltiples caminos a seguir por el flujo de ejecución de nuestro código: la sentencia `switch`. En este caso el camino a seguir se selecciona basándose en el valor de una expresión que se evalua a un valor entero, como en el siguiente ejemplo:

```
1 int mes = 1; // Corresponde al mes de Enero
2 swith(mes) {
3   case 1:
4     System.out.println("El mes es Enero.");
5     break;
6   case 2:
7     System.out.println("El mes es Febrero.");
8     break;
9   case 3:
10    System.out.println("El mes es Marzo.");
11    break;
12  default:
13    System.out.println("Ninguno de los meses anteriores.");
14    break;
15 }
```

En el ejemplo anterior, se evalua el valor de la variable `mes`, y se prueba cada una de las opciones expresadas por un `case`. Cuando coinciden los valores, se ejecuta el código correspondiente al `case` hasta que se encuentra la sentencia `break` en cuyo momento se avandona el bloque de la sentencia `switch`. Exite una

opción *Por defecto* etiquetada como `default` que es opcional y cuyo código se ejecutará si la expresión entera no coincide con ninguno de los `case` anteriores.

Es importante hacer notar que una vez que se encuentra una coincidencia entre la expresión entera y un `case`, se ejecuta su código correspondiente hasta encontrar la sentencia `break`. Esto nos permite obviar esta sentencia si queremos que varios `case` distintos ejecuten el mismo segmento de código, como en el siguiente ejemplo:

```
1 int mes = 1; // Corresponde al mes de Enero
2 switch(mes) {
3   case 1:
4   case 3:
5   case 5:
6   case 7:
7   case 8:
8   case 10:
9   case 12:
10    System.out.println("El mes tiene 31 días.");
11    break;
12  case 4:
13  case 6:
14  case 9:
15  case 11:
16    System.out.println("El mes tiene 30 días.");
17    break;
18  default:
19    System.out.println("El mes es Febrero.");
20    break;
21 }
```

En el ejemplo anterior los meses cuyo ordinal es 1, 3, 5, 7, 8, 10 o 12 tienen 31 días, todos los `case` correspondientes, excepto el de valor 12, no incluye la sentencia `break` por lo que en todos los casos, al seleccionar uno de ellos se ejecutar la sentencia de la línea 10. Lo mismo ocurrirá si el ordinal del mes es 4, 6, 9 u 11, en todos los casos se ejecutará la sentencia de la línea 16.

2.5. Modificadores de acceso.

Ahora ya estamos en situación de volver a la pregunta: ¿Qué sentido tiene declarar métodos de acceso a los atributos de una clase si puedo acceder directamente a ellos? La repuesta es que, como regla general, nunca debemos hacer visibles los atributos de nuestras clases, sólo deben ser visibles desde el interior de las clases. Como resultado, para acceder a los valores de los atributos utilizaremos métodos. Esta regla es una manera de expresar el concepto de *Encapsulación*, una de las piezas centrales de la programación orientada a objetos.

Concepto

Las clases encapsulan atributos y métodos de tal modo que sólo se hace visible una parte de esos atributos y métodos, los estrictamente necesarios para que podamos trabajar con las instancias de esa clase.

La respuesta a la pregunta anterior hace surgir una nueva: ¿Cómo restrinjo la visibilidad de los atributos de una clase?, la respuesta es: mediante los *Modificadores de acceso.*

Definición

Los modificadores de acceso son palabras reservadas de Java mediante las cuales restringimos la visibilidad de los atributos y métodos de una clase.

En Java un modificador de acceso está representado por una palabra reservada que me permite definir la visibilidad de un atributo o un método de la clase. Los cuatro modificadores de acceso que podemos utilizar en Java son:

- `private`.
- `protected`.
- «Vacío» (no escribimos nada).
- `public`.

De momento, y hasta que veamos el capítulo de herencia, vamos a ver el significado de los dos más sencillos: `private` y `public`. Los modificadores de acceso se escriben antes del tipo del atributo o antes del tipo de retorno del método. Veamos cómo quedaría nuestra clase `Persona` asignando la visibilidad adecuada a cada uno miembros:

```
1 package agenda;
2
3 public class Persona {
4  String nombre;
5  String apellidos;
6  String telefono;
7
8  Persona() { }
9
10  Persona(String nombre, String apellidos, String telefono) {
11   this.nombre = nombre;
12   this.apellidos = apellidos;
13   this.telefono = telefono;
14  }
15
16  String getNombre() {
17   return nombre;
18  }
19
20  String getApellidos() {
21   return apellidos;
22  }
23
24  String getTelefono() {
25   return telefono;
26  }
27 }
```

En este caso estamos restringiendo la visibilidad de los atributos de la clase `Persona` de modo que únicamente son visibles desde el interior de la propia clase donde se han definido (modificador `private`). Por otro lado, estamos haciendo visibles los métodos de la clase a cualquier otra clase que los quiera utilizar (modificador `public`).

Buenas prácticas y convenciones

En general, se considera una buena práctica declarar los atributos de una clase como privados (`private`) y si necesitamos acceder a ellos para leer sus valores o modificarlos utilizaremos los métodos `get` o `set`. En caso de que el tipo del valor devuelto sea `boolean` se utilizará `is` en vez de `set`, por ejemplo `isNuevo()` en vez de `getNuevo()` si el valor que se retorna es un `boolean` (`true` o `false`).

Además de los modificadores que nos permiten definir la visibilidad de atributos y métodos de una clase, en Java existen otros modificadores que también se pueden aplicar sobre la definición de atributos y métodos: `static` y `final`.

2.6. Modificadores `static` y `final`.

Un atributo de una clase se puede modificar con la palabra reservada `static`, con ello indicamos que el atributo no pertenece a las instancias de la clase si no a la propia clase. ¿Qué quiere decir esto?, pues que no existe una copia de ese atributo en cada uno de los objetos de la clase, si no que existe una única copia que es compartida por todos los objetos de la clase. Por ello, a los atributos `static` se les llama atributos de la clase.

Una consecuencia de lo anterior es que para acceder a los atributos `static` de una clase no necesitamos crear una instancia de la clase, podemos acceder a ellos a través del nombre de la clase.

De igual modo, podemos modificar los métodos de una clase con la palabra reserva `static`. A estos métodos se les llama métodos de la clase, y, al igual que con los atributos `static`, podemos usarlos a través del nombre de la clase, sin necesidad de crear ninguna instancia de la clase. Pero existe una restricción, los métodos estáticos de una clase sólo pueden acceder a atributos estáticos u otros métodos estáticos de la clase, pero nunca a atributos o métodos que no lo sean. ¿Ves porqué? ¿Qué ocurriría si desde un método estático y usando el nombre de la clase intentases acceder a un atributo de instancia de la clase?

En el siguiente ejemplo de código hemos añadido un contador para saber el número de instancias de la clase `Persona` que se han creado:

```
1 package tipos;
2
3 public class Persona implements Contacto {
4  private String nombre;
5  private String apellidos;
6  private String telefono;
7  private static int nInstancias;
8
9  public Persona() {
10  super();
11  iniciaAtributos();
12 }
13
14 public static int getNInstancias() {
15  return nInstancias;
16 }
```

Fíjate que el método `getNInstancias()` que accede al atributo `nInstancias` es estático. En el siguiente ejemplo de código se está utilizando este método estático a través del nombre de la clase y a través de una instancia concreta:

```
1 public final class Principal {
2  private Principal() {
3   super();
4  }
5
6  private void ejecuta() {
7   Persona unaPersona = new Persona();
8   // Accedemos al método a través de la clase
9   System.out.println("Número de personas creadas: " + Persona.
        getNInstancias());
10  Persona otraPersona = new Persona("James", "Gossling", "555 123 456");
11  // Accedemos al método a través de una instancia concreta
```

Cuando un atributo de una clase los modificamos en su definición con la palabra reservada `final`, estamos indicando que ese atributo no puede cambiar de valor, por ejemplo:

```
1 private final String autor = "Óscar";
```

Una vez definido, este atributo no puede cambiar de valor, si lo intentásemos cambiar el compilador nos daría un error.

Muchas veces los modificadores `static` y `final` se utilizan en combinación para definir constantes, como en el siguiente ejemplo:

```
1 public class Constantes {
2  public static final double PI = 3.141592;
3  ...
4 }
```

De este modo, la constante es accesible desde cualquier otra clase (al ser `public`) y podemos leerla a través del nombre de la clase de este modo `Constantes.PI`, pero si por descuido intentamos modificarla, el compilador de Java nos dará un error.

Regla de convención

Los nombre de las constantes se escriben en mayúsculas.

El modificador `final` también se puede usar sobre un método o sobre la clase. Veremos con detalle lo que esto significa en el Capítulo 3 dedicado a la herencia en Java.

2.7. El recolector de basura.

Hemos visto que para crear instancias de una clase utilizamos el operador `new`. Cuando ya no necesitamos una instancia: ¿Cómo liberamos el espacio en memoria que está ocupando? En Java no existe ningún operador especial para eliminar de la memoria las instancias que no vamos a seguir utilizando. Para liberar la memoria existe un mecanismo mucho más potente, el *Recolector de basura*.

Como ya sabes, el modo de acceder a los objetos en Java es mediante las variables de tipo referencia. El recolector de basura conoce en todo momento todas las referencia que una instancia posee, y de igual modo conoce cuando una instancia ha perdido todas las referencias que apuntaban a ella. Si un objeto pierde

todas la referencias que apuntan a él y las referencias son el único mecanismo que tenemos de acceder a los objetos, significa que ya no podremos acceder a ese objeto, de modo que el recolector de basura puede hacer su trabajo: liberar la memoria ocupada por la instancia.

¿Cómo podemos marcar un objeto para que sea borrado de memoria? Una técnica sencilla es eliminar todas las referencias que apuntan a él como en el siguiente ejemplo:

```
1 Persona unaReferencia = new Persona(); // Esta Persona tiene una
      referencia hacia ella
2 Persona otraReferencia = unaReferencia; // Ahora tiene dos
3 unaReferencia = null; // Le desconectamos la primera referencia
4 otraReferencia = null; // Le desconectamos la segunda referencia
5 // El recolector de basura ya puede hacer su trabajo
```

2.8. Finalización.

En la sección anterior hemos visto cual es el mecanismo que utiliza Java para ir liberando de la memoria los objetos que ya no son accesibles. Todos los objetos poseen un método con la siguiente signatura `protected void finalize() throws Throwable`, en los capítulos siguientes veremos con detalle el significado de las palabras reservadas `protected` y `throws`, así como la clase `Throwable`, lo que nos interesa en este momento es saber que este es el último método de cualquier objeto que se llama antes de que el recolector de basura elimine la instancia de la memoria. Dicho de otro modo, el método `finalize` es la última oportunidad que tenemos como programadores para que nuestras instancias acaben limpiamente. Veamos qué quiere decir esto con más detalle.

Supón que has escrito un clase que abre un fichero para lectura (veremos acceso a ficheros en el capítulo 7), y que por cualquier motivo una instancia de esta clase pierde todas las referencias que apuntan a ella. Cuando actuase el recolector de basura, eliminaría esta instancia de la memoria y el resultado colateral sería que el fichero quedaría abierto. Para que esto no ocurra, en el método `finalize` podemos escribir el código que cierre los ficheros que están abiertos, ya que sabemos que este método será llamado antes de eliminar la instancia de memoria.

Pero no es tan inmediato, el problema que conlleva delegar al método `finalize` estas tareas de limpieza segura antes de acabar es que no sabemos cuando el recolector de basura va a hacer su trabajo, sabemos que lo hará, pero no sabemos cuando. Y aunque podemos «forzar» la actuación del recolector de basura de este modo:

```
1 Runtime r = Runtime.getRuntime();
2 r.gc(); // Solicitamos que el recolector de basura entre en acción.
```

no se garantiza que el recolector de basura vaya a ser invocado inmediatamente.

Luego, como norma general:

Buenas prácticas

No debemos delegar en el recolector de basura la limpieza que han de realizar nuestras clases cuando sus instancias son eliminadas de memoria.

2.9. Comentarios. Comentarios de documentación.

Todos los programadores son conscientes de la importancia de documentar su trabajo. Una tarea de documentación es incluir comentarios en el propio código para que otros programadores puedan conocer en el momento de la lectura de código los detalles de implementación.

Para realizar tareas de documentación Java nos proporciona tres tipos de comentarios:

1. Comentarios de una única línea.
2. Comentarios de más de una línea.
3. Comentarios de documentación.

Los comentarios de una única línea empiezan con
y el texto del comentario restringe su extensión a una única línea.

Los comentarios de más de una línea empiezan con `/*`, el texto del comentario puede ocupar cuantas líneas necesitamos, pero es necesario indicar que el comentario acaba insertando al final `*/`. En el siguiente ejemplo puedes ver cómo se usan ambos tipos de comentario:

```
1 public class Persona {
2  //esto es un comentario de una única línea
3  private String nombre;
4  /* Este comentario ocupa
5   más de una línea
6   de código */
7  private String apellidos;
```

Pero, sin duda, los comentarios de documentación son una herramienta realmente potente en Java. Los comentarios de documentación se incluyen en el código y nos sirven para, a partir de ellos, crear documentación de nuestro código en formato html. En los comentarios de documentación podemos añadir etiquetas que nos permiten enriquecer la documentación generada. Veamos cómo se introducen los comentarios de documentación y las etiquetas que tenemos disponibles.

Un comentario de documentación siempre debe empezar por `/**`, nota que tras la barra se escriben dos asteriscos, y debe acabar por `*/`, como los comentarios de más de una línea.

Dentro de los comentarios de documentación podemos utilizar etiquetas para añadir información que enriquezca la documentación generada. Por ejemplo, podemos utilizar la etiqueta para indicar quien es el autor del código de una clase, como en el siguiente ejemplo:

```
1 /** Implementación de la clase Persona
```

```
 * Esta clase describe a un nuevo contacto
 * en una agenda de teléfonos
 * @author Óscar Belmonte Fernández
 * @version 1.0
*/

public class Persona {
 private String nombre;
```

otros comentarios de documentación:

- @version Indicamos la versión del código.
- @param nombre Descripción del parámetro.
- @return Significado del valor de retorno.
- @deprecated Razón de por qué este método está obsoleto.
- @see #metodo() Referencia cruzada al método.
- @exception Excepción que se puede producir en el método
- @throws Excepción no gestionada

Además, en los comentarios de documentación podemos incluir código HTML. En el listado 2.9 tienes la clase `Persona` documentada.

```
package persona.comentarios;

/** Implementación de la clase Persona
 * Esta clase describe a un nuevo contacto
 * en una agenda de teléfonos
 * @author Óscar Belmonte Fernández
 * @version 1.0
*/

public class Persona {
 private String nombre;
 private String apellidos;
 private String telefono;
 private static int nInstancias = 0;

 /**
  * Constructor por defecto
  */
 public Persona() {
  nInstancias++;
 }

 /**
  * Constructor con parámetros.
  * En nuevas versiones, tanto el nombre como los apellidos serán
  * inmutables, no existirán métodos para camobiarlos
  * @param nombre Nombre del nuevo contacto
  * @param apellidos Apellidos del nuevo contacto
  * @param telefono Teléfono del nuevo contacto
  */
 public Persona(String nombre, String apellidos, String telefono) {
  this.nombre = nombre;
  this.apellidos = apellidos;
  this.telefono = telefono;
  nInstancias++;
 }

 /**
  * Devuelve el número de instancias creadas
  * @return El número de instancias
```

```
41   */
42  public static int getNInstancias() {
43   return nInstancias;
44  }
45
46  /**
47   * Devuelve el nombre del contacto
48   * @return Nombre del contacto
49   */
50  public String getNombre() {
51   return nombre;
52  }
53
54  /**
55   * Devuelve los apellidos del contacot
56   * @return Apellidos del contacto
57   */
58  public String getApellidos() {
59   return apellidos;
60  }
61
62  /**
63   * Devuelve el número de teléfono del contacto
64   * @return Número de teléfono del contacto
65   */
66  public String getTelefono() {
67   return telefono;
68  }
69
70  /**
71   * Cambia el nombre del contacto
72   * @param nombre El nuevo nombre del contacto
73   * @deprecated Este método se eliminará en versiones futuras
74   * @see Persona(String nombre, String apellidos, String telefono)
75   */
76  public void setNombre(String nombre) {
77   this.nombre = nombre;
78  }
79
80  /**
81   * Cambia los apellidos del contacto
82   * @param apellidos Los nuevos apellidos del contacto
83   * @deprecated Este método se eliminará en versiones futuras
84   * @see #Persona(String nombre, String apellidos, String telefono)
85   */
86  public void setApellidos(String apellidos) {
87   this.apellidos = apellidos;
88  }
89
90  /**
91   * Cambia el número de teléfono del contacto
92   * @param telefono El nuevo número de teléfono del contacto
93   */
94  public void setTelefono(String telefono) {
95   this.telefono = telefono;
96  }
97 }
```

Listado 2.9: Código fuente de la clase Persona con comentarios de documentación.

El paquete de desarrollo Java nos proporcionan una herramienta para generar la documentación de nuestras clases en formato HTML a partir del código. Esta herramienta se llama `javadoc`. La generación de código se puede realizar desde consola de este modo:

```
javadoc Persona.java /ruta/a/directorio
```

Si no se especifica la ruta la documentación se generará en el directorio donde se encuentre el fichero `Persona.java`.

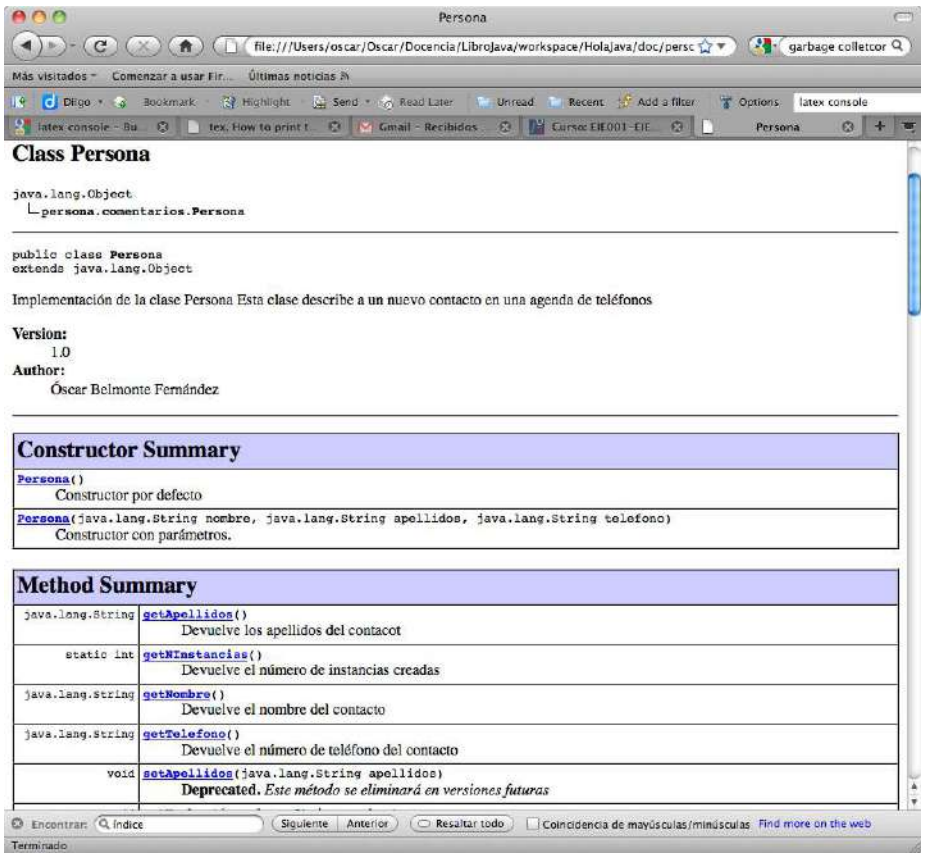

Figura 2.1: Comentarios de documentación generados con la herramienta `javadoc`.

Desde Eclipse también podemos generar la documentación de nuestras clases haciendo click con el botón derecho del ratón sobre el fichero de la clase y después seleccionamos *Export* → *Java Javadoc*. Se generará toda la documentación de nuestra clase, o todo el proyecto si lo hemos seleccionado en vez de una clase individual. En la figura 2.1 se muestra cual es el aspecto de la documentación generada cuando se utiliza un navegador web para su visualización.

Ejercicios.

1. Escribe una clase que abstraiga la idea de *Empresa*. Una empresa puede tener únicamente como atributos su nombre y un teléfono de contacto.

2. Añade comentarios de documentación a la clase *Empresa* y genera la documentación de esta clase.

3. Escribe un sencillo programa para gestionar los contactos de una agenda telefónica (clase `Agenda`). Las operaciones básicas que la agenda telefónica es capaz de realizar son:

a) Insertar nuevas `Persona`s en la agenda.

b) Listar todas las `Persona`s de la agenda.

c) Buscar una `Persona` a partir de su nombre.

Lecturas recomendadas.

- El excelente libro de Arnold y otros [2] es una referencia completa a la definición de clases. Comenta de modo exhaustivo todos los detalles en la definición de una clase, los distintos modificadores de acceso y su significado.
- El modo de presentar los conceptos del lenguaje Java ne la serie de libros de la colección *Head first Java* es muy interesante. En particular, la referencia [3] presenta de una manera muy visual los principales conceptos en la definición de clases en Java.
- Finalmente, una lectura siempre recomendable sobre buenas prácticas y escritura de código limpia es el libro de Robert C. Martin de la referencia [10].

Capítulo 3

Herencia e Interfaces

Contenidos

3.1. Herencia. **52**
3.2. Extensión de una clase. **52**
3.2.1. Sobrescribir atributos. 54
3.2.2. Sobrescribir métodos. 56
3.2.3. La palabra reservada `super`. 59
3.2.4. El constructor por defecto y la clase `Object`. 59
3.2.5. El operador `instanceof`. 60
3.2.6. El modificador `final`. 61
3.2.7. Métodos `static`. 62
3.3. Clases abstractas. **63**
3.4. Interfaces. **65**
3.5. Enumeraciones. **68**
3.6. Paquetes en Java. **69**
3.7. Clases e `interface` anidados **71**

Introducción

En el capítulo 2 hemos visto cómo podemos codificar la abstracción de una entidad del mundo real a código Java. Hemos visto la sintaxis de Java para la creación de clases, atributos y métodos, además de algunos de los modificadores que podemos utilizar sobre ellos. También hemos aprendido a crean instancias a partir de una clase con el operador `new`.

La situación en la que nos encontramos ahora es que nuestras clases abstraen entidades del mundo real pero, ¿Qué ocurre si quiero añadir más funcionalidad a mis clases?, ¿Cómo puedo añadir nuevos atributos a una clase ya existente?, ¿Cómo puedo ampliar el conjunto de métodos que proporciona?

En POO existe un mecanismo fundamental para añadir funcionalidad a una clase el concepto es la *Herencia*, también conocido como *Extensión* o *Derivación*.

La idea básica de la *Herencia* es crear una nueva clase a partir de la definición de otra clase ya existente. La *Herencia* nos permite construir una clase añadien-

do únicamente la nueva funcionalidad a otra clase ya existente. Y también nos permite modificar el comportamiento de la clase original *sobrescribiendo* sus métodos.

3.1. Herencia.

En esta sección vamos a ver cómo se puede ampliar el comportamiento de una clase a través de la herencia. Veremos también el concepto, muy importante, de la vinculación dinámica para encontrar qué método se ha de invocar al utilizar referencias a clases extendidas, así como el uso del operador `instanceof`. Finalmente veremos el significado de los modificadores *final* y *abstract* cuando se aplican a la definición de un método o una clase.

3.2. Extensión de una clase.

Lo primero que hay que destacar es que en Java sólo está permitida la herencia simple, una nueva clase sólo puede extender a una única clase base. Dicho de otro modo, una clase hija no puede tener más de una clase padre.

Característica

Java sólo admite herencia simple. Una clase no puede tener más de una clase padre.

Imaginemos que necesitamos ampliar nuestra clase `Persona`, para añadirle nueva funcionalidad. Queremos que nuestra nueva clase contenga la provincia, la población de residencia y la edad [1] de cada uno de nuestros contactos. Para ello tenemos dos alternativas antagónicas:

1. Reescribir la clase desde cero.

2. Aprovechar al máximo el código existente.

Si optamos por la primera opción, no tenemos nada nuevo que aprender, con lo aprendido hasta ahora podemos resolverlo.

Si optamos por la segunda opción estaremos haciendo uso del mecanismo de *Herencia*. Veamos como se utiliza este mecanismo y cual es su sintaxis en Java.

En Java se dice que una clase extiende a otra clase cuando añade más funcionalidad a una clase ya existente. A la nueva clase se le llama clase hija o extendida, a la clase original se le llama clase padre o base. Para indicar que una clase extiende el comportamiento de otra utilizamos la palabra reservada `extends`.

Supongamos que queremos ampliar la definición de una `Persona` para que contenga datos de su lugar de residencia, como son la `Provincia` y la *Población* y también la *Edad*, y llamemos a esta nueva clase `Ciudadano`. en Java lo hacemos de este modo:

[1] Cuando veamos las clases para manipular fechas en el Capítulo 8 veremos una mejor implementación para obtener la edad de una persona a partir de su fecha de nacimiento

```
1 public class Ciudadano extends Persona {
2  // Definición de la nueva clase extendida
3 }
```

En la definición de la nueva clase podemos incluir nuevos atributos y métodos. En el siguiente código de ejemplo, a la clase `Ciudadano` se le han añadido los tres nuevos atributos antes mencionados y los *getters* y *setters* para estos nuevos atributos. Nuestra nueva clase `Ciudadano` posee tanto los nuevos métodos definidos en ella como los métodos definidos en su clase padre (con las restricciones de accesibilidad que veremos en la sección 3.6.

En el Listado 3.1 aparece la definición completa de la nueva clase `Ciudadano`.

```
1 package tipos;
2
3 public class Ciudadano extends Persona {
4  private String poblacion;
5  private String provincia;
6  private int edad;
7
8  public Ciudadano() {
9   super();
10   iniciaAtributos();
11  }
12
13  @Override
14  protected void iniciaAtributos () {
15   setNombre("Un nombre");
16   edad = 0;
17  }
18
19  public String getPoblacion() {
20   return poblacion;
21  }
22
23  public void setPoblacion(String poblacion) {
24   this.poblacion = poblacion;
25  }
26
27  public String getProvincia() {
28   return provincia;
29  }
30
31  public void setProvincia(String provincia) {
32   this.provincia = provincia;
33  }
34
35  public int getEdad() {
36   return edad;
37  }
38
39  public void setEdad(int edad) {
40   this.edad = edad;
41  }
42 }
```

Listado 3.1: Definición de la clase `Ciudadano`

¿Cómo hacemos uso de los métodos de la clase, tanto de los definidos en la clase extendida como los definidos en la clase base?, sencillamente como lo estábamos haciendo hasta ahora: a través de las referencias, como en el siguiente ejemplo de código:

```
1 Ciudadano ciudadano = new Ciudadano("José", "García", "555 123 456", "
      Alcorcón", "Madrid", 40;
2 System.out.println("Nombre: " + ciudadano.getNombre());
```

```
3 System.out.println("Edad: " + ciudadano.getEdad());
```

Como vemos en la línea 2 del Listado anterior, hacemos uso del método `getNombre()` definido en la clase padre, a partir de una referencia de la clase hija, mientras que en la línea 3 hacemos uso del método `getEdad()` definido en la clase hija.

¿Podemos utilizar una referencia a la clase padre para acceder a los mismos métodos? No, aunque es perfectamente válido asignar una referencia de una clase hija a una referencia a la clase padre, a través de la referencia a la clase padre sólo tendremos acceso a los miembros declarados en ella. En particular, para el ejemplo de la clase padre `Persona` y su clase hija `Ciudadano`, el siguiente código ejemplo contiene un error:

```
1 Ciudadano ciudadano = new Ciudadano();
2 Pesona persona = ciudadano; // Perfectamente válido.
3 persona.getNombre(); // No hay problema, getNombre() está definido en
      Persona.
4 persona.getEdad(); // Error!!!, getEdad() está definido en Ciudadano.
```

También es un error asignar a una referencia de una clase hija una referencia a la clase padre, el siguiente código de ejemplo contiene un error:

```
1 Persona persona = new Persona();
2 Ciudadano ciudadano = persona; // Error!!!
```

Concepto clave

Una referencia de una clase padre admite una referencia a cualquiera de sus clase hijas, pero nunca al contrario.

Piensa qué ocurriría si no existiese esta prohibición, podríamos asignar a una referencia a `Ciudadano` una referencia de su clase padre `Persona`, y a través de la referencia a `Ciudadano` podríamos invocar a, por ejemplo, el método `getEdad()`, pero, la clase `Persona` no posee el atributo `int edad;`, ¿Qué se debería devolver en este caso?

3.2.1. Sobrescribir atributos.

En algunas circunstancias, podemos vernos en la necesidad de definir un atributo en una clase hija con el mismo nombre que en su clase padre, como muestra el siguiente código de ejemplo:

```
1 // Esta es la clase padre
2 public class Distancia {
3  float distancia;
4
5  public Distancia() {
6   distancia = 0;
7  }
8
9  public Distancia(float distancia) {
10  this.distancia = distancia;
11 }
12 // Sigue la definición de esta clase.
```

```
13 }
14
15 // Esta es la clase hija
16 public class DistanciaDoblePrecision extends Distancia {
17  // Este es el atributo sobrescrito
18  double distancia;
19
20  public DistanciaDoblePrecision() {
21   distancia = 0;
22  }
23
24  public DistanciaDoblePrecision(double distancia) {
25   this.distancia = distancia;
26  }
27  // Sigue la definición de esta clase.
28 }
```

En este caso se dice que el atributo `distancia` de la clase hija `DistanciaDoblePrecision` sobrescribe el atributo `distancia` de la clase padre `Distancia`. Cuando una clase hija sobrescribe algún atributo de su clase padre, el atributo de la clase padre queda *oculto* , de modo que si aparece el nombre del atributo en la clase hija se utilizará el atributo definido en esta clase y no el definido en la clase padre. Esto no quiere decir que el atributo con el mismo nombre en la clase padre desaparezca, sino que para acceder a él tendremos que hacer uso de otro mecanismo como veremos más adelante en esta sección.

¿Cómo accedemos al atributo distancia desde fuera de la clase? Ya lo sabemos, a través de referencias. De acuerdo, entonces, ¿Qué mostrará el siguiente ejemplo?:

```
1 Distancia distancia = new Distancia(100);
2 System.out.println("El valor de distancia es: " + distancia.distancia);
3 Distancia distanciaDoblePrecision = new DistanciaDoblePrecision(200);
4 System.out.println("El valor de distanciaDoblePrecision es: " +
      distanciaDoblePrecision.distancia);
```

Lo que mostrará este código es, sorpresa:

```
El valor de distancia es: 100.0
El valor de distancia2 es: 0.0
```

¿Qué ha ocurrido? Nada extraño, simplemente que al acceder al atributo a través de la referencia, se ha buscado este valor en la definición de la clase correspondiente a la referencia, que en los dos casos es `Distancia` y el atributo que se está iniciando en la línea 3 del código anterior es el de la clase hija `DistanciaDoblePrecision` pues el objeto que se crea es de la clase extendida. Comparemos con el resultado de la ejecución de este otro código ejemplo:

```
1 Distancia distancia = new Distancia(100);
2 System.out.println("El valor de distancia es: " + distancia.distancia);
3 DistanciaDoblePrecision distanciaDoblePrecision = new
      DistanciaDoblePrecision(200);
4 System.out.println("El valor de distanciaDoblePrecision es: " +
      distanciaDoblePrecision.distancia);
```

Lo que mostrará este código es:

```
El valor de distancia es: 100.0
El valor de distanciaDoblePrecision es: 200.0
```

En este último ejemplo, lo único que ha cambiado es el tipo de la referencia `distanciaDoblePrecision`, que en este caso es de tipo `DistanciaDoblePrecision`, es decir, la clase hija.

Concepto clave

Cuando una clase hija sobrescribe (oculta) un atributo de la clase padre, el atributo seleccionado se determina por el tipo de la referencia.

3.2.2. Sobrescribir métodos.

Para introducir el modo de sobrescribir métodos imaginemos que hemos añadido al código de la clase `Distancia` un nuevo método que nos permita incrementar la distancia actual:

```
1 public class Distancia {
2  float distancia;
3
4  public Distancia() {
5   distancia = 0;
6  }
7
8  public Distancia(float distancia) {
9   this.distancia = distancia;
10 }
11
12 void incrementaDistancia(float incremento) {
13  distancia += incremento;
14 }
15 // Sigue la definición de esta clase.
16 }
```

Listado 3.2: Definición de la clase `Distancia`

Ahora queremos probar nuestra nueva funcionalidad con este ejemplo:

```
1 DistanciaDoblePrecision distancia = new DistanciaDoblePrecision(100);
2 distancia.incrementaDistancia(100);
3 System.out.println("El valor de distancia es: " + distancia.distancia);
```

El resultado que obtenemos es el siguiente:

```
El valor de distancia es: 100.0
```

¿Cómo es posible? Estamos intentando incrementar la distancia inicial de 100 en otros 100, y parece que no lo conseguimos. ¿Donde está el problema?.

Nuestro nuevo método `incrementaDistancia(float)` está definido en la clase padre, y este método incrementa el valor del atributo que hay definido en ella, no en la clase hija. ¿Cómo podemos arreglarlo? La respuesta es sobrescribiendo el método `incrementaDistancia(float)` en la clase hija de este modo:

```
1 public class DistanciaDoblePrecision extends Distancia {
2  double distancia;
3
4  public DistanciaDoblePrecision() {
5   distancia = 0;
6  }
7
```

```
 8  public DistanciaDoblePrecision(double distancia) {
 9   this.distancia = distancia;
10  }
11
12  @Override
13  void incrementaDistancia(float incremento) {
14   distancia += incremento;
15  }
16  // Sigue la definición de esta clase.
17 }
```

Listado 3.3: Definición de la clase `DistanciaDoblePrecisiona`

Nótese el uso de la anotación `@Override` que indica al compilador de Java que se está intentando sobrescribir un método en la clase padre.

Ahora sí, el resultado obtenido es:

```
El valor de distancia es: 200.0
```

Para que una clase hija sobrescriba un método de su clase padre es necesario que ambos métodos tengan la misma signatura y el mismo tipo de retorno, de lo contrario no se sobrescribe el método, como en el siguiente ejemplo:

```
1 public class DistanciaDoblePrecision extends Distancia {
2  double distancia;
3
4  public DistanciaDoblePrecision() {
5   distancia = 0;
6  }
7
8  public DistanciaDoblePrecision(double distancia) {
9   this.distancia = distancia;
10  }
11
12  // Intentamos sobrescribir cambiando el tipo del argumento
13  // Se produce un error
14  @Override
15  void incrementaDistancia(double incremento) {
16   distancia += incremento;
17  }
18  // Sigue la definición de esta clase.
19 }
```

Aunque es posible que la clase hija sobrescriba un método de la clase padre ampliando el modificador de acceso, por ejemplo, en el caso del Listado 3.2.2 posemos definir el método `incrementaDistancia` como `public void incrementaDistancia(double incremento)`, de modo que hemos ampliado el modificador de acceso desde acceso paquete hasta `public`. Lo que no está permitido es que cuando se sobrescribe un método de la clase padre, el hijo restrinja el modificador de acceso. Siguiendo con el ejemplo, intentar sobrescribir el método `incrementaDistancia` como `private void incrementaDistancia(double incremento)`, daría un error.

Gracias al uso de la anotación `@Override`[2] obtenemos un error que nos informa que el nuevo método no está sobrescribiendo a ningún método en su clase padre. Si eliminamos la anotación no obtenemos ningún error y lo que estaremos haciendo es definiendo un nuevo método que toma como argumento una variable de tipo `double`.

[2]Las anotaciones fueron introducidas en la versión 5 de Java

Buenas prácticas

Para asegurar que los métodos en las clases hijas sobrescriben métodos en la clase padre utilícese la anotación `@Override` en el método sobrescrito.

Ya sabemos que a una referencia a una clase padre le podemos asignar una referencia a cualquier clase hija, de acuerdo, modifiquemos nuestro código de prueba del siguiente modo:

```
1 Distancia distancia = new DistanciaDoblePrecision(100);
2 distancia.incrementaDistancia(100);
3 System.out.println("El valor de distancia es: " + distancia.distancia);
```

¿Qué es lo que obtenemos? Sorpresa de nuevo:

```
El valor de distancia es: 0.0
```

¿Cómo puede ser que obtengamos 0.0 si estamos creando un objeto de tipo `DistanciaDoblePrecision` con un valor inicial de 100 y después lo estamos incrementando en 100 unidades más?. La respuesta, esta vez está recogida en este concepto clave:

Concepto clave

Cuando accedemos a los métodos de un objeto a través de una referencia se selecciona el método a partir del tipo del objeto y no de la referencia a través de la que se accede.

Este concepto es muy importante en POO y a este mecanismo se le llama *Vinculación dinámica* .

¿Qué está ocurriendo entonces? Ocurre que distancia es una referencia de tipo `Distancia`, pero el tipo del objeto al que hace referencia, el que creamos con el operador `new` es `DistanciaDoblePrecision`. Al usar el método `incrementaDistancia(100)` la vinculación dinámica ejecuta el código de `DistanciaDoblePrecision` no el de `Distancia`. Mientras que tal y como sabemos de la Sección 3.2.1, si utilizamos atributos no se hace uso de la vinculación dinámica y se accede al atributo correspondiente al tipo que indica la referencia no el objeto que hay por debajo de ella, por lo tanto si escribimos `distancia.distancia` estamos accediendo al atributo en `Distancia` pero el atributo que se incrementó con `distancia.incrementaDistancia(100)` fue el que incrementó la vinculación dinámica, es decir, el de `DistanciaDoblePrecision`.

Nótese la diferencia fundamental con respecto al acceso a los atributos, donde el atributo al que se accede se determina por el tipo de la referencia y no del objeto.

También es posible que una clase sobrescriba un método ampliando el tipo del valor de retorno, es decir que si en la clase padre `Distancia` tenemos un método como Distancia metodo() la clase hija puede sobrescribirlo con el método DistanciaDoblePrecision metodo(), ya que se ha ampliado el tipo del valor de retorno

desde el original `Distancia` a `DistanciaDoblePrecision`, esta posibilidad fue introducida en la versión 5 de Java.

Pero cuidado, esto último no funciona con los tipos de retorno primitivos, si tenemos en la clase padre un método definido como `public float unMetodo()` que devuelve el tipo primitivo `float` no lo podemos sobrescribir en una clase hija con `public double unMetodo()` que devuelve el tipo primitivo `double`.

3.2.3. La palabra reservada `super`.

Existen casos en los que, desde una clase hija, nos interesa acceder a los métodos o atributos sobrescritos en la clase padre. Si escribimos en la clase hija simplemente el nombre del atributo o del método estaremos haciendo uso de la definición dada para ellos en la clase hija. ¿Cómo accedemos a los miembros sobrescritos desde la clase hija?. La respuesta es haciendo uso de la palabra reservada `super`.

Definición

La palabra reservada `super` es una referencia a la clase padre, del mismo modo que la palabra reservada `this` es una referencia a la propia clase.

3.2.4. El constructor por defecto y la clase `Object`.

En el código de los ejemplos de prueba, nunca hemos utilizado el constructor sin parámetros de las clases `Distancia` (véase el listado 3.2) ni de la clase `DistanciaDoblePrecision` (véase el listado 3.3), luego, podemos intentar eliminar estos dos constructores e iniciar el atributo `distancia` de ambas clases en el momento de la definición, tal y como se muestra en el siguiente listado:

```
1 //---- Definición de la clase Distancia ----//
2 public class Distancia {
3  float distancia = 0;
4
5  // Eliminado el constructor sin parámetros
6
7  public Distancia(float distancia) {
8   this.distancia = distancia;
9  }
10
11 void incrementaDistancia(float incremento) {
12  distancia += incremento;
13 }
14 // Sigue la definición de esta clase.
15 }
16 //---- Definición de la clase DistanciaDoblePrecision ----//
17 public class DistanciaDoblePrecision extends Distancia {
18  double distancia = 0;
19
20  // Eliminado el constructor sin parámetros
21
22  public DistanciaDoblePrecision(double distancia) {
23   this.distancia = distancia;
24  }
25
26  @Override
27  void incrementaDistancia(float incremento) {
28   distancia += incremento;
29  }
30  // Sigue la definición de esta clase.
```

Pero inmediatamente obtendremos el siguiente error en el código de la clase `DistanciaDoblePrecision` *Implicit super constructor Distancia() is undefined. Must explicitly invoke another constructor.* Este error es debido a que, el constructor de la clase hija `public DistanciaDoblePrecision(double distancia)` está intentando invocar implícitamente al constructor de la clase padre `public Distancia()` que no está definido. Este es el mecanismo en la creación de objetos en Java cuando existe relación de herencia entre clases, desde los constructores de las clases hijas, si no se indica lo contrario, se intenta invocar al constructor sin parámetros de la clase padre, que por este motivo es llamado *Constructor por defecto* . Si no se indica lo contrario, lo primero que se hace desde el constructor de una clase hija es llamar al constructor por defecto de la clase padre.

Buenas prácticas

Para evitar problemas en la ceración de objetos, es conveniente definir siempre el constructor por defecto en nuestras clases.

El error anterior lo podemos corregir de dos modos, añadiendo los constructores por defecto a cada una de las clases, o bien, llamando desde el constructor con parámetros de la clase hija al constructor con parámetros de la clase padre, para que no se llame *por defecto* el constructor sin parámetros, que no está definido:

```
1 public DistanciaDoblePrecision(float distancia) {
2  super(0); // Llamamos al constructor con parámetros del padre
3  this.distancia = distancia;
4 }
```

Si optamos por la segunda solución, la llamada al constructor del padre es lo primero que debemos hacer en el construtor del hijo; en el ejemplo anterior si intercambiamos las líneas 3 y 4 obtendremos el siguiente error *Constructor call must be the first statement in a constructor*

La pregunta que nos surge es ¿A qué constructor se llama desde el constructor por defecto de la clase `Distancia` que no está extendiendo a ninguna otra clase, tal y como se muestra en el Listado 3.2? Para responder a esta pregunta necesitamos saber que la clase `Object` es la clase que está en la raíz del árbol de jerarquí de clases en Java, y que si una clase explícitamente no extiende a ninguna otra, implícitamente está extendiendo a la clase `Object`.

Concepto clave

La clase `Object` se encuentra en la raíz del árbol de jerarquía de clases en Java. Cualquier otra clase, bien directamente o bien a través de herencia, es hija de la clase `Object`.

3.2.5. El operador `instanceof`.

Ya sabemos que cuando llamamos a los métodos de un objeto a través de una referencia, es el tipo del objeto (la clase a la que pertenece) el que determina qué método se ha de llamar. A este mecanismo lo hemos llamado *Vinculación*

dinámica. No importa el tipo de la referencia a la que asignemos el objeto siempre que, evidentemente, el tipo de la referencia sea compatible con el tipo del objeto, en tiempo de ejecución el mecanismo de vinculación dinámica determinará cual es el método que se ha de llamar si es que ese método está sobrescrito.

La pregunta que ahora nos surge es: Si el único acceso que tenemos es a través de referencias y el tipo de la referencia no tiene por qué coincidir con el tipo del objeto que tiene asignado, basta con que sean compatibles, ¿Cómo podemos conocer el verdadero tipo del objeto asignado a la referencia?.

Para dar contestación a esta pregunta Java pone a nuestra disposición un operador binario, el operador `instanceof` con el que podemos preguntar si el objeto asignado a una referencia es de un determinado tipo o no; el valor de retorno de este operador es un booleano *true* o *false*. Estos son algunos casos de uso:

```
1 Persona persona = new Persona();
2 System.out.println(persona instanceof Persona); // Devolverá \emph{true}
3 System.out.println(persona instanceof Ciudadano); // Devolverá \emph{
      false}
4 Ciudadano ciudadano = new Ciudadano();
5 System.out.println(ciudadano instanceof Ciudadano); // Devolverá \emph{
      true}
6 System.out.println(ciudadano instanceof Persona); // Devolverá \emph{
      true}
```

Aunque el operador `instanceof` nos puede prestar ayuda en algunos casos, conviene seguir la siguiente buena práctica:

Buenas prácticas

Intenta evitar el uso del operador `instanceof` en tu código, utiliza polimorfismo para no hacer uso de este operator.

3.2.6. El modificador `final`.

En el capítulo 2 vimos el uso del modificador `final` aplicado a los atributos de una clase.

El operador `final` también se puede aplicar a los métodos de una clase, de tal modo que si un método se declara como `final` estamos indicando que ese método no puede ser sobrescrito por ninguna clase hija. Con ello estamos garantizando que el trabajo que realiza el método es siempre el mismo con independencia de si el objeto sobre el que se llama es instancia de la clase padre o instancia de alguna de sus clases hijas. En el siguiente listado se muestra cómo se puede violar el comportamiento al iniciar una instancia por parte de un hijo si el padre no protege el método que inicia los atributos con el modificado `final` :

```
1 // Código de la clase padre
2 public class Persona {
3  private String nombre;
4  private String apellidos;
5  private String telefono;
6  private static int nInstancias;
7
8  public Persona() {
```

```
 9   super();
10   iniciaAtributos();
11  }
12
13  protected void iniciaAtributos() {
14   nombre = "";
15   apellidos = "";
16   telefono = "";
17   nInstancias = 0;
18  }
19  // Sigue la definición de la clase
20
21 // Código de la clase hija
22  public Ciudadano() {
23   super();
24   // Aquí cambiamos el comportamiento de inicio de los atributos
25   iniciaAtributos();
26  }
27
28  @Override
29  protected void iniciaAtributos () {
30   setNombre("Un nombre");
31  }
32  // Sigue la definición de la clase
```

Simplemente añadiendo el modificador `final` al método `iniciaAtributos` de la clase padre, si intentamos sobrescribir este método en la clase hija obtendremos el siguiente error *Cannot override the final method from Persona* advirtiéndonos que no podemos sobrescribir un método declarado como `final` en la clase padre.

Buenas prácticas

Los métodos a los que se llama desde los constructores de una clase deben ser modificados como `final` para prevenir que alguna clase hija modifique el comportamiento al crear instancias de la clase.

Es muy recomendable seguir la anterior buena práctica, piensa que ocurriría si en el constructor de una clase padre que abre una conexión a una base de datos, y una clase hija sobrescribiese las tareas de inicio, y la conexión a la base de datos no se estableciese; toda la aplicación dejaría de funcionar.

El modificador `final` también se puede aplicar sobre una clase de este modo:

```
1 public final class Persona {
2  // La definición de la clase
```

En este caso lo que estamos indicando es que la clase `Persona` no se puede extender porque la hemos declarado como `final`. Un ejemplo de clase `final` en Java es la clase `String` que está declarada como `final` y por lo tanto no se puede extender, es decir no podemos crear hijas de ella.

3.2.7. Métodos `static`.

En el Capítulo 2 vimos el significado del modificador `static` cuando se aplica a métodos. El modificador `static` indica que el método pertenece a la clase y no a las instancias de la clase. Por otro lado hemos visto lo que significa la *Vinculación dinámica*, determinar en tiempo de ejecución el método que se debe llamar al invocarse desde una instancia cuando está sobrescrito. Fíjate

que el mecanismo de sobrescribir métodos funciona en el ámbito de los objetos, mientras que los métodos `static` pertenecen al ámbito de las clases. Es por esto que un método `static` de una clase padre no se puede sobrescribir, los métodos `static` de la clase padre son métodos *ocultos* que no son visibles desde las clases hija. Si en una clase hija declaramos un método con la misma signatura que un método `static` en la clase padre, lo que estamos haciendo realmente es creando un nuevo método en la clase hija sin ninguna relación con el mismo método en la clase padre. Obviamente si intentamos usar la anotación `@Override` para indicar que queremos sobrescribir el método obtendremos un error *This instance method cannot override the static method from ...*

3.3. Clases abstractas.

Hasta este momento, siempre que hemos definido los métodos de las clases que hemos creado, siempre hemos escrito código en la definición de los métodos. A veces es útil simplemente declarar métodos un una clase padre, sin dar ninguna implementación para ellos, y delegar la implementación a las clases hijas que la extiendan. Esta es una técnica muy potente que utiliza el concepto de *Polimorfismo* de la POO. De este modo estamos garantizando que todas las clases hijas de la misma clase padre tiene un método con la misma signatura, aunque, obviamente, cada una de las clase hijas puede tener una implementación distinta para el método polimórfico.

Si queremos indicar que no vamos a dar una implementeación para algún método declarado en la clase, debemos modificarlo con la palabra reservada `abstract`, con la restricción de que si una clase tiene algún método `abstract` ella misma también debe ser declarada como `abstract`.

También podemos declarar una clase como `abstract` sin que ninguno de sus métodos los sea. Si una clase es declarada como `abstract`, sobre ella tenemos la restricción recogida en el siguiente concepto clave:

Concepto clave

No se pueden crear instancias de una clase declarada como `abstract`

De no existir esta restricción ¿Qué ocurriría si se llamase a un método `abstract` de un objeto? ¿Qué código se ejecutaría? Evidentemente de poder ser así tendríamos un grave problema, ya que puede que no existiene ningún código para ejecutar.

Los métodos `abstract` de la clase padre deben ser definidos en las clases hijas, en cuyo caso los métodos en las clase hijas ya no serán `abstract` y tampoco la propia clase hija. Ahora bien, puede existir algún el caso en que una clase hija tampoco defina algún método `abstract` de la clase padre; en este caso la clase hija también deberá ser declarada `abstract` y no podremos crear instancias de ella.

Un ejemplo recurrente para mostrar el uso de las clases `abstract` es una aplicación que dibuje distintas figuras geométricas tales como círculos, triángulos y cuadrados. Podríamos declara el comportamiento común de todas estas clases, por ejemplo el método `dibujate()` en una clase padre llamada `Figuras`, y cada una de las clases hijas tuviese la implementación adecuada para dibujarse

dependiendo de su naturaleza. De modo muy esquemático el código de estas clases podría ser algo como lo mostrado en el Listado 3.4:

```
1 public abstract class Figura {
2  public abstract void dibujate();
3  // Sigue la definición de la clase
4 }
5
6 public class Triangulo extends Figura {
7  public void dibujate() {
8   // Código para dibujar un triángulo
9  }
10  // Sigue la definición de la clase
11 }
12
13 public class Cuadrado extends Figura {
14  public void dibujate() {
15   // Código para dibujar un cuadrado
16  }
17  // Sigue la definición de la clase
18 }
19
20 public class Circulo extends Figura {
21  public void dibujate() {
22   // Código para dibujar un círculo
23  }
24  // Sigue la definición de la clase
25 }
```

Listado 3.4: Definición de una clase `abstract` y algunas clases hijas.

La potencia del código del Listado anterior se hace evidente cuando recordamos que podemos asignar cualquier objeto de una clase hija (`Triangulo`, `Cuadrado` o `Circulo`) a una referencia de la clase padre `Figura`, de modo que podríamos escribir algo como:

```
1 Figura figura = new Circulo();
2 figura.dibujate(); // Dibujará un círculo
3 figura = new Triangulo();
4 figura.dibujate(); // Dibujará un triángulo
5 figura = new Cuadrado();
6 figura.dibujate(); // Dibujará un cuadrado
```

Listado 3.5: Uso de una referencia a una clase padre `abstract` para recorrer instancias de las clases hijas.

De nuevo la *Vinculación dinámica* en cada una de las llamadas al método `dibujate()` determinará el método que se debe invocar.

Buenas prácticas

El diseño de tus aplicaciones debe estar orientado al interface no a la implementación.

¿Qué quiere decir esta buena práctica? La idea es que debes concentrarte en crear buenas abstracciones, es decir debes intentar encontrar el comportamiento común (declaración de los métodos) a tus clases para que las puedas tratar de manera homogénea, con independencia de cómo se materializa ese comportamiento común (implementación de los métodos en cada clase). En el caso de la aplicación para dibujar figuras geométricas, el comportamiento común es el

hecho de que todas las figuras geométricas puede dibujarse. La implementación concreta es el modo en que cada una de las clases hija se dibuja.

El uso de esta buena práctica en el Listado 3.5 es que las referencias deben ser siempre del tipo más general posible (clase abstracta o `interface` como veremos en la siguiente sección), y no de una clase concreta. Rescribamos el código del último listado haciendo caso omiso de esta buena práctica:

```
1 Circulo figura = new Circulo();
2 figura.dibujate(); // Dibujará un círculo
3 figura = new Triangulo(); // Error nuestra figura es un \texttt{Circulo}
      no un \texttt{Triangulo}
4 figura.dibujate();
5 figura = new Cuadrado(); // Error nuestra figura es un \texttt{Circulo}
     no un \texttt{Cuadrado}
6 figura.dibujate();
```

Como ves, no podemos aprovechar el comportamiento polimórfico de nuestras figuras ya que las referencias son de un tipo concreto y no de un tipo abstracto, la clase `Figura`.

En una clase hija también podemos declarar como `abstract` algún método definido en la clase padre y que por lo tanto no es `abstract`. ¿Cuando nos puede interesar esta estrategia? Puede ser interesante para *borrar* el comportamiento por defecto que ofrece la clase padre, ya que si la clase hija a su vez es extendida por otra clases, estas debería definir el método declarado abstract. Obviamente, si una clase hija declara como `abstract` un método de la clase padre, aunque este no fuese `abstract` en la clase padre, la tendremos que declarar como `abstract`.

3.4. Interfaces.

Los `interface` son una nueva construcción del lenguaje Java que da un paso más allá en las clases `abstract`. Puedes pensar que un `interface` es como una clase `abstract` en la que todos sus métodos son `abstract`.

Siguiendo con el ejemplo de las figuras geométricas de la Sección 3.3 podemos definir nuestro primer `interface` como:

```
1 public interface Dibujable {
2  public void dibuja();
3 }
```

Como ves, estamos usando la palabra reservada `interface` para indicar que estamos definiendo un interface.

Las clases no *extienden* a los interfaces si no que los *implementan* y esto se indica con el uso de la palabra reservada `implements` de este modo:

```
1 public class Triangulo implements Dibujable {
2  @Override
3  public void dibuja() {
4   // Código para dibujar un triángulo
5  }
6  // Sigue la definición de la clase
7 }
8
9 public class Cuadrado implements Dibujable {
10 @Override
11 public void dibuja() {
12  // Código para dibujar un cuadrado
```

```
13  }
14  // Sigue la definición de la clase
15 }
16
17 public class Circulo implements Dibujable {
18  @Override
19  public void dibuja() {
20   // Código para dibujar un círculo
21  }
22  // Sigue la definición de la clase
23 }
```

Fíjate que para indicar que las clases están implementando un método declarado en un `interface` se anota el método con `@Override`.

Y ahora, de nuevo, aparece la magia del polimorfismo y la vinculación dinámica: a una referencia de un tipo `interface` le podemos asignar cualquier objeto de una clase que implemente ese `interface`, de modo que el código del siguiente listado es perfectamente válido:

```
1 Dibujable figura = new Circulo();
2 figura.dibuja(); // Dibujará un círculo
3 figura = new Triangulo();
4 figura.dibuja(); // Dibujará un triángulo
5 figura = new Cuadrado();
6 figura.dibuja(); // Dibujará un cuadrado
```

Fíjate que hemos utilizado una referencia de un tipo lo más amplio posible `Dibujable` y el comportamiento se materializa en la creación de las instancias de clases concretas: `Circulo`, `Triangulo` o `Cuadrado`.

En la definición de un `interface` podemos declarar cualquier número de métodos y también cualquier número de constantes como en el siguiente ejemplo:

```
1 public class interface Dibujable {
2  public static final Color BLANCO = new Color(255,255,255);
3  public static final Color NEGRO = new Color(0,0,0);
4  public void dibuja();
5 }
```

El anterior `interface` tiene contiene la declaración de dos constantes, una define el color BLANCO y la otra el color NEGRO.

Una ventaja del uso de `interfaces` para modelar el comportamiento de las clases es que una clase puede implementar cualquier número de `interfaces`. Recuerda que en el caso de la extensión de una clase Java sólo permite herencia simple. En el siguiente Listado se muestra un ejemplo de clase que implementa más de un `inteface`:

```
1 // Declaración de un nuevo \texttt{interface}
2 public class interface Transformable {
3  public void escala(int sx, int sy);
4  public void desplaza(int dx, int dy);
5 }
6
7 // Declaración de la clase
8 public class Circulo implements Dibujable, Transformable {
9  @Override
10  public void dibuja() {
11   // Aquí la definición del método
12  }
13
14  @Override
15  public void escala(int sx, int sy) {
```

```
16   // Aquí la definición del método
17  }
18
19  @Override
20  public void desplaza(int dx, int dy) {
21   // Aquí la definición del método
22  }
23 }
```

Los `interface` al igual que las clases se pueden extender, y además un `interface` puede extender a más de un `interface`, la herencia simple es una restricción en el ámbito de las clases, los `interface` no poseen esta restricción. En el siguiente listado tienes un ejemplo:

```
1 public interface Figura extends Dibujable, Transformable {
2  public void gira(float angulo);
3 }
```

Listado 3.6: Un `interface` que extiende a otros dos

Con el uso de los `interface` la *Buena práctica* de programar orientado al interface toma aún más fuerza. Por cierto no confundas la palabra reservada `interface` con el concepto de interface, este concepto se refiere a los métodos accesibles de una clase que son los que podemos utilizar para trabajar con la clase.

Hasta aquí hemos visto el grueso del trabajo con herencia, hemos visto cual es su potencia y como trabajar con ella en Java. Y todo ello para llevarnos el siguiente jarro de agua fría:

Buenas prácticas

En tus diseños software, favorece la composición frente a la herencia.

¿Qué significa esta buena práctica? ¿No debemos utilizar nunca la herencia? No, todo lo contrario, la herencia es una técnica de POO a objetos muy potente siempre que se la utilice bien. Un mal uso de la herencia es utilizarla para *reaprovechar* código. La herencia significa que entre los conceptos que queremos abstraer existe una clara relación padre-hijo. No debemos utilizar la herencia para *reaprovechar* código entre clases que no están relacionadas lógicamente a traves de la herencia.

Por ejemplo, tiene poco sentido que la clase `Persona` sea clase padre de la clase `Ciudad` simplemente porque una ciudad tenga un nombre. Este no es un buen uso de la herencia. La relación entre esos entes en el mundo real no existe y no debemos trasladarla de un modo artificial a nuestro código.

La composición, significa que una clase contiene como atributos instancias de otras clases. Este mecanismo de relación entre clases es más flexible que la herencia y por lo tanto menos sensible a los cambios que tengamos que hacer en nuestro código, más aún si como referencias a las clases utilizamos interfaces o clases abstractas.

En definitiva, favorecer la composición frente a la herencia significa usar la herencia sólo cuando esté justificado en nuestro diseño y no sólo por comodidad.

3.5. Enumeraciones.

Las enumeraciones son una construcción del lenguaje introducida en la versión 5 de Java. Las enumeraciones nos sirven para definir listas enumeradas de elementos y algo más, ya que en cierto modo son como clases, pueden tener constructores, métodos y atributos. El primer elemento de la enumeración tiene la posición 0.

La única restricción que tiene las enumeraciones sobre las clases es que las enumeraciones no se pueden extender ya que implícitamente toda enumeración está extendiendo a la clase `java.lang.Enum`. Además, el ámbito de los constructores debe ser `private` o de paquete.

La clase `Enum` nos proporciona algunos métodos útiles que podemos utilizar en nuestra propias enumeraciones, como veremos a continuación. Podemos definir una enumeración de una manera tan sencilla como la mostrada en el siguiente Listado:

```
1 public enum Semana {
2  LUNES("Primer día de la semana."),
3  MARTES("Ni te cases ni te embarques."),
4  MIERCOLES("Sin comentarios."),
5  JUEVES("Siempre en medio."),
6  VIERNES("Último día de trabajo."),
7  SABADO("Empieza el fin de semana."),
8  DOMINGO("Mañana de nuevo a trabajar.");
9
10  private String comentario;
11
12  // Constructor acceso de paquete o private.
13  Semana(String comentario) {
14   this.comentario = comentario;
15  }
16
17  public String getComentario() {
18   return comentario;
19  }
20 }
```

Listado 3.7: Definición de una enumeración para los días de la semana

La clase `Enum` nos proporciona el método `values()` que devuelve un array de `String` con todos los nombre de los elementos de la enumeración. Cada uno de los elementos de la enumeración posee dos métodos heredados de la clase `Enum` uno para conocer el nombre del elemento `name()`, y otro para conocer el ordinal del elemento dentro de la enumeración `ordinal()`, además, evidentemente, de los métodos que nosotros mismo hayamos definido (el método `getComentario()`, en el ejemplo del Listado 3.8. El siguiente Listado muestra un ejemplo de uso de la anterior enumeración:

```
1   for(Semana dia: Semana.values()){
2    System.out.println(dia.name());
3    System.out.println(dia.ordinal());
4    System.out.println(dia.getComentario());
5   }
```

Listado 3.8: Uso de una enumeración

3.6. Paquetes en Java.

Los paquetes son una construcción del lenguaje Java que nos permite agrupar clases que están lógicamente relacionadas en el mismo grupo o paquete. Para denotar que una clase pertenece a un determinado paquete, se indica en la definición de la clase con la palabra reservada `package` seguida del nombre del paquete como en el siguiente ejemplo:

```
1 package agenda; // Esta clase está dentro del paquete agenda
2
3 class Persona { // Visibilidad dentro del paquete
4  // Definición de la clase
5 }
```

Regla de convención

Los nombres de paquetes se escriben en minúsculas. Si el nombre de un paquete está compuesto por más de una palabra, se separan mediante puntos.

Un nombre válido de paquete con más de una palabra es: `agenda.datos` o `agenda.io.teclado`.

El ámbito o visibilidad por defecto de una clase es el paquete, por ello, para denotar que la visibilidad de una clase está restringida al paquete en la que está definida no se utiliza ningún modificador de acceso.

Como ya sabemos, la visibilidad también se define sobre los miembros de una clase de tal manera que un miembro puede se público, privado, protegido o de paquete. Veamos con mas detalle como se restringe la visibilidad con cada uno de estos modificadores de acceso.

El modificador de acceso `private` es el más restrictivo de los cuatro, un miembro privado no es accesible por ninguna otra clase. Podemos utilizar este modificador de acceso para ocultar completamente miembros de una clase. Una clase nunca se puede definir como `private`.

El modificador de acceso por defecto, o de paquete, hace visibles los miembros de una clase al resto de clases dentro del mismo paquete. Una clase puede definirse como de paquete.

El modificador de acceso `protected`, se comporta exactamente igual que el modificador por defecto pero además permite que las clases hijas de la clase `protected` puedan usar sus miembros a través de la herencia aunque estas clases hijas pertenezcan a paquetes distintos.

El modificador de acceso `public` asigna la mayor visibilidad a los miembros de una clase. Un miembro público es accesible desde cualquier otra clase sin importar el paquete en el que esté definido, o si existe una relación padre-hija entre ellas.

La Tabla 3.1 muestra todas las posibilidades de acceso entre miembros de clases.

En la Figura 3.1 su muestra gráficamente las posibilidades según el modificador de acceso empleado.

A partir de la versión 5 de Java se introdujo el concepto de `import static` con la intención de facilitar la escritura de código. La idea de los `import static`

¿Es accesible?	`private`	paquete	`protected`	`public`
Misma clase	SI	SI	SI	SI
Clase/subclase del paquete	NO	SI	SI	SI
Subclase otro paquete	NO	NO	SI [3]	SI
Clase otro paquete	NO	NO	NO	SI

Tabla 3.1: Modificadores de acceso y su visibilidad

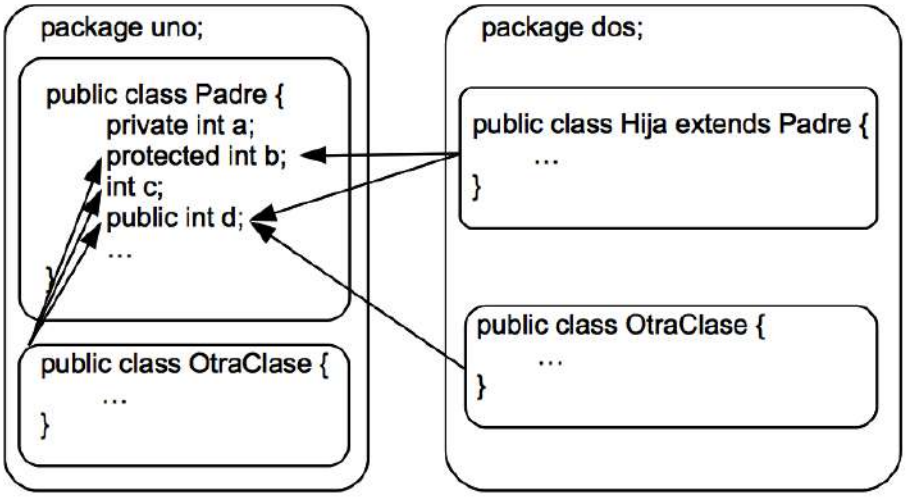

Figura 3.1: Visibilidad de los miembros según el modificador de acceso utilizado.

es incluir una clase o un paquete de clases y poder llamar a los miembros estáticos de las clases importadas sin necesidad de escribir el nombre de la clase, tal y como se muestra en el siguiente Listado:

```
1 // Definición de una clase
2 package paquete.subpaquete;
3
4 public class Clase {
5  public static void metodo() {
6   // Definición del método
7  }
8 }
9
10 // Definición de otra clase
11 import static paquete.subpaquete.Clase;
12
13 public class ClaseQueUsaImports {
14  public void otroMetodo() {
15   metodo();
```

Los `import static` son un arma de doble filo, por un lado facilitan la codificación, pero por otro se pierde la perspectiva de la pertenencia de los miembros `static` a sus clases concretas. Hay que utilizarlos con precaución. Un caso de uso comunmente extendido es en las pruebas unitarias, para incluir los miembros estáticos de los frameworks como JUnit .

3.7. Clases e `interface` anidados

Hasta este momento hemos definido cada una de nuestras clases en un fichero independiente con extensión `.java`. No obstante, en un mismo fichero podemos definir más de una clase siempre que sólo una de ellas sea `public` y su nombre coincida con el del fichero. El ámbito o visibilidad del resto de clases debe ser el paquete (recuerda, visibilidad por defecto). Aunque lo aconsejable es que, con independencia del ámbito, cada clase esté definida en un fichero distinto, ya que esto favorece el mantenimiento del código.

Si embargo, dentro de la definición de una clase podemos definir nuevas clases e interface como se muestra en el siguiente Listado:

```
1 public class Persona {
2  private String nombre;
3  private String apellidos;
4  private Direccion direccion;
5  private class Direccion {
6   private String calle;
7   private int numero;
8   private String puerta;
9   private String poblacion;
10  private String provincia;
11 }
12
13 public interface LeerDatos {
14  public String getNombre();
15 }
```

Listado 3.9: Uso de una enumeración

A la clase `Direccion` así definida se le llama clase interna y, a efectos de programación es una nueva clase como cualquier otra. De igual modo `interface LeerDatos` es un `interface` como otro cualquiera.

Hay un caso particular de creación de clases internas en el que la nueva clase no recibe ningún nombre como se muestra en el siguiente Listado, continuación del anterior:

```
17 public LeerDatos lector = new LeerDatos() {
18  private Persona persona;
19
20  @Override
21  public String getNombre() {
22   return nombre;
23  }
```

Listado 3.10: Uso de una enumeración

Fíjate que en la línea 17 parece que se está intentando instanciar un `interface`, cosa que como sabes no está permitida. Lo que está ocurriendo es que se está creando e instanciando una nueva clase sin nombre, y por lo tanto anónima, que está implementando el `interface LeerDatos`. Se está creando e instanciando la clase interna anónima al mismo tiempo, este es el único momento en el que se puede instanciar una clase interna anónima, ya que por ser anónima no tienen nombre y por lo tanto no podemos definir sus constructores.

Las clases internas anónimas son una construcción muy potente del lenguaje. Veremos toda su potencia en el Capítulo 11 dedicado a la creación de interfaces gráficos de usuario, y en el Capítulo 14 dedicado a la programación concurrente con hilos.

Cuestiones.

1. ¿Tiene sentido declarar los constructores de una clase como `private`? ¿Se podrán crear instancias de una clase en la que todos sus constructores son `private`? ¿Se podrá extender una clase en la que todos sus constructores son `private`?.

Ejercicios.

1. Modifica tu implementación de la clase `Agenda` que escribiste en el ejercicio 3 del Capítulo 2, para que pueda trabajar, de modo transparente, tanto con instancias de tipo `Persona` como con instancias de tipo `Empresa`.

2. Amplia la clase `Persona` para que contenga información sobre la dirección de residencia de la persona.

3. Amplia la clase `Empresa` para que contenga información sobre la dirección de la sede de la empresa.

4. Modifica tu agenda para que sea capaz de trabajar con los nuevos tipos de datos definidos.

Lecturas recomendadas.

- El capítulo 7 del libro de Sierra y Bates [3] expone gráficamente todos los conceptos relacionados con la herencia en Java.
- Para una exposición más detallada sobre clases e `interface` anidados una excelente referecia es el capítulo 5 del libro de James Gosling [2].
- Para un fundamentado razonamiento de porqué favorecer la composición sobre la herencia véase el item 14 de la referencia [4].

Capítulo 4

Control de versiones con *Subversion*

Contenidos

4.1. ¿Qué es un sistema de control de versiones? . . . 76
4.2. Principales características de *Subversion* 76
4.3. Creación de un repositorio 77
4.4. Trabajo con repositorios 78
4.4.1. Obteniendo información del repositorio 82
4.5. Integración con Eclipse 84

Introducción

Este es el primer capítulo dedicado a una herramienta utilizada en el desarrollo de proyectos informáticos y no directamente al lenguaje de programación Java.

Como ya se comentó en la introducción, el objetivo de este libro es mostrar cómo desarrollar proyectos informáticos con tecnología Java en un contexto de desarrollo lo más cercano posible al que el programador va a encontrar en cualquier organización de desarrollo de software. Lo que hace realmente valioso a un programador no es sólo que conozca lenguajes de programación, si no que conozca las herramientas de desarrollo de software más utilizadas y que, desde un principio, se integre suavemente en un equipo de desarrollo. Y esta integración tiene mucho que ver con toda la experiencia que posea con el trabajo en grupo y las herramientas que lo favorecen.

En este capítulo se presenta la herramienta de control de versiones Subversión, que, incluso trabajando individualmente, se hace imprescindible en todo proyecto para la gestión de las diferentes versiones del proyecto a medida que este evoluciona.

4.1. ¿Qué es un sistema de control de versiones?

El escenario más usual en el desarrollo de software es un equipo formado por varios programadores, probablemente coordinado por un jefe de proyecto. Sea cual sea la metodología que se utilice en el desarrollo del proyecto, el código va a estar sujeto a continuas modificaciones por parte de todos los programadores del equipo. En este escenario, no es raro encontrar que dos programadores han modificado el mismo fragmento de código de modo que se llegue a conflictos cuando se quiere unificar el código del proyecto. Otra necesidad del equipo es garantizar que todos los programadores pueden disponer de la última versión del código del proyecto.

Un Sistema de Control de versiones es una herramienta software que, de manera automática, se encarga de facilitar la gestión de las versiones del código de un proyecto de manera centralizada.

4.2. Principales características de *Subversion*

Subversion es una herramienta centralizada de ayuda al control de versiones. Su uso no es exclusivo en el desarrollo de proyectos informáticos, si no que puede utilizarse en cualquier proyecto que requiera de un sistema automático de control de versiones.

El concepto central en *Subversion* es el *Repsositorio* . Por repositorio se entiende la última versión del proyecto que existe en el sistema de control de versiones.

El paradigma que Suversion utiliza es *Copia-Modificación-Fusión* (Copy-Modify-Merge en inglés). En este paradigma, cada uno de los miembros del equipo, cuando empieza a trabajar en el proyecto, hace una copia local del contenido del repositorio; modifica su copia local y finalmente fusiona sus modificaciones locales con el código del repositorio, resolviendo los posibles conflicto que hayan aparecido. Al finalizar esta fase, se dice que se ha creado una nueva versión del proyecto en el repositorio.

Una de las característica principales de *Subversion* es que las actualizaciones en el repositorio son incrementales, sólo se actualizan los ficheros que se han modificado con respecto a la versión anterior. Otra característica es relativa a la numeración de la versión del repositorio, cada vez que se realiza una modificación en el repositorio, se actualiza la versión de todos los ficheros existentes en el repositorio y no únicamente de los ficheros que se han modificado.

Por otro lado, se puede trabajar con *Subversion* de manera local sobre el propio sistema de ficheros donde se realiza el desarrollo, o sobre un servidor en red. Y en este último caso, el servidor utilizado puede ser el propio servidor ad-hoc que viene incluido con la distribución de *Subversion* (`svnserve`), o como un módulo de *Apache*. La elección del modo de trabajo con *Subversion* se verá reflejada en la URL que utilizaremos para acceder al repositorio. Dependiendo del protocolo utilizado, las opciones son las que aparecen en la Tabla 4.1.

En la primera de las opciones de la Tabla 4.1 se accede directamente al repositorio en el sistema de ficheros. En la segunda de las opciones se accede utilizando el servidor ad-hoc que viene incluido en la propia distribución de *Subversion*. En la tercera opción se utiliza *Subversion* a través de un túnel *ssh*. La cuarta opción permite el acceso a un repositorio a través de Apache y el

file:///	El repositorio se encuentra en el disco local.
svn://	El acceso al repositorio se realiza a través del servidor `svnserve`.
svn+ssh://	El acceso al repositorio se realiza a través del servidor `svnserve` utilizando un túnel *SSH*
http://	El acceso al repositorio se realiza a través de Apache con el módulo *WebDAV*.
https://	El acceso al repositorio ser realiza con encriptación *SSL* a través de Apache con el módulo *WebDAV*.

Tabla 4.1: Tipos de acceso a los repositorios *Subversion*.

módulo WebDAV (del inglés *Web-based Distributed Authoring and Versioning*). Finalmente, en la última opción se accede al respositorio a través de un servidor Apache con encriptación *ssl* (del inglés *Secure Socket Layer*).

Cada una de estas opciones tiene sus ventajas y desventajas. En las próximas secciones utilizaremos el protocolo `svn://` para acceder a un repositorio a través del servidor `svnserve`. El trabajo con *Subversion* es independiente del protocolo utilizado.

4.3. Creación de un repositorio

La creación de un nuevo repositorio se hace utilizando la herramienta `svnadmin` incluida en la distribución de *Subversion*. Supongamos que hemos creado el directorio `./Repositorio` (directorio `Repositorio` en la raíz de nuestro directorio de usuario), en nuestro disco duro local. Para crear un repositorio *Subversion* en este directorio, en una consola escribiríamos:

```
~$ svnadmin create ~./Repositorio
```

Si examinamos el contenido del directorio veremos que se han creado los siguiente subdirectorios y ficheros dentro del directorio `./Repositorio`:

```
drwxr-xr-x   8 oscar  staff   272 23 may 18:48 .
drwxr-xr-x  32 oscar  staff  1088 23 may 18:48 ..
-rw-r--r--   1 oscar  staff   229 23 may 18:48 README.txt
drwxr-xr-x   5 oscar  staff   170 23 may 18:48 conf
drwxr-sr-x  16 oscar  staff   544 23 may 18:48 db
-r--r--r--   1 oscar  staff     2 23 may 18:48 format
drwxr-xr-x  11 oscar  staff   374 23 may 18:48 hooks
drwxr-xr-x   4 oscar  staff   136 23 may 18:48 locks
```

El fichero `README.txt` contiene un aviso sobre cómo debe ser usado este directorio, sólo a través de las herramientas que proporciona *Subversion*. El directorio `hooks` contiene scripts básicos para el trabajo con *Subversion*. El directorio `locks` es utilizado por *Subversion* para los bloqueos del repositorio. El directorio `db` es el que emplea *Subversion* para registrar todos los cambios realizados en el contenido del repositorio, es el corazón del repositorio. Finalmente, el directorio `conf` es donde se encuentran los ficheros de configuración para el acceso al servidor de *Subversion*.

A efectos prácticos, el directorio donde vamos a realizar tareas de configuración es `conf`. Su contenido es el siguiente:

```
drwxr-xr-x  5 oscar  staff   170 23 may 18:48 .
drwxr-xr-x  8 oscar  staff   272 23 may 18:48 ..
-rw-r--r--  1 oscar  staff  1080 23 may 18:48 authz
-rw-r--r--  1 oscar  staff   309 23 may 18:48 passwd
-rw-r--r--  1 oscar  staff  2279 23 may 18:48 svnserve.conf
```

En el fichero `svnserve.conf` indicamos las opciones de acceso al repositorio, en particular podemos restringir los permisos de lectura y escritura para cada uno de los usuarios a cada uno de los directorios que contiene nuestro repositorio a través de lo especificado en el fichero `authz`, y los usuarios y claves de acceso al repositorio en el fichero `passwd`.

Como ejemplo únicamente vamos a especificar los usuarios y sus claves en el fichero `passwd` sin modificar el fichero `authz`, lo que implica que todos los usuarios dados de alta en el fichero `passwd` tendrán acceso total a todos los directorios y ficheros del repositorio.

Para activar la opción de acceso a través de usuario autorizado hay que descomentar la línea:

```
password-db = passwd
```

esta línea indica el nombre del fichero de pares usuario/clave para nuestro repositorio. Después de descomentar esta línea debemos editar el fichero `passwd` y añadir los usuario y sus claves siguiendo el ejemplo que encontraremos en él:

```
# harry = harryssecret
# sally = sallyssecret
oscar = clave_secreta
```

Con esto ya tenemos activa la configuración más básica para nuestro repositorio, al que sólo puede acceder el usuario `oscar` con permisos de lectura y escritura para todos los directorios y ficheros del repositorio.

El siguiente paso antes de empezar a trabajar con nuestro repositorio es iniciar el servidor de *Subversion* del siguiente modo:

```
~$ sudo svnserve --daemon
```

Es necesario tener permisos de administrador para iniciar el servidor de *Subversion* como un proceso *daemon*.

A partir de ahora ya podemos empezar a trabajar con nuestro repositorio.

4.4. Trabajo con repositorios

Supongamos que, por claridad, elegimos nombrar al directorio que va a mantener nuestra copia de trabajo como `. /CopiaTrabajo` (en Eclipse nuestra copia de trabajo será algo como `. /workspace/NombreProyecto`, no es necesario que los nombre del proyecto y del repositorio coincidan). El primer paso que debemos dar es importar el contenido del directorio `CopiaTrabajo` al repositorio *Subversion*, suponiendo que nos encontramos en el directorio `CopiaTrabajo`, de este modo:

```
~/CopiaTrabajo$ svn import . svn://localhost/home/oscar/Repositorio/
trunk -m "Import inicial del proyecto"
```

El . corresponde al directorio actual, y la dirección que aparece a continuación `svn://localhost/home/oscar/Repositorio/trunk` corresponde a la dirección donde se encuentra el repositorio. Finalmente `-m Ïmport inicial del proyecto"` es un mensaje descriptivo de lo que estamos haciendo para que, a posteriori, resulte cómodo encontrar una determinada versión del proyecto. En este momento se solicitará la clave del usuario que hemos activado en el fichero de configuración `passwd`.

Te habrás dado cuenta de que estamos añadiendo el subdirectorio `trunk` al repositorio, esto forma parte de las buenas prácticas de trabajo con *Subversion*.

Buenas prácticas

En nuestro directorios *Subversion* es recomendable crear los siguiente subdirectorios:

trunk: Directorio donde se encuentra la versión en desarrollo del proyecto.

branches: Directorio donde se encuentran las posibles ramificaciones del proyecto.

tags: Directorio donde se encuentran las versiones finales (releases).

Para que los ficheros y directorios en `CopiaTrabajo` se conviertan en una copia de trabajo real, necesitamos hacer el `checkout` del repositorio hacia nuestra copia de trabajo de este modo (fíjate en el punto final «.» para indicar que el checkout lo queremos hacer sobre el directorio actual):

```
~/CopiaTrabajo$ svn checkout svn://localhost/home/oscar/Repositorio/
trunk .
```

En este momento tenemos sincronizada la copia en el repositorio con nuestra copia de trabajo. Para seguir la estructura típica de un proyecto Eclipse creemos el directorio `src` para contener los fuentes del proyecto. Cualquier modificación en el directorio de trabajo la tenemos que hacer utilizando *Subversion*, luego, en la consola debemos escribir lo siguiente para crear un directorio que *Subversion* pueda sincronizar con el repositorio:

```
~/CopiaTrabajo$ svn mkdir src
A         src
```

Subversion nos indicará que se ha añadido (A) el directorio src a la copia local. Ahora, dentro del directorio creado en el paso anterior creamos un fichero de definición de clase llamado `Saludo.java`, y le indicamos a Subverion que lo añadimos a nuestra copia local:

```
~/CopiaTrabajo$ touch src/Saludo.java
~/CopiaTrabajo$ svn add Saludo.java
A         src/Saludo.java
```

En este momento el fichero `Saludo.java` no se ha enviado al repositorio, sólo se ha marcado para que la próxima vez que se haga un `commit` este fichero se añada efectivamente al Repositorio.

Lo siguiente que debemos hacer es subir al repositorio todos los cambios que hemos hecho en nuestra copia local, de este modo:

```
~/CopiaTrabajo$ svn commit -m "Añadiendo la clase Saludo al
repositorio"
```

Observa que es necesario añadir un mensaje descriptivo de lo que estamos haciendo con la opción -m "Texto del mensaje", si lo olvidas, *Subversion* te lo pedirá. La respuesta que verás en consola por parte de *Subversion* será parecida a esta:

```
Añadiendo       src
Añadiendo       src/Saludo.java
Transmitiendo contenido de archivos .
Commit de la revisión 2.
```

Donde se nos informa que la última versión disponible en el repositorio es la 2. A la última versión se le llama *HEAD*. Ahora realiza algún pequeño cambio en el fichero `Saludo.java`, como añadir un simple comentario y graba tu fichero. Para conocer el estado de tu copia local con respecto a la última versión existente en el Repositorio puedes utilizar la instrucción `status` con la opción `-v` tecleando en consola:

```
~/CopiaTrabajo$ svn status -v
                 1        1 oscar        .
                 2        2 oscar        src
M                2        2 oscar        src/Saludo.java
```

Este texto nos informa que el fichero `Saludo.java` se ha modificado (letra inicial `M`), y en el próximo `commit` se subirá la versión local al repositorio, la opción `-v` significa `verbose`, es decir queremos una descripción detallada del estado de la copia local. Veamos el resultado de un `commit` escribiendo:

```
~/CopiaTrabajo$ svn commit -m "Una modificación en el fichero
Saludo.java"
Enviando       src/Saludo.java
Transmitiendo contenido de archivos .
Commit de la revisión 3.
```

Este texto nos informa, de nuevo, que no ha habido ningún problema al subir la nueva copia local al Repsositorio.

En un equipo de desarrollo cualquier otro programador puede realizar cambios sobre algún fichero en el Repositorio, o puede añadir nuevos ficheros y directorios al repositorio, para conocer en todo momento cual es el estado de nuestra copia local con respecto a la última versión existente en el Repositorio podemos escribir:

```
~/CopiaTrabajo$ svn status -u
*  3 src/Saludo.java
Estado respecto a la revisión: 5
```

La opción `-u` indica que se compare nuestra versión local con respecto a la última versión en el Repositorio, si no la incluimos la comparación se realizará entre la copia local y la última versión que nos descargamos desde el Repositorio que puede no ser la versión `HEAD` del Repositorio. El * indica que el fichero `Saludo.java` en mi copia de trabajo está en la versión 3 mientras que en el repositorio la última versión es la 5. Si queremos conocer cual es el estado local de nuestros fichero con respecto de la última actualización escribimos:

```
~/CopiaTrabajo$ svn diff
Index: src/Saludo.java
===================================================================
--- src/Saludo.java (revisión: 3)
+++ src/Saludo.java (copia de trabajo)
@@ -1,2 +1,3 @@
 public class Saludo {
+    // Un comentario
 }
\ No newline at end of file
```

Si vemos un signo «+» al inicio de la línea significa que esa línea se ha añadido con respecto de la última actualización que hicimos (que no tiene porqué coincidir con la última versión que existe en el repositorio). Si la línea empieza con un signo «-» indica que esa línea sea ha eliminado. Si ahora intentamos hacer un `commit` obtendremos el siguiente error:

```
~/CopiaTrabajo$ svn commit -m "Intentando un commit que fallará"
Enviando       src/Saludo.java
Transmitiendo contenido de archivos .svn: Falló el commit
(detalles a continuación):
svn: El archivo '/trunk/src/Saludo.java' está desactualizado
```

Este error se debe a que nuestra copia local se encuentra en la versión 3 y la última versión en el repositorio es la 5, luego es necesario que primero actualicemos nuestra copia local a la última versión en el repositorio, y en segundo lugar que enviemos los cambios. Para actualizar nuestra copia local a la última versión del Repositorio (*HEAD*) escribimos:

```
~/CopiaTrabajo$ svn update
G    svn/Saludo.java
Actualizado a la revisión 5.
```

Esta vez, la letra `G` al inicio de la línea indica que *Subversion* ha sido capaz de *mezclar* (*merge* en ingles) la última revisión existente en el Repositorio con los cambios locales en nuestro fichero y no ha encontrado conflictos. Si dos programadores no modifican las mismas líneas de código, si no que las discrepancias aparecen el líneas de código distintas, no aparecerá ningún conflicto cuando *Subversion* intente mezclar el código de nuestra copia local con el de la copia en el Repositorio. En este caso tenemos, en nuestra copia local, la última versión en el Repositorio más nuestros cambios que se han añadido sin conflicto, ahora ya podemos hacer de nuevo un `commit`:

```
~/CopiaTrabajo$  svn commit -m "Ya actualizado subo mis cambios"
Enviando       src/Saludo.java
Transmitiendo contenido de archivos .
Commit de la revisión 6.
```

Ahora sí que ha tenido éxito el `commit` puesto que la versión de nuestra copia local coincide con la última versión en el repositorio.

En otras ocasiones, cuando una línea ha sido modificada por dos o más programadores, *Subversion* no sabrá cómo resolver el conflicto por sí solo, y en el momento de intentar hacer un `update` seremos informados de que existe un conflicto, como en el caso siguiente:

```
~/CopiaTrabajo$  svn diff
Index: src/Saludo.java
===================================================
--- src/Saludo.java (revisión: 7)
+++ src/Saludo.java (copia de trabajo)
@@ -1,4 +1,7 @@
 @author Oscar
 public class Saludo {
+<<<<<<< .mine
+=======
     // Un comentario, un poco largo
+>>>>>>> .r7
 }
\ No newline at end of file
```

En este mensaje se nos informa que hay un conflicto ya que nuestra copia local contiene la línea del comentario que ha sido eliminada en el repositorio, de hecho, las líneas extra que aparecen en el código se han añadido realmente al fichero `Saludo.java`. En este caso debemos resolver el conflicto a mano, y una vez resuelto (por ejemplo, eliminando todas las líneas insertadas y manteniendo el comentario) se lo indicamos a *Subversion* del siguiente modo:

```
~/CopiaTrabajo$  svn resolved Saludo.java
Se resolvió el conflicto de 'src/Saludo.java'
```

De nuevo, podemos seguir trabajando con nuestro repositorio como hasta el momento o hasta que aparezca un nuevo conflicto.

4.4.1. Obteniendo información del repositorio

Sin ánimo de ser exhaustivos con respecto a las posibilidad para obtener información sobre un repositorio Subversión, aquí mostramos algunas de las opciones para conocer el estado del repositorio y de nuestra copia local. *Subversion* nos proporciona instrucciones para conocer en cualquier momento información sobre el repositorio.

Con `svn log` obtenemos información sobre los mensajes que fueron adjuntados con cada nuevo commit, tal y como se muestra a continuación:

```
------------------------------------------------------------
r2 | oscar | 2010-05-17 09:44:03 +0200 (lun, 17 may 2010) | 1 line

Código del capítulo Clases.
------------------------------------------------------------
r1 | oscar | 2010-05-17 09:43:33 +0200 (lun, 17 may 2010) | 1 line

Initial import.
------------------------------------------------------------
```

Si estamos interesados en alguna revisión en particular, podemos indicarlo con la opción `-r` como en el siguiente ejemplo:

```
caterva:LibroJava oscar$ svn log -r 10
------------------------------------------------------------
r10 | oscar | 2010-06-25 10:31:51 +0200 (vie, 25 jun 2010) | 1 line

Para el Capítulo de Entrada/Salida.
------------------------------------------------------------
```

Para conocer el estado del repositorio podemos usar `svn list`. De nuevo, si estamos interesados en el estado del repositorio para una determinada revisión podemos utilizar la opción `-r` tal y como se muestra en el siguiente ejemplo:

```
caterva:LibroJava oscar$ svn list -r 3
clases/
herencia/
```

Si lo que nos interesa es conocer el estado de las últimas modificaciones de nuestra copia local con respecto al repositorio, podemos utilizar la instrucción `svn status`, con lo que obtendremos información del modo siguiente:

```
?        Punto.java
!        Nuevo.java
A       Punto3D.java
C       Principal.java
D       PruebaPunto.java
M       Ejemplo.java
L        Hola.java
```

La letra mayúscula de la primera columna, antes del nombre de cada fichero, es un código que nos indica el estado del fichero o directorio con el siguiente significado:

En la tabla 4.2, el último código indica que el fichero ha quedado bloqueado. Esta situación puede ocurrir cuando, por ejemplo, al intentar hacer un `commit` la conexión con el repositorio remoto se pierde y no se puede acabar el envío. Al realizar cambios en la copia local, *Subversion* va acumulando en una lista todas las tareas pendientes. En el momento en que se desea sincronizar la copia local con la remota, se bloquean los ficheros que se van a sincronizar. Si por alguna razón alguna de las tareas pendientes no se puede llevar a cabo, el resultado será que algún fichero puede quedar bloqueado.

?	El fichero no se está añadido al repositorio
!	No existe una copia local de este fichero
A	El fichero se ha marcado para añadir al repositorio
C	Existen conflictos entre la copia local y el repositorio
D	El fichero se ha marcado para ser borrado del repositorio
M	Hay modificaciones en la copia local del fichero
L	El fichero está bloqueado

Tabla 4.2: Códigos con información sobre el estado de la copia local del fichero sobre la copia remota.

Para eliminar los bloqueos podemos utilizar la instrucción `svn cleanup`, esta instrucción comprueba el listado de tareas a realizar, y si queda alguna pendiente, intenta realizarla, al final de lo cual, se eliminará el bloqueo sobre los ficheros.

Finalmente, si lo que nos interesas conocer con detalle son los cambios producidos en los ficheros entre la copia local y la existente en el repositorio podemos utilizar `svn diff`.

4.5. Integración con Eclipse

Aunque el trabajo con *Subversion* desde línea de instrucciones es bastante sencillo, resulta interesante no tener que abandonar Eclipse para trabajar con el Repositorio de nuestro proyecto. Como se comentó en la Sección 1.4.1, se puede añadir nueva funcionalidad a Eclipse instalando nuevos *plug-ins*, y precisamente existen excelentes *plug-ins* para trabajar con *Subversion* desde Eclipse. Uno de ellos es *Subclipse*, (`http://subclipse.tigris.org/`), aunque existe otros excelentes *plug-ins* como *Subversive* (`http://www.eclipse.org/subversive/`). Elegir entre uno u otro acaba siendo cuestión de gustos ya que todos ellos son excelentes, la mejor idea es probar algunos de ellos y quedarnos con el que más se adapte a nuestra forma de trabajo.

La última versión del *plug-in* Subclipse puedes encontrarla en la dirección `http://subclipse.tigris.org`. En la sección *Dowload and Install* encontrarás la última *release* y la URL desde donde instalar el *plug-in*.

Para instalar un *plug-in* [1] selecciona la opción *Help* → *Install new Software* se abrirá una ventana con una sección *Work with* donde debes introducir la URL del *plug-in* Subclipse [2]. Al pulsar *Enter* te aparecerán todos los *plug-ins* disponibles en esa dirección, aunque algunos de ellos son opcionales, conviene instalarlos todos, así que marca todos ellos y pulsa el botón *Next*. En la siguiente ventana se te mostrarán a modo de información todos los paquetes que se instalarán. Pulsa de nuevo *Next*. En la siguiente ventana se te pide que aceptes los términos de la licencia, para ello selecciona la opción *I accept the terms and of the license agreements* y pulsa el botón *Finish*, verás como todos los paquetes de clases necesarios para el nuevo *plug-in* se descargan hacia tu máquina. Hacia el final de la instalación se abrirá una ventana de advertencia indicándote que

[1] Estas instrucciones son válidas para la versión 3.5 (Galileo) y la 3.6 (Helios) de Eclipse

[2] En el momento de escribir este libro la última versión de este *plug-in* es la 1.6 y su URL es `http://subclipse.tigris.org/update_1.6.x`

parte del software que estás instalando no está firmado, por esta vez pulsa la tecla *OK* para seguir con la instalación. Finalmente se te pedirá que para acabar con la instalación está recomendado reiniciar Eclipse, pulsa el botón *Yes* y Eclipse se reiniciará y ya podrás usar el *plug-in* recién instalado.

Ahora, dispondrás de una nueva *Perspectiva* que te permitirá visualizar tus repositorios *Subversion*. Para abrir esta nueva *Perspectiva*, selecciona la opción de menú de Eclipse *Window* → *Show perspective* → *Other....* Al hacerlo se abrirá una ventana, en ellaselecciona *SVN Repository Exploring*. Dentro de esa nueva *Perspectiva* puedes abrir una nueva vista con la opción de menú *Window* → *Show View* → *SVN Repositories*. Finalmente, se abrirá una *Vista* con el título *SVN Repositories* que inicialmente estará vacía.

En esta nueva vista es donde puedes añadir conexiones a los repositorios *Subversion* que quieras. Como ejemplo, vamos a crear una conexión al repositorio local que tenemos en `svn://localhost/home/oscar/Repositorio/trunk` (esta dirección no tiene por qué coincidir con la que hayas elegido tú), para ello, sobre la vista *SVN Repositories* pulsa el botón derecho de tu ratón y en el menú emergente que aparecerá selecciona *New* → *Repository Location*, se abrirá una ventana solicitándote la URL del repositorio al que te quieres conectar, introduce `svn://localhost/home/oscar/Repositorio`, fíjate que no hemos introducido *trunk* al final de la URL, pulsa *Finish*, verás que la nueva URL te aparece en la vista *SVN Repositories*, despliega la URL y verás bajo ella el directorio *trunk* y bajo él, el fichero *Saludo.java*. Para hacer el *chekout* del repositorio selecciona *trunk* con el botón derecho y en el menú contextual que te aparecerá selecciona *Checkout...* se abrirá una nueva ventana con la opción seleccionada por defecto *Check out as a project configured using New Project Wizard*, pulsa *Finish*, y en la nueva ventana despliega la opción *Java* y selecciona la opción *Java Project*, a partir de aquí, el *Wizard* es el que ya conoces y utilizas cada vez que quieres crear un nuevo proyecto, así que simplemente introduce un nombre para el nuevo proyecto y pulsa *Finish*.

Al final de este proceso tendrás una copia local del proyecto en Eclipse sobre la que puedes trabajar tal y como trabajas sobre cualquier otro proyecto Eclipse, con la enorme ventaja de que todo el trabajo con el repositorio puedes hacerlo desde el propio Eclipse. Cada vez que queramos ver las discrepancias entre nuestra copia local y la existente en el repositorio, es decir la información que en consola obteníamos con `svn diff`, ahora la podemos obtener, de manera gráfica, pulsado el botón derecho del ratón sobre el proyecto en Eclipse y eligiendo del menú contextual *Team* → *Synchronize with repository*, pasaremos a la *perspectiva Team Synchronization* donde nos aparecerá toda la información de sincronización con el repositorio. La interpretación de los iconos es la que se muestra en la Tabla 4.3.

Si existe conflicto en alguno de nuestros ficheros de código podemos abrir una *Vista* donde se nos muestra, al mismo tiempo, el estado de nuestra copia local y el estado del fichero en el repositorio, para que podamos resolver los conflictos cómodamente. Una vez resueltos los conflictos pulsamos el botón derecho del ratón y elegimos *Mark as Merged* para indicar que hemos resuelto los conflictos y que nuestra copia local está lista para ser subida al repositorio (*commit*), solo nos restará hacer botón derecho sobre el nombre del fichero y seleccionar *Commit...*, se abrirá una ventana para introducir un comentario para el *commit* y pulsamos *OK*.

Flecha azul hacia la izquierda	La versión del repositorio es más actual que la copia local.
Flecha gris hacia la derecha	La versión local es más actual que la existente en el repositorio.
Doble flecha roja	Existe un conflicto que *Subversion* no sabe resolver entre la copia local y la existente en el repositorio.

Tabla 4.3: Significado de los iconos en la perspectiva *Team Synchronization*

Lecturas recomendadas.

- La referencia obligada para conocer cualquier aspecto de *Subversion* es [7]. Existe una versión gratuita que se puede descargar desde `http://subversion.apache.org/`.
- Otra referencia más compacta es el capítulo 4 del la referencia [13], donde se detallan los puntos principales para empezar a trabajar con *Subversion*.

Capítulo 5

Excepciones

Contenidos

5.1. ¿Qué es una excepción? 87
5.1.1. Tipos de excepciones 88
5.2. Cómo se gestiona una excepción 88
5.3. Creación de excepciones propias 91

Introducción

Es evidente que lo deseable es que no se produzcan errores durante la ejecución de un programa. A todos nos provoca rechazo utilizar herramientas que fallan, ya sean herramientas software o cualquier otro tipo de herramientas. En el capítulo 6 veremos la técnica de *Test unitarios* para comprobar nuestro software con objeto de eliminar el mayor número posible de errores durante la fase de desarrollo de nuestras aplicaciones. No obstante, durante la ejecución de nuestras aplicaciones existirán situaciones anómalas susceptibles de provocar un mal funcionamiento de nuestro software. Piensa por ejemplo en el caso en que un usuario intenta guardar un fichero en un directorio protegido contra escritura.

Existen lenguajes de programación, como C++, que proporcionan un mecanismo opcional de reacción frente a estas situaciones anómalas, pero el programador no está obligado a utilizarlo, es como se ha dicho, una opción.

En Java existe un mecanismo de reacción ante situaciones anómalas muy parecido al de C++ con la gran diferencia de que el programador sí que está obligado a usarlo en aquellas situaciones susceptibles de provocar un error, lo que conduce a la producción de código más robusto frente a este tipo de fallos en tiempo de ejecución.

5.1. ¿Qué es una excepción?

En Java, una excepción es una situación anómala en tiempo de ejecución. Piensa en el ejemplo de la introducción en el que un usuario intenta guarda un fichero en un directorio protegido contra escritura. Piensa también en el acceso a una

posición fuera de un array estático. Todo este tipo de situaciones se producen en tiempo de ejecución, y aunque podemos estar prevenidos contra ellas, no podemos evitar completamente que vayan a ocurrir.

5.1.1. Tipos de excepciones

En Java existen tres grandes grupos de excepciones dependiendo de su naturaleza:

1. Excepciones de la propia máquina virtual. Estas excepciones causadas por un mal funcionamiento de la propia máquina virtual (Sí, la máquina virtual Java también es una pieza de software y como tal está sujeta a fallos). Este tipo de errores, por su naturaleza, son ajenos al programador y por lo tanto no estamos obligados a gestionarlos. Si este tipo de errores se produce durante la ejecución de nuestras aplicaciones puede ocurrir que nuestra aplicación se cierre y veamos un mensaje de error de la propia máquina virtual. Pero quede el lector tranquilo, es extremadamente difícil encontrarse con un error de esta naturaleza. En la Figura 5.1 la clase `Error` es la clase padre de todo este grupo de excepciones.

2. El siguiente grupo de situaciones excepcionales son aquellas tan comunes cómo intentar acceder a una posición inexistente de un array estático; o intentar hacer un casting incompatible sobre una variable de tipo referencia. El código donde se puede dar este tipo de situaciones es tan común que añadir más código para gestionarlas sobrecargaría terriblemente la escritura de nuestros programas, por lo que no es necesario gestionar este tipo de excepciones, aunque si queremos siempre lo podemos hacer. En la Figura 5.1 la clase `RunTimeException` es la clase padre de este grupo de excepciones.

3. El tercer y último tipo de excepciones está formado por el resto de situaciones que no son las anteriores, como por ejemplo, de nuevo, intentar escribir en un directorio protegido contra escritura. Este es el tipo de excepciones que sí estamos obligados a gestionar y para los que Java proporciona un potente mecanismo de gestión. En la Figura 5.1 la clase `Exception` es la clase padre de este grupo de excepciones.

5.2. Cómo se gestiona una excepción

Java proporciona un mecanismo de gestión de errores a través de los bloques `try...catch...finally`, tal y como se muestra en el siguiente Listado:

```
1 try {
2  Fichero f = abreFichero(nombre); // Quizás el fichero no exista
3  String linea = f.leeLinea(); // Quizás se produzca un error durante la
       lectura
4 } catch(FileNotFoundException e) {
5  // Código de recuperación del error
6  System.out.println(e.getMessage()); // Muestra una descripción del
       error
7 } catch(IOException e) {
8  // Código de recuperación del error
9 } finally {
```

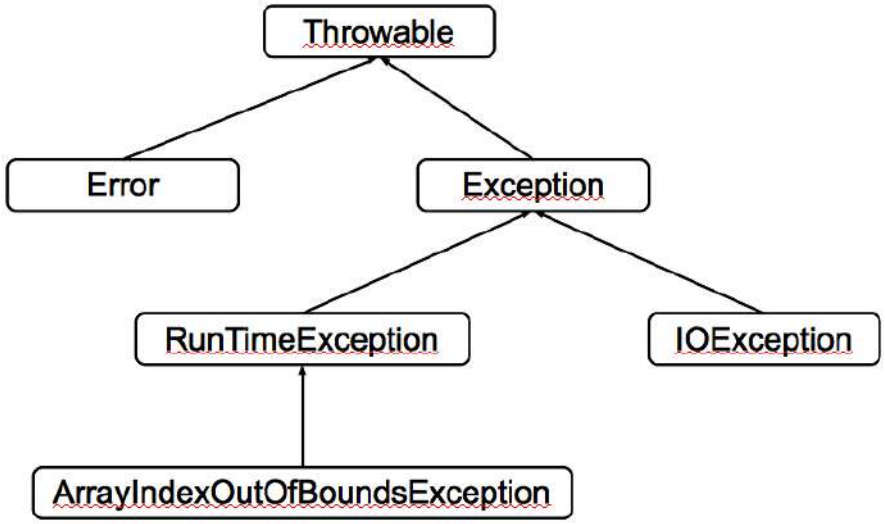

Figura 5.1: Parte del árbol de jerarquía de las excepciones en Java.

```
10  // Código común
11 }
12 // Otras líneas de código
13 System.out.println("Aquí sigue la ejecución");
```

Listado 5.1: Ejemplo bloque `try{...} catch{...} finally{...}`

En el ejemplo anterior, el método de la línea 2 está intentando abrir un fichero, y una posible situación anómala es que el fichero no exista (`FileNotFoundException`) lo que puede provocar un error. En la línea 3 se está intentando leer una línea desde el fichero que sí está abierto, lo que también puede provocar algún tipo de error de entrada/salida.

Antes de pasar a ver la técnica para gestionar estas situaciones anómalas, fíjate que cuando se produce un error, lo que recibe el bloque `catch{...}` es una referencia del tipo del error correspondiente, que, entre otras cosas, lleva una descripción sobre el error que se produjo. Recuerda, todo en Java es un objeto y en particular los errores en tiempo de ejecución son instancias de clases que representan errores.

La técnica para gestionar esta situación es:

1. Encerrar en un bloque `try{...}` el o los métodos susceptibles de provocar un error.
2. Atrapar en bloques `catch{...}` separados, cada uno de los posibles errores que se pueden producir en el bloque `try{...}`.
3. Opcionalmente podemos definir un bloque `finally{...}` que se ejecutará con independencia de que se genere alguna de las excepciones gestionadas o no. Es decir si existe el bloque `finally{...}` su código siempre se ejecutara se produzca o no una excepción.

En el listado 5.1, si en la línea 2 se produce un error porque no se encontrase el fichero que se desea abrir, la siguiente línea de código 3 no se ejecutaría, la

ejecución pasaría directamente al bloque `catch(FileNotFoundException e)`, y tras su ejecución al bloque `finally` de la línea 10, ya que tenemos definido uno. Después de la gestión del error, la ejecución seguirá en la línea 13. Si tanto el código en el bloque `finally{...}` como el código posterior a la gestión de la excepción se ejecuta (código en la línea 13), ¿Qué sentido tiene incluir el bloque `finally{...}`?. Antes de contestar a esta pregunta veamos cómo podemos obviar la gestión de una excepción en el momento en que esta se produce y delegar su gestión en el método que invocó al actual en la pila de llamadas.

Existen casos es los que nos interesa delegar la gestión de la excepción, la excepción ocurre en la definición de un determinado método pero no queremos añadir el código de gestión de la excepción en la definición de ese método, ¿Cómo indicamos que un método no va a gestionar una determinada excepción?, con el uso de la palabra reservada `throws` como en el siguiente listado:

```
1 public void metodo() throws FileNotFoundException {
2  // Aquí la definición del método
3 }
```

En este caso estamos delegando la gestión de la excepción al método que llamó a este otro, que es quien delega la gestión de la excepción.

La palabra reservada `throws` también se utiliza para lanzar excepciones propias tal y como vamos a ver en la Sección 5.3.

Ahora ya podemos contestar a la pregunta sobre la utilidad del bloque `finally{...}` observando el ejemplo del Listado 5.2. Si se produjese la excepción `IOException` en la línea 4 durante el proceso de lectura, se abandonaría la ejecución del método delegando la gestión de estas excepciones al método que invocó a este (`ejecuta()`). La línea 6 nunca se ejecutaría y el fichero quedaría abierto, su referencia perdida al salir del método y no podríamos cerrar el fichero.

```
1 // Este método delega la gestión de las excepción FileNotFoundException
      y IOException
2 private void ejecuta() throws FileNotFoundException, IOException {
3  FileReader fr = new FileReader("fichero.txt");
4  int caracter = fr.read();
5  System.out.println("caracter: " + caracter);
6  fr.close();
7 }
```

Listado 5.2: Tanto la excepción FileNotFoundException como IOException se delegan

En el ejemplo del Listado 5.3, tanto si se produce una excepción, como si no se produce, el bloque `finally{...}` siempre se ejecutará y el fichero se cerrará en cualquier caso, se produzca o no una excepción.

```
1 // Este método delega la gestión de las excepción FileNotFoundException
      y IOException
2 private void ejecuta() throws FileNotFoundException, IOException {
3  FileReader fr = null;
4  int caracter = 0;
5  try{
6   fr = new FileReader("fichero.txt");
7   caracter = fr.read();
8  } finally {
```

```
9   System.out.println("caracter: " + caracter);
10  if(fr != null) fr.close();
11 }
12 }
```

Listado 5.3: En este caso el fichero siempre se gracias al uso del bloque `finally{...}`

Otro detalle importante es el orden de los errores que se atrapan en los bloques `catch{...}`. Compara el orden del ejemplo del Listado 5.1. Fíjate que el orden en los bloques `try{...} catch{...} finally{...}` va desde el más específico (`FileNotFoundException`) al más general (`IOException`). Piensa qué ocurriría si se intercambiase el orden, la clase padre aparecería en el primer bloque `catch{...}` de modo que tanto si se produjera una excepción de tipo `IOException` como si fuera de tipo `FileNotFoundException` ambas provocaría la ejecución del bloque `catch{IOException}` ya que este bloque atrapa referencias de la clase padre y cualquier clase hija. Recuerda, las excepciones también son instancias de objetos, y por lo tanto sobre ellas es válido todo lo que aprendimos sobre herencia en el capítulo 3.

5.3. Creación de excepciones propias

En el desarrollo de nuestras propias aplicaciones resulta interesante poder lanzar excepciones propias ante situaciones inesperadas. Java proporciona un mecanismo para definir nuevas excepciones y lanzarlas en los casos en los que se produzcan situaciones anómalas.

El mecanismo para definir y lazar excepciones propias es el siguiente:

1. Definir la nueva clase que representa a nuestra excepción.
2. Lanzar la excepción en las situaciones anómalas.
3. Gestionar la excepción como cualquier otra.

Al utilizar este mecanismo estamos creando excepciones que son tratadas del mismo modo que las excepciones predefinidas en Java. Veamos cada uno de estos pasos con un ejemplo. Supongamos que queremos generar una excepción si se solicita una posición no válida dentro de nuestra aplicación de la Agenda. Lo primero que debemos hacer es definir la clase que representa a nuestra nueva excepción. El detalle que debemos tener en cuenta es que nuestra excepción debe ser hija de la clase `Exception`, tal y como se muestra en el Listado 5.4:

```
1 public class TemperaturaNoValidaException extends Exception {
2  public TemperaturaNoValidaException() {
3   super("La temperatura no puede se menor que -273°C");
4  }
5 }
```

Listado 5.4: Definición de una excepción propia

Fíjate que el constructor por defecto llama a `super` del padre con un `String` que es una descripción del error que ha ocurrido. Este `String` se podrá recuperar en el bloque `catch{...}` correspondiente como veremos más adelante.

El siguiente paso es lanzar la excepción en caso de producirse una situación anómala, como en el Listado 5.5 :

```
1 public class ConversorTemperaturas {
2  private final double CERO_ABSOLUTO = -273.15;
3
4  public ConversorTemperaturas() {
5   super();
6  }
7
8  public double celsiusAFharenheit(double celsius) throws
       TemperaturaNoValidaException {
9   if(celsius < CERO_ABSOLUTO) throw new TemperaturaNoValidaException();
10  return 9.0/5.0 * celsius + 32.0;
11 }
12
13 public double celsiusAReamur(double celsius) throws
       TemperaturaNoValidaException {
14  if(celsius < CERO_ABSOLUTO) throw new TemperaturaNoValidaException();
15  return 4.0/5.0 * celsius;
16 }
17 }
```

Listado 5.5: Definición de una excepción propia

Cabe resaltar un par de cosas del Listado 5.5, la primera es que el método desde el que se lanza la excepción indica que va a hacerlo con el uso de la palabra reservada `throws` seguida del nombre de la excepción. La segunda es que para lanzar la excepción utilizamos la palabra reservada `throw` y creamos una nueva instancia de la excepción con `new`, recuerda que una excepción al fin y al cabo no es más que una instancia de una clase.

Nuestra excepción se gestiona como cualquier otra ya definida en el paquete estándar de Java, mediante el bloque `try{...} catch{...} finally{...}`, tal y como se muestra en el Listado 5.6. En la línea 7 de este listado se muestra cómo recuperar el texto descriptivo de la excepción que proporcionamos en la definición de la clase `TemperaturaNoValidaException`. Un método útil para recuperar toda la traza de ejecución de nuestro programa hasta el momento en el que se produzco la excepción es `printStackTrace()` definido en la clase `Throwable` que es la clase de la que heredan todas las excepciones en Java.

```
1  for(int celsius = 0; celsius < 101; celsius += 5) {
2   try {
3    fharenheit = conversor.celsiusAFharenheit(celsius);
4    reamur = conversor.celsiusAReamur(celsius);
5    System.out.println(celsius + "\t" + fharenheit + "\t" + reamur);
6   } catch(TemperaturaNoValidaException e) {
7    System.out.println(e.getMessage());
8   }
9  }
```

Listado 5.6: Definición de una excepción propia

Lecturas recomendadas

- El capítulo 8 de la referencia [2] presenta todos los detalles de la gestión de excepciones y cómo crear excepciones propias.

Capítulo 6

Pruebas unitarias con JUnit

Contenidos

6.1. ¿Qué son las pruebas unitarias? 94
6.1.1. Principios FIRST para el diseño de pruebas unitarias 94
6.2. Pruebas unitarias con JUnit 95
6.2.1. Creación de clases de prueba 95
6.2.2. La anotación `@Test` 96
6.2.3. Las anotaciones `@Before` y `@After` 98
6.2.4. Las anotaciones `@BeforeClass` y `@AfterClass` 99
6.2.5. Pruebas con batería de datos de entrada 100
6.2.6. Ejecutar varias clases de prueba. Test Suites 101
6.3. Cobertura de las pruebas 102
6.3.1. EclEmma y su plug-in para Eclipse 103

Introducción

Llegados a este punto ya somos capaces de escribir aplicaciones Java utilizando los principios de la POO. Sabemos cómo definir clases, como utilizar la herencia o la composición para ampliar el comportamiento de nuestras clases. Incluso somos capaces de controlar las situaciones anómalas que pueden darse durante la ejecución de nuestras aplicaciones a través de la gestión de excepciones.

El siguiente paso que usualmente se suele dar es comprobar la validez del código realizando pruebas. A veces, estas pruebas son *caseras*, probamos unos cuantos valores de entrada en nuestra aplicación, valores de los que conocemos cual es la salida esperada, y confiamos que en el resto de casos nuestro código esté libre de errores. Y en este momento es cuando la confianza se convierte en engaño, nuestro código está plagado de errores que no hemos sido capaces de detectar y que tarde o temprano saldrán a la luz sumiéndonos en la oscuridad, paradojas de la vida.

La primera idea que debemos fijar es que hacer pruebas de código no debe ser una opción, es un requerimiento, por defecto, en todo desarrollo de proyectos informáticos.

La segunda idea que debemos fijar es que las pruebas de código no deben ser manuales, si no automatizadas. Si son manuales, por aburrimiento o falta de tiempo acabaremos por no hacerlas. Las pruebas automatizadas forman parte del código del proyecto, son tan importantes como el código que se está probando y por lo tanto debemos dedicarles el mismo empeño que al desarrollo del código de nuestra aplicación.

En este capítulo no se va a mostrar cómo diseñar buenas pruebas de código, y lo que es más importante, no se va a mostrar cómo escribir código que se pueda probar fácilmente, en la sección de referencias de este capítulo se dan algunos títulos que son de lectura obligada a todo desarrollador que quiera poner en práctica la prueba de código en sus proyectos.

En este capítulo se va a mostrar cómo utilizar una herramienta para comprobar código. Esta excelente herramienta es JUnit.

6.1. ¿Qué son las pruebas unitarias?

Las pruebas unitarias se realizan sobre una clase, para probar su comportamiento de modo aislado, independientemente del resto de clases de la aplicación. Este requisito a veces no se cumple, piensa en el código de una clase que accede a una base de datos, y que la prueba de la clase se base en el resultado que se recupera de la base de datos, resulta imposible comprobar esta clase de modo aislado, aunque existen técnicas (como los *Mock Objects*) que minimizan estas dependencias.

6.1.1. Principios FIRST para el diseño de pruebas unitarias

Cuando se diseñan pruebas unitarias es importante seguir los principios *FIRST*. Cada una de las letras de esta palabra inglesa está relacionada con un concepto. Veámoslos:

Fast : La ejecución del código de pruebas debe ser rápida. Si las pruebas consumen demasiado tiempo acabaremos por no hacerlas.

Independent : Una prueba no puede depender de otras. Cada prueba debe ser unitaria, debe poder realizarse de modo aislado.

Repetable : Las pruebas se deben poder repetir en cualquier momento y la cantidad de veces que sea necesario. El resultado de una prueba debe ser siempre el mismo.

Self-validating : Sólo hay dos posibles resultados de una prueba: «La prueba pasó con éxito» o «La prueba falló».

Timely : Las pruebas han de escribirse en el momento de escribir el código, y no al final de toda la fase de desarrollo [1]

[1] La metodología de desarrollo *Test Driven Development (TDD)* lleva este principio al inicio del proceso de desarrollo de tal modo que las pruebas de código se escriben antes que el propio código que se intenta probar.

6.2. Pruebas unitarias con JUnit

JUnit es una herramienta para realizar pruebas unitarias automatizadas. JUnit está integrada en *Eclipse*, no es necesario descargarse ningún paquete ni instalar un nuevo plug-in para utilizarla. *Eclipse* facilita la creación de pruebas unitarias.

Para mostrar con un ejemplo cómo se escriben pruebas unitarias de código con JUnit vamos a utilizar las clases `ConversorTemperaturas` y `TemperaturaNoValidaException` que vimos en el capítulo anterior.

6.2.1. Creación de clases de prueba

Para crear una clase de prueba en *Eclipse* seleccionamos *File* → *New* → *Other...*, en la ventana de diálogo que se abrirá seleccionamos *Java* → *JUnit* → *JUnit Test Case*, se abrirá una nueva ventana de diálogo, en la parte superior seleccionamos *New JUnit 4 test*, introducimos el nombre para la clase de prueba y su paquete, y por comodidad, en la parte inferior de esta ventana pulsamos *Browse* y seleccionamos la clase que deseamos probar, que en nuestro caso es `ConversorTemperaturas`, y pulsamos *Next*, en la nueva ventana veremos que se nos ofrece la posibilidad de seleccionar los métodos que queremos probar, en nuestro caso vamos a seleccionar `celsiusAFharenheit(double)` y *celsiusAReamur(double)* y pulsamos *Finish*. La clase de prueba que se creará automáticamente será la que se muestra en el Listado 6.1.

Cabe destacar varios puntos de este Listado:

1. Fíjate en el uso del `import static` de la línea 3, es útil para no tener que incluir el nombre de la clase `Assert` cuando utilizamos alguno de sus métodos estáticos, como `fail`. Un `import static` me permite utilizar todos los métodos `static` de una clase sin necesidad de anteponer al método el nombre de la clase [2].

2. Observa que se han creado dos métodos de prueba, una para cada método que seleccionamos sobre la clase a probar, y que la signatura de ambos es **public void** nombreDelMetodo(). Los métodos de prueba deben ser públicos no retornar ningún valor y su lista de argumentos debe estar vacía.

3. Fíjate que sobre cada uno de los métodos de prueba aparece la anotación `@Test` que indica al compilador que es un método de prueba.

4. Por defecto, cada uno de los métodos de prueba tiene una llamada a `fail("Mesaje con descripción.")`.

```
1 package test;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 public class TestConversorTemperaturas {
```

[2] Los `static import` se introdujeron en la versión 5 de Java, y aunque son cómodos de utilizar, el uso de JUnit es un caso, pueden provocar confusión en el programador, ya que al no aparecer el nombre de la clase tendremos la duda de si el método pertenece a la clase actual o a un clase de la que se ha hecho un `static import`. En general, el uso de los `static import` está desaconsejado.

```
8
9  @Test
10 public final void testCelsiusAFherenheit() {
11  fail("Not yet implemented");
12 }
13
14 @Test
15 public final void testCelsiusAReamur() {
16  fail("Not yet implemented");
17 }
18
19 }
```

Listado 6.1: Código generado automáticamente por *Eclipse* para una clase de prueba.

6.2.2. La anotación `@Test`

Como ya se ha dicho, la anotación `@Test` sirve para indicar que un determinado método es un método de prueba. Vamos a escribir el primer código de prueba tal y como se muestra en el Listado 6.2. Fíjate que también hemos añadido **throws** TemperaturaNoValidaException para indicar que no queremos gestionar esta posible excepción en el código del método de prueba.

```
1  @Test
2  public void testCelsiusAFharenheit() throws
       TemperaturaNoValidaException {
3   ConversorTemperaturas conversor = new ConversorTemperaturas();
4   assertEquals(32, conversor.celsiusAFharenheit(0), 0);
5  }
```

Listado 6.2: Un método de prueba.

Lo primero que hacemos en el método de prueba es crear una instancia de la clase `ConversorTemperaturas`, y después utilizar el método assertEquals(valorEsperado, valorObtenido, error). Este método comprueba que la diferencia entre el `valorEsperado` y el `valorObtenido` es menor que `error`. Si es así, se ha pasado la prueba, de lo contrario la prueba falla. En nuestro caso, se está aseverando que la diferencia entre el valor que devuelve el método `celsiusAFharenheit(0)` y el valor `32` es cero. Escribamos el código para la segunda de las pruebas tal y como se muestra en el Listado 6.3.

```
1  @Test
2  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
3   ConversorTemperaturas conversor = new ConversorTemperaturas();
4   assertEquals(0, conversor.celsiusAReamur(0), 0);
5  }
```

Listado 6.3: Un segundo método de prueba.

De modo análogo al primer método de prueba, en este caso estamos aseverando que la diferencia entre el valor que devuelve la llamada al método `celsiusAReamur(0)` y el valor `0` es cero.

Para ejecutar las pruebas desde *Eclipse* pulsa el botón derecho del ratón sobre la clase de prueba y en el menú emergente selecciona la opción *Run As* → *JUnit Test*, verás que se abre una nueva vista con el resultado de la ejecución de las pruebas, que en nuestro caso es *Runs: 2/2 Errors: 0 Failures: 0* que nos

indica que se han realizado 2 pruebas, ninguna de ellas a provocado un error y ninguna de ellas a provocado un fallo.

¿Cual es la diferencia entre un fallo y un error en el contexto de las pruebas unitarias con JUnit?. Un fallo es una aseveración que no se cumple, un error es una excepción durante la ejecución del código. Generemos un fallo de modo artificial para ver el resultado, cambiemos la línea `assertEquals(32, conversor.celsiusAFharenheit(0), 0);` por esta otra `assertEquals(0, conversor.celsiusAFharenheit(0), 0);`, y ejecutemos de nuevo la prueba, en este caso obtendremos un fallo que nos informará que el valor esperado de la prueba era `0` mientras que el valor obtenido es `32.0`.

Añadamos otro método de prueba que genere un error, para ver la diferencia con un fallo, tal y como se muestra en el Listado 6.4.

```
1 public void testTemperaturaNoValidaFharenheit() throws
      TemperaturaNoValidaException {
2  ConversorTemperaturas conversor = new ConversorTemperaturas();
3  conversor.celsiusAFharenheit(-400);
4 }
```

Listado 6.4: Un método de prueba que genera un error.

Al ejecutar de nuevo las pruebas esta vez obtendremos la excepción *La temperatura no puede ser menor que -273ºC*. ¿Y si lo que queremos es precisamente comprobar que se lanza la excepción?, es decir, ¿Y si nuestra prueba pasa precisamente si se genera la excepción? Para ello basta con añadir el atributo `expected=TemperaturaNoValidaException.class` a la anotación `@Test` quedando de este modo `@Test(expected=TemperaturaNoValidaException.class)`. Si ejecutamos de nuevo las pruebas veremos que todas ellas pasan.

Otra técnica que no utiliza el atributo `expected` de la anotación `@Test` para comprobar que se produce una excepción es la mostrada en el Listado 6.5. Esta vez el método está etiquetado únicamente con `@Test`, y detrás de la línea de código que esperamos que produzca la excepción escribimos `fail("Para temperaturas por encima de -273 la prueba debe pasar.")`. Si la excepción se produce al ejecutarse la línea 5, la ejecución de la prueba continuará en el bloque `catch (TemperaturaNoValidaException e)` y la prueba pasará, que es lo que esperamos.

Si no se produce la excepción en la línea 5, se ejecutará la sentencia `fail(...)` y la prueba no pasará, cosa que será indicativa de que algo ha ido mal ya lo que intentamos probar es que la excepción sí que se produce.

```
1 @Test
2 public void testTemperaturaNoValidadFahrenheit() {
3  ConversorTemperaturas conversor = new ConversorTemperaturas();
4  try {
5   conversor.celsiusAFharenheit(-400);
6   fail("Para temperaturas por encima de -273 la prueba debe pasar.");
7  } catch (TemperaturaNoValidaException e) {
8  }
9 }
```

Listado 6.5: Un método de prueba que genera un error.

Este segundo método de prueba de excepciones es el recomendado, ya que es más fácil interpretar qué es lo que se está intentando probar.

6.2.3. Las anotaciones `@Before` y `@After`

Si revisas el código de los tres métodos de prueba anteriores verás que lo primero que hacemos es crear una instancia de la clase `ConversorTemperaturas`. JUnit nos proporciona un mecanismo para extraer el código que se repite en todos los métodos, y que debe ejecutarse antes de cualquiera de ellos, a través de las anotaciones `@Before` y `@After`. Si anotamos un método con `@Before` su código será ejecutado antes de cada uno de los métodos de prueba, si tenemos tres métodos de prueba será ejecutado antes de cada uno de los tres métodos. Por su lado, si anotamos un método con `@After` será ejecutado después de la ejecución de cada uno de los métodos de prueba. Por lo tanto, podemos usar la anotación `@Before` para iniciar todas las infraestructuras necesarias a la ejecución de las pruebas y la anotación `@After` para limpiar estas infraestructuras.

En nuestro caso, la clase de prueba quedaría tal y como se muestra en el Listado 6.6.

```
1 import static org.junit.Assert.*;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import conversor.ConversorTemperaturas;
8 import conversor.TemperaturaNoValidaException;
9
10 public class TestConversorTemperaturas2 {
11  private ConversorTemperaturas conversor;
12
13  @Before
14  public void creaConversorTemperaturas() {
15   conversor = new ConversorTemperaturas();
16  }
17
18  @After
19  public void destruyeCnversorTemperarturas() {
20   conversor = null;
21  }
22
23  @Test
24  public void testCelsiusAFharenheit() throws
        TemperaturaNoValidaException {
25   assertEquals(32, conversor.celsiusAFharenheit(0), 0);
26  }
27
28  @Test
29  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
30   assertEquals(0, conversor.celsiusAReamur(0), 0);
31  }
32
33  @Test(expected=TemperaturaNoValidaException.class)
34  public void testTemperaturaNoValidaFharenheit() throws
        TemperaturaNoValidaException {
35   conversor.celsiusAFharenheit(-400);
36  }
37 }
```

Listado 6.6: Uso de las anotaciones `@Before` y `@After`.

Las anotaciones `@Before` y `@After` las puedes utilizar tantas veces como te sea necesario, puede haber más de un método anotado con alguna de estas anotaciones. Todos los métodos que estén anotados con `@Before` se ejecutarán antes de cada uno de los métodos de prueba y todos los métodos que estén anotados con `@After` se ejecutarán después de cada uno de los métodos de

prueba.

6.2.4. Las anotaciones `@BeforeClass` y `@AfterClass`

Podemos mejorar un poco más nuestra clase de prueba con el uso de dos nuevas etiquetas `@BeforeClass` y `@AfterClass`. Fíjate que la clase que estamos probando `ConversorTemperaturas` no tiene estado, y por lo tanto el resultado de las llamadas a sus métodos es independiente del orden en el que se hagan, por lo que no es necesario crear una instancia nueva antes de cada una de las pruebas, si no que la misma instancia nos sirve para las tres pruebas.

Si anotamos un método de una clase de prueba con `@BeforeClass` ese método se ejecutará una única vez antes de la ejecución de cualquier método de prueba. Por otro lado, si anotamos un método de una clase de prueba con `@AfterClass` el método será ejecutado una única vez después de haberse ejecutado todos los métodos de prueba, tal y como se muestra en el Listado 6.7.

```
1 import static org.junit.Assert.assertEquals;
2
3 import org.junit.AfterClass;
4 import org.junit.BeforeClass;
5 import org.junit.Test;
6
7 import conversor.ConversorTemperaturas;
8 import conversor.TemperaturaNoValidaException;
9
10 public class TestConversorTemperaturas3 {
11  private static ConversorTemperaturas conversor;
12
13  @BeforeClass
14  public static void creaConversorTemperaturas() {
15   conversor = new ConversorTemperaturas();
16  }
17
18  @AfterClass
19  public static void destruyeCnversorTemperarturas() {
20   conversor = null;
21  }
22
23  @Test
24  public void testCelsiusAFharenheit() throws
        TemperaturaNoValidaException {
25   assertEquals(32, conversor.celsiusAFharenheit(0), 0);
26  }
27
28  @Test
29  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
30   assertEquals(0, conversor.celsiusAReamur(0), 0);
31  }
32
33  @Test(expected=TemperaturaNoValidaException.class)
34  public void testTemperaturaNoValidaFharenheit() throws
        TemperaturaNoValidaException {
35   conversor.celsiusAFharenheit(-400);
36  }
37 }
```

Listado 6.7: Uso de las anotaciones `@BeforeClass` y `@AfterClass`.

Fíjate en el importante detalle que aparece en el Listado 6.7, los métodos anotados con `@BeforeClass` y `@AfterClass` deben ser ambos `static` y por lo tanto, los atributos a los que acceden también deben ser `static`, tal y como vimos en 2.6.

6.2.5. Pruebas con batería de datos de entrada

Cada uno de los métodos de prueba de los ejemplos anteriores utiliza un trío de datos, valor esperado, valor real y error para comprobar cada uno de los casos de prueba. Si queremos escribir una nueva prueba para otro trío de valores es tedioso crear un método sólo para él. JUnit proporciona un mecanismo para probar baterías de valores en vez de únicamente tríos aislados.

Lo primero que debemos hacer es anotar la clase de prueba con `@RunWith(Parameterized.class)` indicando que va a ser utilizada para realizar baterías de pruebas. La clase de prueba ha de declarar un atributo por cada uno de los parámetros de la prueba, y un constructor con tantos argumentos como parámetros en cada prueba. Finalmente necesitamos definir un método que devuelva la colección de datos a probar anotado con `@Parameters` . De este modo, cada uno de los métodos de prueba será llamado para cada una de las tuplas de valores de prueba.

En el Listado 6.8 se muestra un ejemplo de clase de prueba para una batería de pruebas sobre la clase `ConversorTemperaturas`.

```
1 import static org.junit.Assert.*;
2
3 import java.util.Arrays;
4 import java.util.Collection;
5
6 import org.junit.AfterClass;
7 import org.junit.BeforeClass;
8 import org.junit.Test;
9 import org.junit.runner.RunWith;
10 import org.junit.runners.Parameterized;
11 import org.junit.runners.Parameterized.Parameters;
12
13 import conversor.ConversorTemperaturas;
14 import conversor.TemperaturaNoValidaException;
15
16 @RunWith(Parameterized.class)
17 public class TestConversorTemperaturas4 {
18  private double celsius;
19  private double fharenheit;
20  private double reamur;
21  private double error;
22  private static ConversorTemperaturas conversor;
23
24  public TestConversorTemperaturas4(double celsius, double fharenheit,
        double reamur, double error) {
25   this.celsius = celsius;
26   this.fharenheit = fharenheit;
27   this.reamur = reamur;
28   this.error = error;
29  }
30
31  @Parameters
32  public static Collection<Object[]> datos() {
33   return Arrays.asList(new Object[][]{
34     {0.0, 32.0, 0.0, 0.0}, // {celsius, fharenheit, reamur, error}
35     {15, 59.0, 12.0, 0.0},
36     {30, 86.0, 24.0, 0.0},
37     {50, 122.0, 40.0, 0.0},
38     {90, 194.0, 72.0, 0.0}
39   });
40  }
41
42  @BeforeClass
43  public static void iniciaConversor() {
44   conversor = new ConversorTemperaturas();
45  }
46
47  @AfterClass
```

```
48  public static void eliminaConversor() {
49   conversor = null;
50  }
51
52  @Test
53  public void testCelsiusAFharenheit() throws
        TemperaturaNoValidaException {
54   assertEquals(fharenheit, conversor.celsiusAFharenheit(celsius), error)
         ;
55  }
56
57  @Test
58  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
59   assertEquals(reamur, conversor.celsiusAReamur(celsius), error);
60  }
61 }
```

Listado 6.8: Ejemplo de definición de una clase que realiza una batería de pruebas.

De modo resumido, estos son los pasos para definir una clase que ejecuta baterías de pruebas:

1. Anotar la clase de prueba con `@RunWith(Parameterized.class)`.
2. Declarar un atributo en la clase por cada parámetro de prueba.
3. Definir un constructor con tantos argumentos como parámetros de prueba.
4. Definir un método que devuelva una colección con todas las tuplas de prueba, y anotarlo con `Parameters`.

Internamente y de modo esquemático, lo que JUnit hace en el caso de las baterías de pruebas es crear una instancia de la clase de prueba a partir del constructor con tantos argumentos como parámetros en cada una de las tuplas de prueba, y seguidamente llama a cada uno de los métodos de prueba.

6.2.6. Ejecutar varias clases de prueba. Test Suites

Lo común, como hemos visto, es tener varias clases de prueba ya que a veces no tiene sentido una única clase donde se realicen todas las pruebas. Piensa por ejemplo en las pruebas parametrizadas, quizás tengas algún caso de prueba sobre el que no tenga sentido realizar pruebas parametrizadas, como por ejemplo comprobar que se produce una excepción.

Por lo tanto, si tenemos varias clases de prueba, ¿Cómo podemos ejecutar todas las pruebas, o al menos algunas de ellas sin tener que ejecutarla cada clase de modo independiente? Para ello existe un mecanismo en JUnit llamado *Test Suites*, que no son más que agrupaciones de clases de prueba que se ejecutan una tras otra.

Básicamente lo que hacemos es anotar una clase para que JUnit la reconozca como una suite de pruebas y con otra anotación añadimos a esta clase todas las clases de prueba que nos interese ejecutar, tal y como se muestra en el Listado 6.9.

```
1 import org.junit.runner.RunWith;
2 import org.junit.runners.Suite;
3 import org.junit.runners.Suite.SuiteClasses;
4
```

```
5 @RunWith(Suite.class)
6 @SuiteClasses({
7  TestConversorTemperaturas.class,
8  TestConversorTemperaturas2.class,
9  TestConversorTemperaturas3.class,
10  TestConversorTemperaturas4.class
11 })
12 public class AllTests {
13 }
```

Listado 6.9: Ejemplo de una suite de pruebas *Test Suite*.

De este modo bien podemos ejecutar una única clase de prueba para probar el funcionamiento correcto de una clase en particular de todas las que forman nuestro proyecto, o bien podemos ejecutar toda la *suite* de clases de prueba para probar todo nuestro proyecto.

6.3. Cobertura de las pruebas

Una duda que nos surge al realizar pruebas unitarias es la cantidad de líneas de código que han cubierto las pruebas, ¿Ha quedado algún fragmento de código que no se ha ejecutado ni una sola vez para todas las pruebas? ¿Cómo podemos saberlo?.

Lo ideal es que nuestras pruebas cubran el 100 % del código de la clase que estamos probando. Pero no caigas en el engaño de pensar que por cubrir con pruebas el 100 % del código estás cubriendo todos los casos posibles de la ejecución de ese código, ni mucho menos. Puedes cubrir el 100 % de la ejecución del código con casos triviales que nunca fallarán y no sacarán a la luz los posibles errores de tu código.

Para que veas con claridad la diferencia entre cobertura de código y pruebas exhaustivas el Listado 6.10 te muestra un método a probar y un par de métodos de prueba. Los métodos de prueba cubren el 100 % del código del método que se está probando pero, ¿Qué pasa si alguna de las referencias que se pasan al método de prueba es `null`? Evidentemente el método que se está probando contiene un error que no será descubierto por las pruebas aunque estas estén cubriendo el 100 % del código.

```
1 // Método que se va a probar
2 public int quienEsMayor(Persona primera, Persona segunda) {
3  if(primera.edad > segunda.edad) return -1;
4  if(primera.edad < segunda.edad) return 1;
5  else return 0;
6 }
7
8 // Tres métodos de prueba
9 @Test
10 public void masViejoElPrimero() {
11  Persona primera = new Persona();
12  primera.setEdad(100);
13  Persona segunda = new Persona();
14  segunda.setEdad(50);
15  assertEquals(-1, quienEsMayor(primera, segunda), 0);
16 }
17
18 @Test
19 public void masViejoElSegundo() {
20  Persona primera = new Persona();
21  primera.setEdad(50);
22  Persona segunda = new Persona();
23  segunda.setEdad(100);
```

```
24  assertEquals(1, quienEsMayor(primera, segunda), 0);
25 }
26
27 @Test
28 public void mismaEdad() {
29  Persona primera = new Persona();
30  primera.setEdad(50);
31  Persona segunda = new Persona();
32  segunda.setEdad(50);
33  assertEquals(0, quienEsMayor(primera, segunda), 0);
34 }
```

Listado 6.10: Los tres métodos cubren el 100 % del código del método que se está probando, pero este método contiene un error ya que no se comprueba que las referencias sean distintas de `null`

Existe modelos teóricos que dan las pautas a seguir para garantizar que las pruebas son exhaustivas, de modo que se contemplan todos los posibles casos de fallo de nuestro código. La referencia [5] presenta de modo exhaustivo las pruebas de código que se han de realizar para garantizar que se cubre el 100 % de los posibles caminos de ejecución.

6.3.1. EclEmma y su plug-in para Eclipse

Afortunadamente existen excelentes herramientas que miden, de modo automático, la cobertura de nuestras pruebas unitarias. Una de esas herramientas, aunque no la única es *Ecl-Emma* de la que existe un *plug-in* para *Eclipse* y cuya página web es `http://www.eclemma.org/`. Para instalar este *plug-in* basta seguir los mismo pasos que se mostraron en la Sección 4.5, pero siendo la dirección del *plug-in* la que se indica en la página web de la herramienta.

Una vez instalado este *plugin* te aparecerá un nuevo botón en *Eclipse* a la izquierda del botón ejecutar. Si pulsas este botón cuando está seleccionada una clase de prueba, se abrirá una nueva vista de nombre *Coverage* en la que se te mostrará todos los resultados de la cobertura de la prueba, y lo que es de gran ayuda, de cada una de las líneas de código de la clase que se está probando se coloreará su fondo en *verde*, si la línea ha sido cubierta completamente por la prueba; *amarilla*, si ha sido cubierta sólo parcialmente; o *roja*, si no ha sido cubierta.

En la vista *Coverage* se muestra para cada clase probada, una tabla con el porcentaje de líneas de código cubiertas por la prueba.

Sin duda, la herramienta *Ecl-Emma* y su *plugin* para *Eclipse* son excelentes herramientas que contribuyen a aumentar la calidad de nuestro código.

Lecturas recomendadas.

- Un excelente libro, de autores españoles, donde se trata de un modo completo las pruebas de software es la referencia [5]. De lectura obligada si se quiere poner en práctica las pruebas de software.

- Otra excelente referencia de autores españoles es [6]. En la primera parte de este libro se expone con claridad los principios del Diseño de Software Dirigido por Pruebas. En la segunda parte del libro se aplica lo expuesto en la primera parte al desarrollo de una aplicación de ejemplo. Aunque los

lenguajes de programación que muestran los ejemplos con .Net y Python, la aplicación a Java con JUnit es directa.

- El capítulo 10 de [13] presenta las cómo realizar pruebas unitarias con JUnit, y en el capítulo 12 muestra como trabajar con *Cobertura* para analizar el grado de cobertura de nuestras pruebas. *Cobertura* es otra excelente herramienta para el análisis de cobertura de las pruebas unitarias.

- Otro excelente título que debería figurar en todos las estanterías de un buen desarrollador es *Clean code* de Robert C. Martin, también conocido como *uncle Bob*. En el capítulo 9 de esta referencia [10] está dedicado a las buenas prácticas en el diseño de test unitarios de prueba.

Capítulo 7

Entrada y Salida

Contenidos

7.1. Flujos (*Streams*) 106
7.2. Flujos de bytes 107
7.3. Flujos de caracteres 108
7.4. Conexión entre flujos de bytes y de caracteres . . 109
7.5. El sistema de ficheros y flujos a ficheros 110
7.5.1. El sistema de ficheros 110
7.5.2. Flujos a ficheros 110
7.5.2.1. Flujos de bytes a ficheros 110
7.5.2.2. Flujos de caracteres a ficheros 111
7.6. Serialización 112

Introducción

Ya conocemos como definir clases que contienen datos y métodos que nos permiten trabajar con las instancias de estas clases. En muchos casos, una vez que hemos trabajado con nuestros datos nos interesa almacenarlos de manera permanente, de tal modo que sea posible recuperar nuestros datos más tarde para seguir trabajando sobre ellos.

Todo lenguaje de programación proporciona una serie de mecanismos para realizar operaciones de entrada y salida de datos. Decimos que los datos son de entrada cuando llegan a nuestra aplicación desde una fuente de datos, y que son de salida cuando nuestra aplicación envía datos a algún sumidero de datos.

El lenguaje de programación Java nos proporciona un paquete, con una gran cantidad de clases, para poder realizar entrada/salida en nuestras aplicaciones. Verás, que las operaciones de entrada/salida son susceptibles de lanzar gran cantidad de excepciones que vamos a tener que gestionar tal y como vimos en el capítulo 5.

La potencia de la aproximación de Java a las operaciones de entrada/salida es que Java utiliza un concepto transversal con independencia del dispositivo sobre el que se trabaja. Independientemente de si la salida es hacia un fichero,

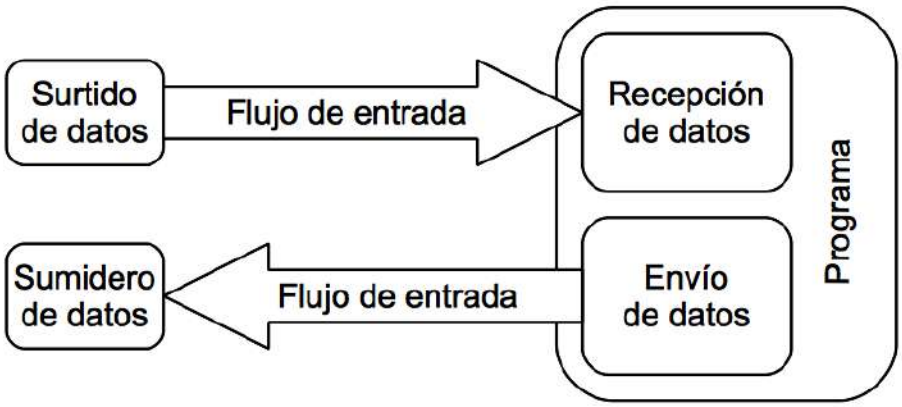

Figura 7.1: El concepto de flujo representado gráficamente.

un `Socket` o una conexión de Internet, el mecanismo de entrada/salida es el mismo: el uso de *Flujos (Streams)*.

7.1. Flujos (*Streams*)

Los flujos en Java son los canales por donde transita la información. Los flujos pueden ser de entrada, de salida, o tener ambas direcciones. Utilizaremos flujos de entrada cuando a nuestras aplicaciones lleguen datos, es decir, cuando queramos leer datos que nos llegan desde alguna fuente de datos. Por el contrario, utilizaremos flujos de salida cuando nuestras aplicaciones quieran enviar datos a algún sumidero de datos.

La potencia de los flujos está relacionada con su independencia de los dispositivos de entrada/salida a los que se estén conectando. Desde el punto de vista de nuestro código, no importa que el dispositivo de salida sea una consola en pantalla, un Socket, o un fichero en nuestro disco duro, el mecanismo de salida siempre es el mismo. Por otro lado, no importa que el dispositivo de entrada sea el teclado, una conexión a una URL, o un fichero en nuestro disco duro, el mecanismo de entrada siempre es el mismo. Las operaciones de entrada/salida en Java siempre se realizan a través de flujos que son independientes de las fuentes o sumideros de datos. En la Figura 7.1 se muestra gráficamente el concepto de flujo.

En Java existen dos grandes categorías de flujos, cada una de ellas con sus propias clases para realizar operaciones de entrada salida: los flujos de bytes y los flujos de caracteres. Utilizaremos unos u otros para realizar operaciones de entrada/salida dependiendo de la naturaleza de los datos que recibamos desde una fuente de datos o enviemos a un sumidero de datos.

En las siguientes secciones se va a presentar una gran cantidad de nuevas clases, lo que implica que vas a ver muchos nombres de clase nuevos. Al principio puede parecer abrumador, pero presta atención al nombre de las clases y verás que es muy significativo, veremos los detalles en la nominación de las clases en las siguientes secciones. Por otro lado, también verás que existe simetría entre los nombres de las clases que realizan operaciones de lectura y las que realizan

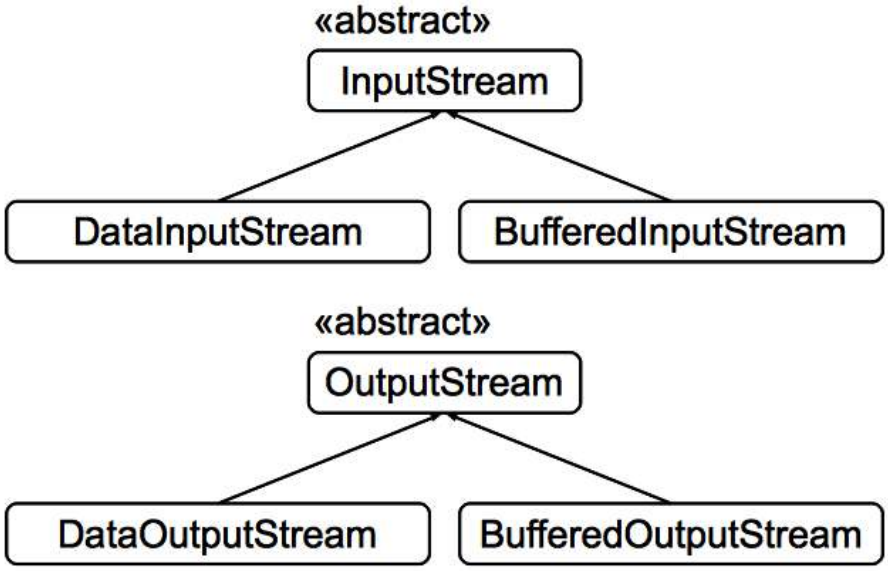

Figura 7.2: Parte de la jerarquía de clases para flujos de bytes.

operaciones de escritura. Y finalmente, también verás que existe simetría en el nombre de las clases que correspondes a flujos de bytes y los nombres de las clases que corresponden a flujos de caracteres. Esta doble simetría y el criterio para nominar a las clases te resultará de gran ayuda para reconocer cual es el cometido de una clase simplemente a través de su nombre.

7.2. Flujos de bytes

Los flujos de bytes nos permiten leer bytes desde una fuente de datos o escribir bytes hacia un sumidero de datos, es decir nos permiten la lectura/escritura de datos binarios. Las clases que nos permiten leer/escribir sobre flujos de bytes existen en Java desde las primeras versiones del lenguaje, y por ello, dispositivos de entrada como el teclado, o dispositivos de salida como una consola en pantalla son ambos flujos de bytes, aunque lo que finalmente se lee o escriba a ellos sean caracteres.

Existe simetría en el modo de nombrar a las clases que realizan operaciones de lectura sobre flujos de bytes y las que realizan operaciones de escritura. Si la operación es de lectura, el nombre de la clase contendrá la palabra *Input*, si el flujo es de escritura, la clase contendrá la palabra *Output*.

Todas las clases que realizan operaciones de lectura de bytes extienden a la clase `abstract InputStrem`, por su lado, todas las clases que realizan operaciones de escritura de bytes extienden a la clase `abstract OutputStrem`. Fíjate que ambas clases son `abstract` y por lo tanto no se pueden instanciar directamente. En la Figura 7.2 se muestra algunas clases de la jerarquía de flujos de bytes.

La clase `DataInputStream` permite abrir un fichero para leer tipos de datos primitivos, así por ejemplo, esta clase proporciona el método `float readFloat()`

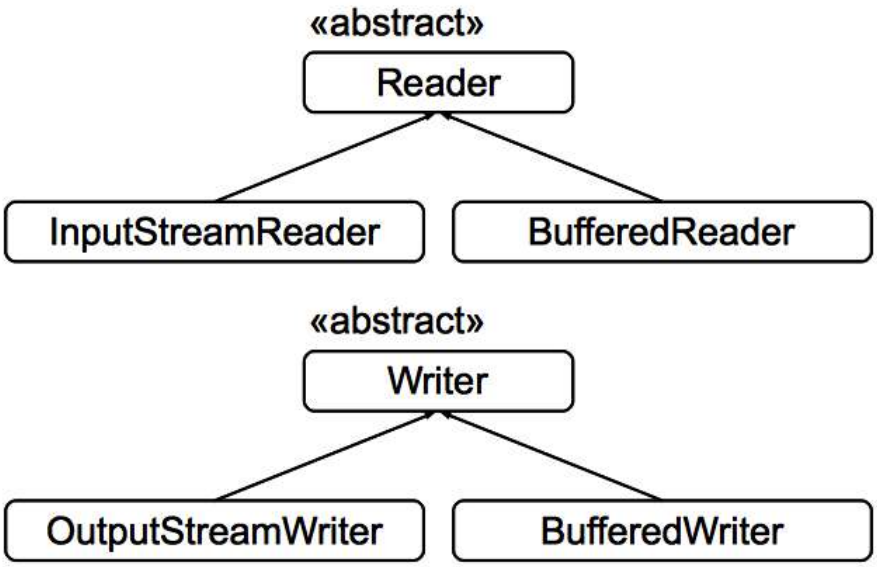

Figura 7.3: Parte de la jerarquía de clases para flujos de caracteres.

que devuelve un número real de precisión simple leído desde un flujo, y `boolean readBoolean()` que devuelve un booleano leído desde un flujo de bytes.

De modo análogo, la clase `DataOutputStream` permite abrir un flujo para escribir en él tipos de datos primitivos, y en este caso contamos con métodos como `void writeFloat(float f)` para escribir en un flujo de byte un número real de precisión sencilla y `final void writeBoolean(boolean b)` para escribir datos booleanos sobre un flujo de bytes.

Las clases `BufferedInputStream` y `BufferedOutputStream` efectúan lectura y escritura de bytes desde un flujo utilizando una memoria intermedia (*buffer*) con el objeto de acelerar el proceso.

7.3. Flujos de caracteres

La Figura 7.3 muestra una pequeña parte de la jerarquía de clases existente en Java para los flujos de caracteres. Fíjate en la analogía con la Figura 7.2. De nuevo, tenemos un par de clases abstractas, `abstract Reader` en el caso de lectura de flujos de caracteres y `abstract Writer` en el caso de escritura.

De nuevo, y también de modo análogo a los flujos de bytes, disponemos de dos subclases que proporcionan una memoria intermedia para mejorar el rendimiento del proceso de lectura/escritura con flujos de caracteres. Estas clases son `BufferedReader` que nos proporciona un flujo de lectura de caracteres con buffer, y `BufferedWriter` que nos proporciona un flujo de escritura de caracteres con buffer.

En el caso de la clase `BufferedReader`, esta clase cuenta con el método `String readLine()` que nos permite leer cadenas de caracteres.

7.4. Conexión entre flujos de bytes y de caracteres

Como acabas de ver en las secciones anteriores, para un determinado tipo de flujo existe varias clases definidas en el árbol de jerarquía, de modo que las clases hijas van añadiendo nueva funcionalidad sobre la que proporciona la clase padre. Tomando como ejemplo los flujos de entrada de caracteres, la clase `Reader` proporciona métodos para leer caracter a caracter desde alguna fuente de datos, y su clase hija `BufferedReader` añade un buffer intermedio en la lectura de modo que podemos leer líneas de caracteres (`String`) desde la fuente de datos. Por otro lado los constructores de `BufferedReader` son:

- BufferedReader(Reader in)
- BufferedReader(Reader in, int sz)

ambos necesitan una referencia a `Reader` para invocarse. La nueva instancia de `BufferedReader` se construye *envolviendo* a la instancia de `Reader`.

Y esta es la idea clave del trabajo con flujos en Java, obtener de algún modo flujos de tipo básico e ir construyendo sobre ellos nuevas instancias hasta llegar a un flujo que nos proporcione la funcionalidad que necesitamos.

Por otro lado, el primer tipo de flujos que se introdujo en Java desde la versión 1.0 fueron los flujos de bytes, y desde la versión 1.1 aparecieron los flujos de caracteres. No es por lo tanto de extrañar que tanto el teclado, como la pantalla/consola sean flujos de bytes.

Con todo esto, la pregunta que nos surge es ¿Cómo se conectan los flujos de caracteres con los flujos de bytes para que, por ejemplo, podamos leer cadenas de caracteres directamente desde el teclado? La respuesta a esta pregunta es que Java proporciona clases de conexión entre ambos tipos de flujos:

`InputStreamReader` toma una instancia de `InputStream`, flujo de entrada de bytes, y sobre él que crear un flujo de lectura de caracteres.

`OutputStreamWriter` toma una instancia de `OutputStream`, flujo de salida de bytes, y sobre él que crea un flujo de escritura de caracteres.

El Listado 7.1 muestra un ejemplo de uso de esta técnica para leer cadenas de caracteres desde teclado:

```
1 InputStream is = System.in; // El teclado es Java es System.in
2 InputStreamReader isr = new InputStreamReader(is); // Lo decoramos como
      un flujo de caracteres
3 BufferedReader br = new BufferedReader(isr); // Lo decoramos con un
      flujo con memoria intermedia
4 String linea = br.readLine(); // Ya podemos leer cadenas de texto desde
      el teclado.
```

Listado 7.1: Técnica para leer caracteres desde teclado

En la línea 1 del Listado 7.1 simplemente definimos la referencia `is` hacia el teclado, que es el flujo de entrada de byte desde el que queremos leer. En la línea 2, convertimos el flujo de entrada bytes a un flujo de entrada de caracteres con la ayuda de la clase `InputStreamReader`, en este momento ya podríamos leer caracteres desde el teclado, pero es más eficiente utilizar una memoria intermedia.

En la línea 3 estamos creando una instancia de la clase `BufferedReader` sobre el flujo de entrada de caracteres (`InputStreamReader`), para poder finalmente leer cadenas de caracteres con la ayuda del método `String readLine()`, tal y como se muestra en la línea 4 [1].

7.5. El sistema de ficheros y flujos a ficheros

Un caso particular de fuente y sumidero de datos son los ficheros. Desde nuestros programas podemos leer los datos contenidos en un fichero, sean estos datos de tipo binarios o caracteres, y podemos escribir datos a un fichero, sea estos datos de tipo binarios o caracteres.

Como el acceso para realizar entrada/salida es independiente del dispositivo, lo que necesitaremos en este caso es algún medio para acceder al sistema de ficheros. Para ello Java nos proporciona la clase `File`. Esta clase nos da acceso al sistema de ficheros y sobre esta clase podemos construir flujos de entrada/salida par tipos de datos tanto binarios como caracteres.

7.5.1. El sistema de ficheros

La clase `File` nos da acceso al sistema de ficheros, con independencia del sistema operativo sobre el que se ejecute. Gracias a esta clase, podemos obtener información tal como la lista de ficheros o directorios bajo un directorio dado, o comprobar si el camino con el que se construye la instancia de `File` hace referencia a un fichero o a un directorio.

7.5.2. Flujos a ficheros

Los flujos a ficheros nos permiten acceder a la información contenida en los ficheros de nuestro sistema con el fin de leer desde ellos o escribir información hacia ellos. De nuevo, podemos distinguir dos tipos de flujos a ficheros dependiendo de si la información contenida en ellos es de tipo binario o caracteres.

7.5.2.1. Flujos de bytes a ficheros

La clase `FileInputStream` nos permite crear un flujo de lectura hacia un fichero para leer desde él datos de tipo binario. Podemos instanciar esta clase a partir de una referencia a `File` o bien a partir de un `String` que represente el camino hasta el fichero.

De modo análogo, la clase `FileOutputStream` nos permite crear un flujo de escritura hacia un fichero para escribir en él datos de tipo binario. Podemos instanciar esta clase también a partir de una referencia a `File` o bien a partir de un `String` que represente el camino hasta el fichero. En el momento de la creación del flujo podemos indicar si queremos conservar el posible contenido del fichero en el momento de la creación del flujo a través de un argumento de tipo booleano en el constructor.

[1] En el Capítulo 8 se mostrará un clase de utilidad `Scanner` que facilita enormemente la lectura de datos desde teclado, y en general desde cualquier flujo de entrada. No obstante lo que aquí se ha mostrado es un ejemplo del mecanismo general de conversión entre flujos de caracteres y flujos de bytes

7.5.2.2. Flujos de caracteres a ficheros

La clase `FileReader` nos permite crear un flujo de lectura hacia un fichero, para leer desde él datos de tipo caracter. De modo análogo al caso de `FileInputStream`, podemos instanciar esta clase s partir de una referencia a `File` o bien a partir de un `String` que represente el camino hasta el fichero.

Finalmente, la clase `FileWriter` nos permite crear un flujo de escritura de caracteres hacia un fichero para escribir en él datos de tipo caracter. De modo análogo a la clase `FileOutputStream`, podemos instanciar esta clase a partir de una referencia `File`, de un `String` que indique el camino al fichero, e indicar en el momento de la creación si el fichero conserva o no el posible contenido.

Como ejemplo del manejo de ficheros, el Listado 7.2 muestra cómo abrir un flujo a un fichero de caracteres para leer desde él línea a línea su contenido y mostrarlo por consola.

```
1 import java.io.BufferedReader;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4 import java.io.FileReader;
5 import java.io.IOException;
6
7 public class LecturaFlujoTexto {
8  public LecturaFlujoTexto() {
9   super();
10  }
11
12  private void ejecuta(String camino) {
13   File fichero = new File(camino);
14   FileReader flujoLectura;
15   BufferedReader flujoBuffer = null;
16   try {
17    try {
18     flujoLectura = new FileReader(fichero);
19     flujoBuffer = new BufferedReader(flujoLectura);
20     String linea;
21     while((linea = flujoBuffer.readLine()) != null) {
22      System.out.println(linea);
23     }
24    } finally {
25     if(flujoBuffer != null)
26      flujoBuffer.close();
27    }
28   } catch (FileNotFoundException e) {
29    e.printStackTrace();
30   } catch (IOException e) {
31    e.printStackTrace();
32   }
33  }
34
35  public static void main(String[] args) {
36   new LecturaFlujoTexto().ejecuta(args[0]);
37  }
38 }
```

Listado 7.2: Lectura desde un flujo de texto hacia un fichero

Como curiosidad del Listado 7.2, fíjate que hay un bloque `try{...} finally{...}` en las líneas 17-27 que está incluido dentro de otro bloque `try{...}`. El uso de estos bucles anidados facilita la lectura de código, y su ejecución es la siguiente: si se produce alguna excepción durante el trabajo con el fichero (líneas de código 18-23), se ejecutará el bloque `finally{...}` con lo que se cerrará el fichero, y la excepción se propagará al bloque `try{...}` externo, que es quien tiene los bloques `catch{...}` para atrapar a cada una de

las excepciones posibles. Este modo de codificación es ampliamente utilizado y conviene que lo incorpores como técnica de escritura de tu propio código.

7.6. Serialización

La *Serialización* es el mecanismo por el cual Java es capaz de convertir un objeto en una secuencia de bytes. De este modo, podemos crear un flujo a partir de la secuencia de bytes que representa al objeto para escribirlo en un fichero o enviarlo a través de un `Socket` por ejemplo.

El concepto de serialización es extremadamente potente, si somos capaces de obtener una secuencia de bytes del objeto y sobre ella crear un flujo, podemos enviar el objeto a cualquier dispositivo que lo acepte. Y de modo análogo, podríamos conectarnos a una fuente de datos a través de un flujo de entrada y obtener objetos desde ella. De hecho, esta técnica es tan potente que es la pieza sobre la que descansan otras tecnologías Java como la invocación remota de método (*Remote Method Innvocation - RMI*), y gran parte de las tecnologías Java en su edición *Enterprise* [2].

En esta sección vamos a presentar la serialización y cómo utilizarla para almacenar un objeto en un fichero, y en el Capítulo 15 veremos cómo utilizar la serialización para enviar y recibir objetos a través de un Socket.

Para indicar que un objeto es *Serializable* debe implementar la interface `Serializable`. Esta interface no declara ningún método, es únicamente una marca semántica para que Java sepa que en algún momento podemos querer serializar el objeto. Cuando el objeto serializable se convierta en una secuencia de bytes, esta secuencia, además de incluir el valor de los atributos de la instancia incluye más información, como la clase de la instancia y un número de serie de la clase, para poder hacer un control de versiones de la clase. Por convención, este número de control se declara como `private static final long serialVersionUID`. El valor de este atributo lo podemos fijar nosotros mismos, pero eso no garantiza que otro programador pueda utilizar el mismo número de serie para una clase completamente distinta. Java nos proporciona una herramienta, integrada en el JDK, que genera números de versión a partir de la clase compilada, esta herramienta es `serialver`, su uso es `serialver [-classpath] nombreClase` y un ejemplo de uso es:

```
$ serialver -classpath . serializacion.Persona
serializacion.Persona:    static final long serialVersionUID =
7360368493593648647L;
```

En este caso la herramienta`serialver` ha generado el número de serie `7360368493593648647L` para la clase `Persona`. Y es muy improbable que `serialver` genere el mismo número de versión para cualquier otra clase, lo que nos garantiza que este número de versión es único e identifica a la clase `Persona`.

[2] Java 2 Enterprise Edition está enfocada al desarrollo de aplicaciones de servidor. Son aplicaciones que se no se ejecutan en las máquinas de un cliente, si no en un servidor remoto. Usualmente, el ciente accede al servidor a través de un navegador web, aunque no es la única opción posible

Eclipse es capaz de invocar a `serialver` de manera transparente. Si nuestra nueva clase implementa la `interface Serializable` y olvidamos incluir el atributo `serialVersionUID`, *Eclipse* nos mostrará un aviso. Si corregimos el aviso que nos da *Eclipse* seleccionando la opción `Add generated serial verion ID`, se añadirá el número de serie obtenido con `serialver`.

El Listado 7.3 muestra la clase `Persona` que implementa la `interface Serializable` y a la que se le ha añadido el número de versión generado con `serialver`.

```
1 package serializacion;
2
3 import java.io.Serializable;
4
5 public class Persona implements Serializable{
6  private static final long serialVersionUID = 7360368493593648647L;
7  String nombre;
8  String apellidos;
9  String telefono;
10
11  Persona() { }
12
13  Persona(String nombre, String apellidos, String telefono) {
14   this.nombre = nombre;
15   this.apellidos = apellidos;
16   this.telefono = telefono;
17  }
18
19  String getNombre() {
20   return nombre;
21  }
22
23  String getApellidos() {
24   return apellidos;
25  }
26
27  String getTelefono() {
28   return telefono;
29  }
30
31  @Override
32  public String toString() {
33   return "Persona [apellidos=" + apellidos + ", nombre=" + nombre
34     + ", telefono=" + telefono + "]";
35  }
36 }
```

Listado 7.3: La clase `Persona` lista para ser serializada

El Listado 7.4 muestra un ejemplo de como serializar esta clase y la secuencia de bytes, y cómo almacenar la secuencia de bytes obtenida en un fichero para su recuperación posterior.

```
1 package serializacion;
2
3 import java.io.FileNotFoundException;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6 import java.io.ObjectOutput;
7 import java.io.ObjectOutputStream;
8 import java.io.OutputStream;
9
10 public class EscritorPersona {
11  public EscritorPersona() {
12   super();
13  }
14
15  private void ejecuta(String fichero) {
```

```
16   OutputStream streamEscritura;
17   ObjectOutput streamEscrituraPersona = null;
18   try {
19    try {
20    streamEscritura = new FileOutputStream(fichero);
21    streamEscrituraPersona = new ObjectOutputStream(streamEscritura);
22    streamEscrituraPersona.writeObject(new Persona("James", "Gosling", "
          555 123 456"));
23    } finally {
24     if(streamEscrituraPersona != null) streamEscrituraPersona.close();
25    }
26   } catch (FileNotFoundException e) {
27    e.printStackTrace();
28   } catch (IOException e) {
29    e.printStackTrace();
30   }
31  }
32
33  public static void main(String[] args) {
34   new EscritorPersona().ejecuta(args[0]);
35  }
36
37 }
```

Listado 7.4: Serialización de la clase `Persona` hacia un fichero de bytes

Si ejecutamos el programa Java anterior, introduciendo por línea de instrucciones como nombre de fichero `persona.ser` [3] se creará un fichero de tipo binario cuyo contenido es una instancia de la clase `Persona` cuyos atributos tienen los valores asignados.

El Listado 7.5 muestra un ejemplo completo de cómo leer un objeto serializado almacenado en un fichero de tipo binario.

```
1 package serializacion;
2
3 import java.io.FileInputStream;
4 import java.io.FileNotFoundException;
5 import java.io.IOException;
6 import java.io.InputStream;
7 import java.io.ObjectInput;
8 import java.io.ObjectInputStream;
9
10 public class LectorPersona {
11  public LectorPersona() {
12   super();
13  }
14
15  private void ejecuta(String fichero) {
16   InputStream streamLectura;
17   ObjectInput streamLecturaPersona = null;
18   try {
19    try {
20     streamLectura = new FileInputStream(fichero);
21     streamLecturaPersona = new ObjectInputStream(streamLectura);
22     Persona persona = (Persona)streamLecturaPersona.readObject();
23     System.out.println(persona);
24    } finally {
25     if(streamLecturaPersona != null) streamLecturaPersona.close();
26    }
27   } catch(FileNotFoundException e) {
28    e.printStackTrace();
29   } catch(IOException e) {
30    e.printStackTrace();
31   } catch(ClassNotFoundException e) {
32    e.printStackTrace();
33   }
```

[3] Por convención, para los ficheros que contienen datos de objetos serializados se utiliza la extensión `.ser`

```
34  }
35
36  public static void main(String[] args) {
37   new LectorPersona().ejecuta(args[0]);
38  }
39
40 }
```

Listado 7.5: Des-serialización de la clase `Persona` desde un fichero de bytes

Fíjate cómo, de nuevo, se han utilizado bloques `try{...}` anidados para facilitar la gestión de excepciones.

En el capítulo 15 se mostrará otro ejemplo de serialización de objetos pero esta vez se enviará la secuencia de bytes que representa al objeto a través de los flujos que proporciona un `Socket`, con lo que podremos escribir un servidor capaz de enviarnos, a través de una red de comunicación de datos, objetos serializados y podremos recuperarlos en un cliente.

Ejercicios

1. Amplia la funcionalidad de la aplicación de la agenda, para que sea posible almacenar los datos de los contactos en un fichero de texto para su posterior recuperación. Escribe tanto el código de escritura como de lectura.

2. Sigue ampliando la funcionalidad de tu aplicación de la agenda para que sea posible serializarla a un fichero de tipo binario para su posterior recuperación. Escribe tanto el código de escritura como el de lectura.

Lecturas recomendadas

- La referencia [2] dedica todo el Capítulo 15 al estudio del mecanismo de entrada/salida en Java.
- La referencia [3] dedica también todo el Capítulo 14 al estudio del mecanismo de entrada/salida en Java. En particular las secciones dedicadas a la serialización de objetos son muy interesantes.

Capítulo 8

Algunas clases de utilidad del paquete estándar

Contenidos

8.1. La clase `Scanner` 118
8.2. Trabajo con cadenas de caracteres 120
8.2.1. La clase `String` 120
8.2.2. Las clases `StringBuffer` y `StringBuilder` 121
8.3. Clases recubridoras 122
8.4. Colecciones 124
8.5. Trabajo con fechas 128
8.5.1. La clase `Date` 128
8.5.2. Las clases `Calendar` y `GregorianCalendar` 129
8.6. Matemáticas 129
8.6.1. La clase `Math` 129
8.6.2. La clase `Random` 130

Introducción

La edición estándar de Java proporciona una amplísima colección de clases ya definidas. Estas clases ya definidas son de gran utilidad, a medida que como programadores vamos conociendo nuevas clases de esta colección nuestra productividad aumenta.

En este Capítulo presentamos un pequeño grupo de clases de gran utilidad, aunque existen otras muchas tan útiles como las aquí presentadas. Obviamente, por cuestión de espacio, sólo presentamos las que creemos más interesantes llegados a este punto.

8.1. La clase `Scanner`

En el Capítulo 7 comentamos que en Java tanto la consola como el teclado son flujos de bytes. Esto se debe a que en la versión 1.0 de Java no existían flujos de caracteres. Para guardar la compatibilidad hacia atrás, tanto el teclado como la consola siguen siendo flujos de caracteres en las versiones actuales de Java.

Como finalmente lo que nos interesa leer desde el teclado son caracteres, tenemos que aplicar la técnica del recubrimiento que vimos en el Capítulo 7 para conseguir leer cadenas de caracteres. El uso de la clase de utilidad `Scanner` nos oculta los flujos y nos permite leer directamente de flujos, en particular desde el teclado.

La clase `Scanner` está presente en Java desde la versión 5. Esta clase permite analizar una cadena de texto utilizando para ello expresiones regulares [1]. Las cadenas se pueden proporcionar directamente en el momento de crear la instancia de clase `Scanner` o a través de un flujo.

Supongamos que tenemos un fichero de texto que almacena datos de una agenda de teléfonos con el formato `Persona: Nombre: Apellidos: Telefono`, un ejemplo de contenido del fichero es[2]:

```
Persona: LUISA: GARCIA MORENO: 313372295
Persona: ROSARIO: GONZALEZ ESTEBAN: 560248322
Persona: MANUEL: SANZ GARCIA: 571365702
Persona: FRANCISCO: VAZQUEZ FERRER: 690109651
Persona: VICTOR: MUÑOZ LOPEZ: 500661266
```

El Listado 8.1 muestra un ejemplo de uso de la clase `Scanner` para leer línea a línea de este fichero. El método `hasNext()` nos sirve para comprobar si hay más elementos a devolver.

```
1  package utilidad;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.util.Scanner;
6
7  public final class UtilidadScanner {
8   private static final String FICHERO = "agenda.txt";
9   private UtilidadScanner() {
10   super();
11  }
12
13  private void ejecuta() {
14   try {
15    Scanner scanner = new Scanner(new File(FICHERO));
16    while(scanner.hasNext())
17     System.out.println(scanner.nextLine());
18   } catch (FileNotFoundException e) {
19    e.printStackTrace();
20   }
21  }
22
23  public static void main(String[] args) {
24   new UtilidadScanner().ejecuta();
25  }
```

[1] Para más información sobre qué son expresiones regulares el lector puede consultar `http://en.wikipedia.org/wiki/Regular_expression`

[2] Estos datos se han generado aleatoriamente tomando como base los datos estadísticos del *Instituto Nacional de Estadística*. Estos datos estadísticos se pueden consultar en la dirección `http://www.ine.es`

```
26
27 }
```

Listado 8.1: Lectura de líneas de texto desde un fichero con el uso de la clase `Scanner`

La clase `Scanner` tiene métodos para poder leer tipos de datos primitivos tal y como se muestra en el Listado 8.2.

```
1 package utilidad;
2
3 import java.io.File;
4 import java.io.FileNotFoundException;
5 import java.util.Scanner;
6
7 public final class UtilidadScanner2 {
8  private static final String FICHERO = "agenda.txt";
9
10  private UtilidadScanner2() {
11   super();
12  }
13
14  private void ejecuta() {
15   try {
16    Scanner scanner = new Scanner(new File(FICHERO));
17    while(scanner.hasNext()) {
18     analizaLinea(scanner.nextLine());
19    }
20   } catch (FileNotFoundException e) {
21    e.printStackTrace();
22   }
23  }
24
25  private void analizaLinea(String linea) {
26   Scanner scanner = new Scanner(linea);
27   scanner.useDelimiter(": ");
28   String persona, nombre, apellidos;
29   int telefono;
30   persona = scanner.next();
31   nombre = scanner.next();
32   apellidos = scanner.next();
33   telefono = scanner.nextInt();
34   System.out.println(nombre + "," + apellidos + "," + telefono);
35  }
36
37  public static void main(String args[]) {
38   new UtilidadScanner2().ejecuta();
39  }
40 }
```

Listado 8.2: Lectura de líneas de texto desde un fichero con el uso de la clase `Scanner`

El resultado de la ejecución de este código es el siguiente:

```
FRANCISCO JOSE,ALVAREZ MARTIN,90727037
ROBERTO,CASTRO RUIZ,945953372
MANUEL,PEREZ HERRERA,520908284
JULIA,ALVAREZ ORTEGA,482596843
TOMAS,VARGAS MARTINEZ,691825532
```

Si construimos la instancia de `Scanner` sobre el flujo de bytes que representa al teclado `System.in`, con la clase `Scanner` podremos leer tipos de datos primitivos, tal y como muestra el Listado 8.3.

```
1 Scanner lectorTeclado = new Scanner(System.in);
2 System.out.print("Introduce un entero: ");
3 int entero = lectorTeclado.nextInt();
4 System.out.println("Introduce un real: ");
5 float real = lectorTeclado.nextFloat();
6 System.out.println("Enero = " + entero + "; real = " + real);
```

Listado 8.3: La clase `Scanner` nos permite leer tipos primitivos desde teclado.

La clase `Scanner` nos permite leer desde cualquier flujo de entrada, no sólo desde el teclado.

8.2. Trabajo con cadenas de caracteres

Ya conoces una clase que representa y manipula cadenas de caracteres, la clase `String`. No obstante, Java proporciona clases más eficientes para trabajar con cadenas, ya que la clase `String` es inmutable y al manipular las cadenas de texto que representa, se crean nuevas instancias de esta clase, con el coste que supone la creación de objetos. A efectos prácticos la inmutabilidad de la clase `String` significa que cuando concatenamos dos cadenas el resultado es una nueva cadena, no se *amplia* ninguna de las cadenas originales para albergar la nueva cadena.

```
1 String cadenaConcatenada = "Hola" + ", como estás";
```

Listado 8.4: "La concatenación de dos cadenas crea una nueva cadena."

En el Listado 8.4 se crean tres objetos de tipo `String`, el primero de ellos contiene la cadena de caracteres *Hola*, el segundo contiene *, como estás* y el tercero contiene `Hola, como estás`.

8.2.1. La clase `String`

La clase `String` representa una cadena de caracteres inmutable, es decir, una vez creada no se puede modificar la secuencia de caracteres. Por la tanto es útil utilizar esta clase cuando no vamos a hacer manipulaciones continuadas sobre la cadena que representa.

El único operador sobrecargado en Java es el operador + cuando se aplica sobre cadenas con el significado de concatenarlas, tal y como muestra el Listado 8.5. Al concatenar dos cadenas se crea una nueva cadena para almacenar el resultado, de ahí la ineficiencia al utilizar el operador + sobre `String`.

```
1 String primera = "Hola ";
2 String segunda = "mundo."
3 String resultado = primera + segunda;
```

Listado 8.5: Uso del operador `+` para concatenar dos cadenas de caracteres.

Para comparar dos cadenas de caracteres, caracter a caracter, no debemos cometer el error de utilizar el operador `==` ya que este operador compara la igualdad de dos referencias. Para comparar dos cadena de caracteres utilizamos el método `public boolean equals(Object o)`, que compara el `String` actual con la representación como `String` del objeto que se le pasa como argumento. El

método `equals(Object o)` distingue entre mayúsculas y minúsculas, si queremos comparar dos cadenas con independencia del uso de mayúsculas/minúsculas utilizaremos el método `public boolean equalsIgnoreCase(String s)`.

Para averiguar el número de caracteres de una cadena utilizamos el método `public int length()`. Si queremos convertir todas las letras de una cadena a minúsculas utilizamos el método `public String toLowerCase()`, y el método `public String toUpperCase` en caso de que la queramos en mayúsculas.

La clase `String` también posee el método sobrecargado `static String valueOf(boolean/char/int/long/float/double)` para convertir tipos de datos primitivos a su representación como cadenas de caracteres.

Un método interesante que nos permite trocear una cadena de caracteres a partir de una subcadena contenida en ellas es `String split(String s)`, donde el argumento es una expresión regular. El Listado 8.6 muestra un ejemplo de uso del método `split`, fíjate que estamos dividiendo la cadena original buscando el patrón representado por otra cadena, `", "`.

```
1 String inicial = "Esta cadena, contiene comas, por la que quiero trocear
      .";
2 String trozos[] = inicial.split(", ");
3 for(String trozo: trozos)
4 System.out.println(trozo);
```

Listado 8.6: Uso del método `split` para trocear una cadena.

El resultado de la ejecución de este código es:

```
Esta cadena
contiene comas
por la que quiero trocear.
```

Para poder dar formato a cadenas al estilo de *C*, la clase `String` nos proporciona el método `public static String format(String cadena, Object... argumentos`. El Listado 8.7 muestra un sencillo caso de uso.

```
1 System.out.println(String.format("El valor de PI es: \%2.2f ", 3.1415));
```

Listado 8.7: Ejemplo de formato usando el método format de la clase `String`

El resultado de la ejecución del Listado 8.7:

```
El valor de PI es: 3,14
```

Si se necesitan formatos más sofisticados, la clase `Formatter` es de gran ayuda.

8.2.2. Las clases `StringBuffer` y `StringBuilder`

La clase `StringBuffer` también representa una cadena de caracteres como la clase `String` pero esta vez la cadena que representa puede cambiar. Esta clase es la recomendada si, por ejemplo, queremos concatenar dos cadenas, ya que el resultado no crea una nueva cadena, si no que se modifica la original para representar la cadena concatenada final. Para ello la clase `StringBuffer` posee el método sobrecargado `StringBuffer`

`append(boolean/int/long/float/double/String/StringBuffer)` que añade la representación como `String` del argumento a la cadena actual.

La clase `StringBuffer` posee otros métodos interesantes de manipulación. Por ejemplo, el método `int indexOf(String s)` devuelve la posición de la primera ocurrencia de la cadena `s` dentro de la cadena original. El método sobrecargado `StringBuffer insert(int offset, boolean/char/int/long/float/double/String)` inserta la representación del segundo argumento en la cadena original a partir del offset indicado en el primer argumento. El Listado 8.8 muestra un ejemplo de uso de estos métodos.

```
1 StringBuffer sb = new StringBuffer("Hola.");
2 sb.insert(sb.indexOf("."), " Java");
3 System.out.println(sb);
```

Listado 8.8: Uso de los métodos `indexOf` y `insert` de la clase `StringBuffer`

Los métodos que manipulan la representación de la cadena dentro de la clase `StringBuffer` están sincronizados, luego se pueden utilizar en aplicaciones en las que varios hilos están accediendo a la misma referencia de la clase `StringBuffer`. Veremos el uso de hilos y lo que significa que un método esté sincronizado en el Capítulo 14.

Por su parte, la clase `StringBuilder` funciona exactamente igual que la clase `StringBuffer`, de hecho los métodos que proporciona la clase `StringBuilder` son exactamente los mismo que la clase `StringBuffer`, pero esta vez ninguno de ellos está sincronizado por razones de eficiencia[3].

8.3. Clases recubridoras

Como ya sabes, en Java existen dos grandes grupos de tipos de datos, los tipos de datos primitivos y los tipos de datos referencia. Sin embargo, Java proporciona clases que recubren los tipos de datos primitivos para poder trabajar con ellos a través de referencias, es decir, como con cualquier otro objeto. Esto es especialmente útil al trabajar con colecciones, tal y como veremos en la Sección 8.4.

Tal y como muestra la Tabla 8.1, para cada tipo primitivo existe una clase recubridora. Crear una clase recubridora a partir de un tipo primitivo es muy sencillo, tal y como muestra el Listado 8.9, donde se crean clases recubridoras tanto a partir de datos primitivos como a partir de la representación como cadena de texto de los datos primitivos.

```
1 Integer entero = new Integer(15);
2 Integer enteroString = new Integer("10");
3 Boolean booleanoVerdadero = new Boolean(true);
4 Boolean booleanoFalso = new Boolean("false");
```

Listado 8.9: Ejemplo de creación de clases recubridoras.

Para recuperar, como tipos primitivos, los valores que almacena una clase recubridora, estas proporcionan métodos tal y como muestra el Listado 8.10

[3]Como veremos en el Capítulo 14, el acceso a métodos sincronizados tiene un sobrecoste temporal debido al uso de cerrojos.

Tipo primitivo	Clase recubridora
void	java.lang.Void
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

Tabla 8.1: Para cada uno de los tipos primitivos de la columna de la izquierda, Java proporciona una clase recubridora, en la columna de la derecha.

```
1 int enteroPrimitivo = entero.intValue();
2 boolean booleanoPrimitivo = booleanoVerdadero.booleanValue();
```

Listado 8.10: Recuperación de los datos como tipos primitivos a partir de las clases recubridoras.

Sin embargo, en la mayoría de los casos es de gran utilidad hacer uso del mecanismo de *Autoboxing* introducido en la versión 5 de Java. Este mecanismo convierte, de modo automático y transparente para el usuario, tipos primitivos a recubridores en una asignación siempre que el tipo primitivo y la clase recubridora sean compatibles, tal y como muestra el Listado 8.11.

```
1 Integer entero = 15;
2 int enteroPrimitivo = entero;
3 Boolean booleanoVerdadero = true;
4 boolean booleanoPrimitivo = booleanoVerdadero;
```

Listado 8.11: Ejemplos de autoboxing para la conversión entre tipos de datos primitivos y sus correspondientes clases recubridoras.

El mecanismo de *Autoboxing* es especialmente útil cuando se utilizan las clases colección, tal y como veremos en la Sección 8.4.

```
1 int entero = Integer.parseInt("10");
2 double real = Double.parseDouble("3.141592");
```

Listado 8.12: Métodos para obtener tipos primitivos a partir de cadenas de caracteres.

Un grupo de métodos especialmente útil cuando se procesan datos de entrada de tipo texto, como por ejemplo los parámetros de llamada a nuestra aplicación, y los queremos convertir a tipos primitivos son los que muestra el Listado 8.12, de modo análogo cada clase recubridora, tiene un método para realizar la correspondiente conversión.

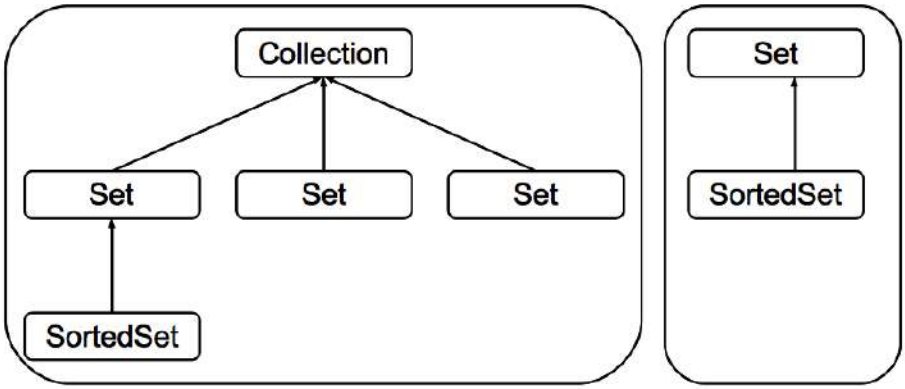

Figura 8.1: Interfaces básicos presentes en el *Java Collections Framework*.

Interface	**Implementación**
Set< E >	HashSet< E >
SortedSet< E >	TreeSet< E >
List< E >	ArrayList< E >, LinkendList< E >, Vector< E >
Queue< E >	LinkendList< E >
Map< E >	HasMap< E >, HashTable< E >
SortedMap< E >	TreeMap< E >

Tabla 8.2: Varias clases colección y los interfaces que implementan.

8.4. Colecciones

Las clases que forman parte del grupo de clases colección son aquellas que nos sirven para almacenar referencias a objetos, e implementan estructuras de datos tales como listas, tablas de dispersión, conjuntos, etcétera. El conjunto de interfaces, clases y algoritmos que nos permiten trabajar con estructuras de datos se agrupan bajo el *Java Collections Framework*.

La Figura 8.4 muestra los interfaces básicos definidos en el *Java Collections Framework*. La parte de la derecha de esta figura muestra los interfaces que representan la funcionalidad de las clase que implementan colecciones de referencias como «conjuntos»; mientras que en la parte derecha de la figura se muestran los interfaces que representan la funcionalidad de las clases que implementen colecciones donde cada elemento de la colección representa una pareja clave/valor.

Las clases que representan colecciones de referencias pueden implementar uno o más de estos interfaces. La Tabla 8.2 muestra algunas clases colección y los interfaces que implementan.

Fíjate que en la Tabla 8.2 hay una novedad, los símbolos $< E >$ a continuación de las interfaces o las clases, esta sintaxis significa que la interface o la clase almacena un tipo *Genérico*. En el momento de creación de la clase debemos especificar cual es el tipo de los elementos que va a contener la clase. Veremos con detalle cómo utilizar genéricos en nuestra propias clases en el Capítulo 9. En

este capítulo sólo veremos como trabajar con ellos en el momento de creación de las clases y las ventajas que supone el uso de genéricos, que fueron introducidos en la versión 5 de Java. Aunque, es posible instanciar las clases colección sin especificar el tipo de los elementos que contendrá. En este caso, el compilador sólo mostrará un aviso con el siguiente texto *ArrayList is a raw type. References to generic type ArrayList< E > should be parameterized* no un error. Sin embargo, es muy aconsejable declarar el tipo de los elementos al crear la colección.

En particular, la clase `ArrayList`$< E >$ representa una secuencia indexada de elementos, cada uno de ellos ocupa una posición dentro de la estructura, y se puede acceder a un elementos dentro de la estructura a través de su índice.

Un ejemplo de cómo usar las clases colección se muestra en los siguientes listados: el Listado 8.13 muestra la interface `Figura` que declara un método para que las clases que la implementen definan la funcionalidad del cálculo del área de una figura. El Listado 8.14 define la clase `Circulo` de cuyas instancias podemos calcular el área. Análogamente, el Listado 8.15 define la clase `Rectangulo` de cuyas instancias también podemos calcular el área. Finalmente, el Listado 8.16 define la clase `TrianguloRectangulo` de cuyas instancias también podemos calcular el área. El Listado 8.17 muestra un ejemplo de cómo utilizar la clase `ArrayList`$< Figura >$ para contener referencias a clases que implementen la interface `Figura`, a ella podemos añadir círculos, cuadrados y triángulos, y los podemos recuperar utilizando un bucle `for...each`.

```
1 package colecciones.figuras;
2
3 public interface Figura {
4  public static final double PI = 3.141592;
5  public double getArea();
6 }
```

Listado 8.13: Interface que declara la funcionalidad del cálculo del área de una figura geométrica.

```
1 package colecciones.figuras;
2
3 public class Circulo implements Figura {
4  private double radio;
5
6  public Circulo() {
7   super();
8  }
9
10  public Circulo(double radio) {
11   this.radio = radio;
12  }
13
14  @Override
15  public double getArea() {
16   return PI*radio*radio;
17  }
18
19  @Override
20  public String toString() {
21   StringBuilder builder = new StringBuilder();
22   builder.append("Circulo [radio=");
23   builder.append(radio);
24   builder.append("]");
25   builder.append(" Area=");
26   builder.append(getArea());
27   return builder.toString();
28  }
```

```
29 }
```

Listado 8.14: Esta clase representa un círculo del que se puede calcular su área.

```
1 package colecciones.figuras;
2
3 public class Rectangulo implements Figura {
4  private double base;
5  private double altura;
6
7  public Rectangulo() {
8   super();
9  }
10
11  public Rectangulo(double base, double altura) {
12   super();
13   this.base = base;
14   this.altura = altura;
15  }
16
17  @Override
18  public double getArea() {
19   return base*altura;
20  }
21
22  @Override
23  public String toString() {
24   StringBuilder builder = new StringBuilder();
25   builder.append("Rectangulo [altura=");
26   builder.append(altura);
27   builder.append(", base=");
28   builder.append(base);
29   builder.append("]");
30   builder.append(" Area=");
31   builder.append(getArea());
32   return builder.toString();
33  }
34 }
```

Listado 8.15: Esta clase representa un rectángulo del que se puede calcular su área.

```
1 package colecciones.figuras;
2
3 public class TrianguloRectangulo implements Figura {
4  private double base;
5  private double altura;
6
7  public TrianguloRectangulo() {
8   super();
9  }
10
11  public TrianguloRectangulo(double base, double altura) {
12   super();
13   this.base = base;
14   this.altura = altura;
15  }
16
17  @Override
18  public double getArea() {
19   return base*altura/2;
20  }
21
22  @Override
23  public String toString() {
24   StringBuilder builder = new StringBuilder();
25   builder.append("TrianguloRectangulo [altura=");
26   builder.append(altura);
27   builder.append(", base=");
```

```
28    builder.append(base);
29    builder.append("]");
30    builder.append(" Area=");
31    builder.append(getArea());
32    return builder.toString();
33  }
34 }
```

Listado 8.16: Esta clase representa un triángulo rectángulo del que se puede calcular su área.

```
1 package colecciones.figuras;
2
3 import java.util.ArrayList;
4
5 public final class Principal {
6  public static void main(String[] args) {
7    ArrayList<Figura> figuras = new ArrayList<Figura>();
8    figuras.add(new Circulo(1));
9    figuras.add(new Rectangulo(1, 2));
10   figuras.add(new TrianguloRectangulo(1, 2));
11   for(Figura figura: figuras)
12    System.out.println(figura);
13  }
14 }
```

Listado 8.17: Ejemplo de uso de la clase ArrayList< *Figura* >.

La ejecución de esta pequeña aplicación muestra el siguiente resultado:

```
Circulo [radio=1.0] Area=3.141592
Rectangulo [altura=2.0, base=1.0] Area=2.0
TrianguloRectangulo [altura=2.0, base=1.0] Area=1.0
```

Si en el `ArrayList`< *Figura* > del Listado 8.17 intentásemos añadir una instancia de una clase que no implementa el interface `Figura`, obtendríamos un error en tiempo de compilación. Si no indicásemos el tipo de los datos que maneja la colección en el momento de la creación del `ArrayList`, el compilador no hubiese detectado el error, y se produciría en tiempo de compilación al extraer el elemento erróneo y modelarlo a la interface común `Figura`, tal y como muestra el Listado 8.18 y el resultado de su ejecución.

```
1 package colecciones.figuras;
2
3 import java.util.ArrayList;
4
5 public class Principal2 {
6  public static void main(String[] args) {
7    ArrayList figuras = new ArrayList();
8    figuras.add(new Circulo(1));
9    figuras.add(new Rectangulo(1, 2));
10   figuras.add(new TrianguloRectangulo(1, 2));
11   figuras.add(new Integer(1));
12   for(Object figura: figuras)
13    System.out.println((Figura)figura);
14  }
15 }
```

Listado 8.18: Ejemplo de uso de la clase ArrayList sin especificar el tipo de los elementos de la colección. Se producirá un error en tiempo de ejecución.

```
Circulo [radio=1.0] Area=3.141592
Rectangulo [altura=2.0, base=1.0] Area=2.0
TrianguloRectangulo [altura=2.0, base=1.0] Area=1.0
Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to colecciones.figuras.Figura at
colecciones.figuras.Principal.main(Principal.java:13)
```

Otros métodos útiles de la clase `ArrayList`$< E >$ (que comparte con el resto de clases que implementan la interface `List`) son: `E get(int posicion)`, devuelve el elementos en la posición indicada; `void clear()` elimina todos los elementos; `boolean contains(Object o)`, de vuelve `true` si el elementos está en la lista y `false` en caso contrario; `boolean isEmpty()`, devuelve `true` si no el `ArrayList`$< E >$ no contiene ningún elemento y `false` en caso contrario; `int size()`, devuelve el número de elementos.

8.5. Trabajo con fechas

El paquete de clases de utilidad de Java nos proporciona un conjunto de clases para trabajar con fechas y especificar el formato cuando la fecha se muestre como texto.

8.5.1. La clase `Date`

La clase `Date` representa un instante del tiempo con una precisión de milisegundos. Para obtener el instante de tiempo actual, simplemente creamos una instancia de esta clase a partir de su constructor por defecto. La información se almacena como un entero `long` cuyo origen de tiempos es el 1 de Enero de 1970 a las 00:00:00 horas.

Muchos de los métodos de esta clase están obsoletos, en particular, todos aquellos métodos relacionados con el trabajo con fechas. Para trabajar con fechas, por ejemplo para saber qué día de la semana será mi próximo cumpleaños, se utiliza la clase `GregorianCalendar` tal y como veremos en la Sección 8.5.2.

Un sencillo ejemplo de uso de la clase `Date` se muestra en el Listado 8.19, lo que mostraría por consola:

```
Ahora: Fri Jul 02 10:19:40 CEST 2010
```

donde se está utilizando un formato anglosajón para mostrar la fecha.

```
1 System.out.println("Ahora: " + new Date());
```

Listado 8.19: Uso de la clase `Date` para mostrar el instante actual.

La clase `SimpleDateFormat` nos permite definir el formato con el que queremos mostrar las fechas. Para una descripción de los símbolos que se pueden utilizar al especificar el formato, se puede consultar la documentación de esta clase en esta dirección `http://java.sun.com/javase/6/docs/api/java/text/SimpleDateFormat.html`, aquí sólo mostraremos el ejemplo del Listado 8.20, donde se está utilizando **EEEE** para mostrar el día de la semana, **dd** para mostrar el ordinal del día dentro del mes, **MMMM** para mostrar el nombre

del mes completo, **yyyy** para mostrar el año, **hh** para mostrar la hora en formato 24 horas, **mm** para mostrar los minutos de la hora, y **ss** para mostrar los segundos dentro del minuto. Con este formato, el texto obtenido es:

```
Ahora: viernes 02 de julio de 2010 (10:30:29)
```

```
1 SimpleDateFormat sdf = new SimpleDateFormat("EEEE dd 'de' MMMM 'de' yyyy
      '('hh':'mm':'ss')'");
2 System.out.println("Ahora: " + sdf.format(new Date()));
```

Listado 8.20: Uso de la clase `SimpleDateFormat` para definir el formato de una fecha como una cadena de texto.

8.5.2. Las clases `Calendar` y `GregorianCalendar`

La clase `Calendar` nos permite convertir instantes temporales, representados por la clase `Date`, a un fecha expresada en días, meses, año, horas, minutos, segundos. La clase `Calendar` es abstracta, por lo que no se puede instanciar. La clase `Calendar` nos proporciona la funcionalidad mínima para trabajar con fechas, y otras extensiones de esta clase implementan calendarios concretos. Este es el caso de la clase `GregorianCalendar` que implementa el cálculo de fechas en el calendario gregoriano. Esta clase nos permite, por ejemplo, saber qué día de la semana será dentro de 40 días, tal y como muestra el Listado 8.21.

```
1  SimpleDateFormat sdf = new SimpleDateFormat("EEEE dd 'de' MMMM 'de'
       yyyy");
2  GregorianCalendar calendario = new GregorianCalendar();
3  System.out.println("Ahora. " + sdf.format(calendario.getTime()));
4  calendario.add(Calendar.DAY_OF_YEAR, 40);
5  System.out.println("Dentro de 40 días: " + sdf.format(calendario.
       getTime()));
```

Listado 8.21: La clase `GregorianCalendar` nos permite trabajar con fechas como, por ejemplo, sumar una cantidad de días a la fecha actual para conocer la nueva fecha

Todas las constantes para indicar el día, mes, etcétera están definidas en la clase `Calendar`.

8.6. Matemáticas

En el paquete estándar de Java encontramos algunas clases para hacer cálculos matemáticos, algunas de ellas son la clase `Math` y la clase `Random`.

8.6.1. La clase `Math`

La clase `Math` nos proporciona algunas de las funciones matemáticas trigonométricas, logarítmicas y otras de utilidad. Todos los métodos de esta clase son `static`, por lo que no hace falta crear una instancia para poder utilizarlos.

El Listado 8.22 muestra algunos ejemplos sencillos de las funciones matemáticas que proporciona la clase `Math`.

```
1  System.out.println(String.format("El seno de %2.2f es %1.3f", Math.PI
       /4, Math.sin(Math.PI/4)));
2  System.out.println(String.format("La raíz cuadrada de 2 es %1.4f",
       Math.sqrt(2)));
3  System.out.println(String.format("El logaritmo natural de %1.3f es
       %1.3f", Math.E, Math.log(Math.E)));
```

Listado 8.22: Algunas funciones de la clase `Math`.

El resultado de la ejecución del Listado 8.22 es el siguiente:

```
El seno de 0,79 es 0,707
La raíz cuadrada de 2 es 1,4142
El logaritmo natural de 2,718 es 1,000
```

8.6.2. La clase `Random`

La clase `Random` genera secuencias aleatorias de números o valores booleanos. La secuencia de números es reproducible si al crear la instancia de `Random` utilizamos la misma semilla. Los valores de las secuencias son equiprobables, excepto si utilizamos el método `double nextGaussian()`, en cuyo caso la secuencia generada sigue una distribución aleatoria de media 0 y desviación estándar 1. El Listado 8.23 muestra algunos ejemplos de uso de los métodos que proporciona esta clase.

```
1  Random equiprobables = new Random();
2  System.out.println("Una secuencias aleatoria equiprobable de números
       entre 0 y 100.");
3  for(int i = 0; i < 10; i++)
4   System.out.print(equiprobables.nextInt(100) + "; ");
5
6  System.out.println("Una secuencias aleatoria gaussiana de números.");
7  for(int i = 0; i < 10; i++)
8   System.out.print(String.format("%.2f; " , equiprobables.nextGaussian
       ()));
```

Listado 8.23: Secuencias de valores aleatorios generados por la clase `Random`.

Cuestiones.

1. Si lees el API de la clase `Calendar` observarás que aunque esta clase es abstracta, posee el método `public static Calendar getInstance()` que devuelve una referencia a `Calendar` perfectamente funcional y con la que podemos trabajar con fechas como si de la clase `GregorianCalendar` se tratase. ¿Cómo es esto posible siendo la clase `Calendar` abstracta?.

Ejercicios.

1. Escribe un programa que calcule la cuota mensual de una hipoteca por el sistema francés, dada por la siguiente fórmula:

$$mensualidad = \frac{capital \cdot n}{1 - \frac{1}{(1+n)^{(12 \cdot años)}}}$$

donde $n = \dfrac{interes}{1200}$.

Lecturas recomendadas.

- El Capítulo 17 de la referencia [2] muestra nuevas clases de utilidad.

Capítulo 9

Programación con genéricos

Contenidos

9.1. ¿Qué son los tipos de datos genéricos? 133
9.2. Métodos genéricos 134
9.3. Clases genéricas 135
9.4. Ampliación del tipo genérico 138
9.4.1. Tipos genéricos con límite superior 139
9.4.2. Comodines 139
9.5. Borrado de tipo y compatibilidad con código heredado 141

Introducción

En este capítulo se presenta la programación con genéricos, que fue introducida en Java en la versión 5. Los genéricos nos permiten definir clases que trabajarán con instancias de objetos de los que no especificamos su tipo en el momento de la definición de la clase. El tipo de las referencias que la clase manejará se especifica en el momento de crear las instancias de la clase genérica.

El concepto de programación con genéricos es muy potente, como has visto en la sección 8.4 dedicada a las clases colección. Las clases colección son contenedores de referencias de las que se especifica su tipo en el momento de crear la instancia de la clase colección. Como vimos, una ventaja del uso de genéricos es asegurar que las referencias con las que trabaja la clase genérica son de tipo compatible con el especificado en el momento de la instanciación de la clase genérica, de lo contrario, obtendremos un error en tiempo de compilación, error que pasaría desapercibido sin el uso de tipos genéricos.

9.1. ¿Qué son los tipos de datos genéricos?

Sin detenernos en detalles, en la sección 8.4 vimos cómo utilizar las clases colección, y vimos que estas clases pueden trabajar con cualquier tipo de datos, basta con indicarlo en el momento de instanciar la clase colección. Vimos la

gran ventaja que esto suponía al no trabajar con referencias a la clase `Object` que debíamos modelar al tipo adecuado al extraer los elementos de la colección. Al trabajar sin genéricos podemos cometer errores (introducir una referencia de tipo incompatible con el resto de elementos en la colección), de los que no nos daremos cuenta en tiempo de compilación, causando graves problemas durante la ejecución de nuestras aplicaciones. El uso de genéricos hace posible la detección de estos errores de incompatibilidad de tipos durante la fase de compilación haciendo que nuestro código sea más robusto.

Un tipo de datos genérico es un tipo de datos que no se especifica, únicamente se indica que se utilizará algún tipo de dato pero no se indica el tipo concreto hasta que no se utiliza. Los tipos de datos genéricos se pueden utilizar en la definición de un método, o en la definición de una clase.

9.2. Métodos genéricos

Un método definido en una clase puede trabajar con un tipo genérico aunque la clase no lo haga. El Listado 9.1 muestra un ejemplo de un método que trabaja con un tipo genérico. Fíjate en la declaración del método `private <T>void muestraNombreClase(T t)`, la `<T>` indica que se va a utilizar un tipo genérico, de modo que la lista de argumentos `(T t)` se interpreta como *una referencia de tipo genérico T*. El modo de uso es una simple llamada al método como por ejemplo `metodoGenerico.muestraNombreClase(new Float(1));`, en este caso, el tipo genérico `<T>` se *sustituye* por el tipo particular `Float`, de modo que el método se reescribe como `private void muestraNombreClase(Float t)`.

```
1 package genericos;
2
3 import tipos.Persona;
4
5 public final class MetodoGenerico {
6  private MetodoGenerico() {
7   super();
8  }
9
10  private <T> void muestraNombreClase(T t) {
11   System.out.println("Soy una instancia de la clase: " + t.getClass().
         getCanonicalName());
12  }
13
14  public static void main(String[] args) {
15   MetodoGenerico metodoGenerico = new MetodoGenerico();
16   metodoGenerico.muestraNombreClase(new Float(1));
17   metodoGenerico.muestraNombreClase(new Persona());
18  }
19 }
```

Listado 9.1: Definición y uso de un método genérico.

Sintaxis de Java

Para indicar que un método trabaja con tipos genéricos se escribe $< T >$ entre el modificador de acceso y el tipo de dato de retorno del método.

Lo que ocurre en el momento de la compilación en el ejemplo anterior, es que se sustituye el símbolo de tipo genérico `<T>` por el tipo concreto (`Float`) en

el ejemplo del Listado 9.1.

Si el método trabaja con un par de tipos genéricos que pueden ser diferentes se indica como **private** <T, U> **void** metodo(T t, U u).

Convención de codificación

Se usa la letra `<T>` para indicar el primer tipo genérico, si es necesario indicar más tipos genéricos se toman las letras mayúsculas que siguen a `T`, por ejemplo `<T, U>`. La convención utiliza las siguientes letras para indicar tipos genéricos:

- E, Indica *Elemento*, como en el caso de las clase Colección.
- K, Indica *Clave*, como en el caso de los Mapas.
- N, Indica *Numero*.
- T, S, U, V, etc. Indica *Tipo*.
- V, Indica *Valor*, como en el caso de los Mapas.

9.3. Clases genéricas

Supongamos que queremos definir una clase que represente una medida tomada por un sensor, de modo que cada medida almacena el valor del dato medido y un comentario descriptivo. Inicialmente no conocemos los tipos de medidas que nos puede devolver un sensor, existen sensores de temperatura que miden temperaturas, y también sensores de viento que miden la intensidad y dirección del viento, por lo que no podemos decidir en el momento de la definición de la clase *Sensor* la naturaleza del dato que representa la medida. ¿Cómo lo podemos resolver? Evidentemente, utilizando genéricos. La idea es dejar sin especificar el tipo de datos que representa la medida tal y como se muestra en el Listado 9.2.

```
1 package sensores;
2
3 public abstract class Sensor<T> {
4  protected T medida;
5  private String descripcion;
6
7  public T getMedicion() {
8   return medida;
9  }
10
11  public final void setDescripcion(String descripcion) {
12   this.descripcion = descripcion;
13  }
14
15  public String getDescripcion() {
16   return descripcion;
17  }
18 }
```

Listado 9.2: La clase `Sensor` no especifica el tipo de dato que proporciona, se indica con `<T>`.

Ahora es el momento de crear algunos sensores concretos. El primer sensor que implementaremos es un sensor de temperaturas. Ya que la temperatura se puede especificar como un número real, elegimos la clase `Float` para representar

la medida de temperatura de modo que nuestra clase `SensorTemperatura` la podemos definir tal y como muestra el Listado 9.3.

```
1 package sensores;
2
3 import java.util.Random;
4
5 public class SensorTemperatura extends Sensor<Float> {
6  private static final float TEMPERATURA_MAXIMA = 45;
7  private Random temperatura = new Random();
8
9  public SensorTemperatura() {
10  super();
11  medida = new Float(TEMPERATURA_MAXIMA*temperatura.nextFloat());
12  setDescripcion();
13 }
14
15 public final void setDescripcion() {
16  super.setDescripcion("Dato de temperatura en grados Celsius");
17 }
18 }
```

Listado 9.3: La clase `SensorTemperatura` define un sensor capaz de registrar temperaturas representadas como `Float`

Para que se vea con mayor claridad el uso de genéricos, vamos a completar el ejemplo definiendo un nuevo sensor, esta vez un sensor de velocidad del viento capaz de tomar datos de la intensidad y dirección del viento. Esta vez ningún tipo de dato primitivo nos sirve para representar la medida de la velocidad del viento, ya que este tiene intensidad y dirección, así que definimos una nueva clase que represente la medida de la velocidad del viento, tal y como muestra el Listado 9.4.

```
1 package sensores;
2
3 public class VelocidadViento {
4  private float intensidad;
5  private Direccion direccion;
6
7  public class Direccion {
8   float vx;
9   float vy;
10
11  public Direccion(float vx, float vy) {
12   this.vx = vx;
13   this.vy = vy;
14  }
15
16  public String toString() {
17   return "[" + vx + ", " + vy + "]";
18  }
19 }
20
21 public VelocidadViento() {
22  super();
23 }
24
25 public VelocidadViento(float intensidad, float vx, float vy) {
26  this.intensidad = intensidad;
27  direccion = new Direccion(vx, vy);
28 }
29
30 public double getIntensidad() {
31  return intensidad;
32 }
33
34 public Direccion getDireccion() {
```

```
35   return direccion;
36  }
37
38  @Override
39  public String toString() {
40   return "Intensidad: " + intensidad + " Dirección: " + direccion;
41  }
42 }
```

Listado 9.4: La clase `VelocidadVieto` define una clase que almacena la intensidad y la dirección de una medida del viento.

Con esta nueva clase, la definición de un sensor de velocidad del viento es la mostrada en el Listado 9.5

```
1 package sensores;
2
3 import java.util.Random;
4
5 import sensores.VelocidadViento.Direccion;
6
7 public class SensorVelocidadViento extends Sensor<VelocidadViento> {
8  private static final float VELOCIDAD_MAXIMA = 100;
9  private Random viento = new Random();
10
11  public SensorVelocidadViento() {
12   super();
13   setMedida();
14   setDescripcion();
15  }
16
17  public final void setDescripcion() {
18   super.setDescripcion("Mide la velocidad y dirección del viento.");
19
20  }
21
22  private final void setMedida() {
23   float angulo = (float)(viento.nextFloat() * Math.PI);
24   float seno = (float)(Math.sin(angulo));
25   float coseno = (float)(Math.cos(angulo));
26   medida = new VelocidadViento(VELOCIDAD_MAXIMA*viento.nextFloat(), seno
         , coseno);
27  }
28 }
```

Listado 9.5: La clase `SensorVelocidadVieto` define un sensor capaz de registrar la velocidad del viento.

En ambos casos de sensores, estamos generando de modo aleatorio el valor del dato leído, y este no se modifica durante la ejecución de la aplicación. Cuando veamos en el Capítulo 14 cómo trabajar con Hilos en Java, reescribiremos el código de esta aplicación para que sea más realista. Finalmente, un ejemplo del uso de estas clase se muestra en el Listado 9.6.

```
1 package principal;
2
3 import sensores.SensorTemperatura;
4 import sensores.SensorVelocidadViento;
5
6 public final class Principal {
7  private Principal() {
8   super();
9  }
10
11  private void ejecuta() {
12   SensorTemperatura sensorT = new SensorTemperatura();
13   System.out.println("Temperatura: " + sensorT.getMedicion());
```

```
14   SensorVelocidadViento sensorV = new SensorVelocidadViento();
15   System.out.println("Viento: " + sensorV.getMedicion());
16  }
17
18  public static void main(String[] args) {
19   Principal principal = new Principal();
20   principal.ejecuta();
21  }
22 }
```

Listado 9.6: Este listado muestra cómo usar los sensores los tipos de sensores anteriormente definidos.

Un ejemplo del resultado de la ejecución del código del Listado 9.6 es:

```
Temperatura: 1.4107025
Viento: Intensidad: 40.831844 Dirección: [0.7000265, -0.7141168]
```

Siguiendo el mismo esquema, basta definir el tipo de dato que mide el sensor, podemos crear cualquier tipo de sensor extendiendo a la clase `Sensor`, y automáticamente será capaz de devolvernos el dato que mide.

9.4. Ampliación del tipo genérico

Para seguir profundizando en las posibilidades de los tipos genéricos, retomemos el ejemplo de la Sección 8.4 de las figuras: círculos, triángulos rectángulos y rectángulos de las que podemos calcular su área. Sabemos que la clase `ArrayList<E>` es un contenedor que trabaja con genéricos, luego podemos crear un `ArrayList<Figura>` para almacenar en él cualquier tipo de figura que se pueda dibujar, tal y como muestra el Listado 9.7.

```
1  // Una lista de figuras.
2  ArrayList<Figura> figuras = new ArrayList<Figura>();
3  // Círculos, triángulos rectángulos y rectángulos son figuras.
4  figuras.add(new Circulo());
5  figuras.add(new TrianguloRectangulo());
6  figuras.add(new Rectangulo());
```

Listado 9.7: Una lista de figuras.

Ahora creamos una lista sólo para almacenar círculos, y una vez creada, ya que la clase `Circulo` implementa el `interface Figura`, intentamos asignar a la lista de figuras `figuras` la lista que sólo almacena círculos `circulos`, tal y como muestra el Listado 9.8.

```
7  // Ahora un ArrayList sólo de círculos
8  ArrayList<Circulo> circulos = new ArrayList<Circulo>();
9  circulos.add(new Circulo(3));
10 circulos.add(new Circulo(5));
11 //  figuras = circulos; // ERROR!!!
```

Listado 9.8: Una lista de círculos.

Para nuestra sorpresa, en la línea de código 11 hay un error, no podemos asignar a un `ArrayList<Figuras>` un `ArrayList<Circulo>`, aunque la clase `Circulo` implemente el `interface Figura`, la conversión de tipos no es posible. Si pudiésemos hacer la asignación, podríamos añadir a la lista de círculos cualquier otra figura a través de la referencia `figuras`.

Si intentamos escribir un método capaz de mostrar el área de todas las figuras de una lista tal y como muestra el Listado 9.9, no tendremos ningún error cuando mostremos las áreas de las figuras de la referencia `figuras`, pero obtendremos el mismo error que en el Listado 9.8 si lo intentamos con la referencia `circulos`. El motivo es el mismo que antes, el tipo `ArrayList<Figura>` es diferente al tipo `ArrayList<Circulo>`.

```
20  private void muestraAreas(ArrayList<Figura> lista) {
21   for(Figura elemento: lista) {
22    System.out.println(elemento.getClass().getCanonicalName() + ": " +
          elemento.getArea());
23   }
24  }
```

Listado 9.9: Un método que recibe una lista de elementos de tipo `interface Figura`.

¿Cómo podemos resolver esta encrucijada? ¿Cómo podemos indicar que queremos una lista de cualquier cosa que implemente la `Figura` para poder asignar a una lista de figuras unas veces listas de sólo círculos y otras veces listas de sólo rectángulos?

Java nos da una solución para este caso, los tipos genéricos con límite superior.

9.4.1. Tipos genéricos con límite superior

El Listado 9.10 muestra un método en el que se da un límite superior al tipo de datos genérico. `<E extends Figura>` indica que los elementos que se pueden almacenar en la colección pueden ser de cualquier subtipo de `Figura`. Fíjate que aunque `Figura` es una `interface` se utiliza la palabra reservada `extends` y no `implements`. De este modo, podemos llamar a este método con cualquier `ArrayList<T>` siempre que sus elementos implementen la `interface Figura`.

```
22  private <E extends Figura> void muestraAreas2(ArrayList<E> lista) {
23   for(Figura elemento: lista) {
24    System.out.println(elemento.getClass().getCanonicalName() + ": " +
          elemento.getArea());
25   }
26  }
```

Listado 9.10: Un método que recibe una lista genérica de elementos que implementan la `interface Figura`.

De este modo podemos restringir los tipos concretos de las clases genéricas. En el Listado 9.10 estamos restringiendo el tipo concreto de la clase genérica a algún tipo que extienda, o implemente, el tipo `Figura`.

9.4.2. Comodines

La línea 13 del Listado 9.11 presenta una nueva construcción del lenguaje: `ArrayList<? extends Figura>figurasGeneral`, donde `?` es un comodín. Esta construcción significa un `ArrayList` de cualquier tipo que implemente (o extienda) a Figura, es decir, a la referencia `figurasGeneral` sí que le podemos asignar cualquier otro `ArrayList` de tipos concretos si esos tipos implementan

(o extienden) el `interface Figura`. De nuevo, fíjate que aunque, como en este caso, las clases finales implementan un interfaz, el `ArrayList` utiliza la palabra reservada `extends`, dicho de otro modo se utiliza siempre `extends` con el significado de *subtipo* sea este por extensión (`extends`) o por implementación (`implements`) de una interface.

```
12  // Una lista con límite superior
13  ArrayList<? extends Figura> figurasGeneral = circulos;
14  ArrayList<Rectangulo> rectangulos = new ArrayList<Rectangulo>();
15  rectangulos.add(new Rectangulo());
16  rectangulos.add(new Rectangulo(1, 2));
17  figurasGeneral = rectangulos;
```

Listado 9.11: Con el uso de comodines podemos definir listas de tipos que extiendan la `interace Figura`

Ahora sí, tanto en la línea 13 como 17 podemos hacer la asignación. Pero debemos pagar un precio por esta nueva posibilidad, y este es que no podemos añadir elementos al `ArrayList` a través de la referencia `figurasGeneral`. No podemos escribir algo como figurasGeneral.add(new Circulo(). Es importante no olvidar esta restricción.

¿Para qué nos sirve entonces esta posibilidad? Aunque no podamos añadir nuevos elementos a esta lista, sí que podemos trabajar con los elementos que hay en ella, de modo que podemos reescribir el método del Listado 9.9 para mostrar el área de todas las figuras contenidas en la lista, con independencia del tipo de elementos con el que se definió la lista tal y como muestra el Listado 9.12.

```
28 private void muestraAreas3(ArrayList<? extends Figura> lista) {
29  for(Figura elemento: lista) {
30   System.out.println(elemento.getClass().getCanonicalName() + ": " +
         elemento.getArea());
31  }
32 }
```

Listado 9.12: Un método que recibe una lista genérica de elementos que implementan la `interface Figura` utilizando comodines.

El método `private void muestraAreas3(ArrayList<? extends Figura>lista)`, es capaz de mostrar el área de los elementos de cualquier lista de figuras, y no hemos utilizado el límite superior en la restricción de tipo `<E extends Figura>` como en el caso del Listado 9.10.

De modo análogo, podemos indicar que una clase genérica trabaja con referencias de cualquier clase padre de una clase dada, por si esto nos pudiera interesar. En este caso la sintaxis de la construcción se muestra en el Listado 9.13.

```
1 ArrayList<? super Circulo> otraLista;
```

Listado 9.13: El tipo de esta clase puede ser cualquier padre de la clase `Circulo`.

9.5. Borrado de tipo y compatibilidad con código heredado

Como has podido ver, la programación con genéricos es muy potente, de hecho, todas las clases del *Java Collection Framework* usan tipos genéricos. Sin embargo, existe un problema de incompatibilidad con código heredado anterior a la versión 5 de Java en las que no existen tipos genéricos. ¿Cómo se resuelve? de uno modo transparente para el programador, el compilador de Java sustituye los tipos genéricos por verdaderos tipos cuando se determinan estos en tiempo de compilación, es decir, si el método del Listado 9.10 en el código se llama una vez con un `ArrayList<Circulo>` y otra vez con un `ArrayList<Recttangulo>`, se crean dos versiones de este método con cada uno de estos dos tipos concretos. A esta técnica se la llama *Borrado de tipo*.

Ejercicios.

1. Crea otros tipos de sensores, por ejemplo un sensor de presión que mida la presión atmosférica, y un sensor de color que mida colores. En este último caso, elige el espacio de colores RGB.

2. Crea una clase genérica `Garage` que permita almacenar en ellas cualquier tipo de vehículos `Coches`, `Furgonetas` y `Motos` por ejemplo, es una ubicación especificada por un número real (como la plaza de garage) o el DNI del usuario especificado como un `String`.

Lecturas recomendadas.

- En esta dirección `http://java.sun.com/docs/books/tutorial/java/generics/index.html` puedes encontrar la referencia básica de Sun sobre el uso de genéricos en Java.

- En esta dirección `http://java.sun.com/docs/books/tutorial/extra/generics/index.html` puedes encontrar otra interesante y detallada referencia, de Gilad Bracha, a la programación con genéricos en Java.

Capítulo 10

Construcción de proyectos con *Ant*

Contenidos

10.1. Qué es *Ant* **144**
10.2. Definición del proyecto **144**
10.2.1. Objetivos 145
10.2.2. Tareas 145
10.3. Compilar el código fuente de un proyecto **146**
10.4. Propiedades **146**
10.5. Estructuras `path-like` **147**
10.6. Ejecución de las Pruebas Unitarias **148**
10.7. Generación de la documentación **150**
10.8. Empaquetado de la aplicación **151**
10.9. Ejecución y limpieza **151**

Introducción

Hasta ahora, hemos utilizado un entorno de desarrollo integrado, como Eclipse, para realizar todas las tareas relativas a nuestro proyecto. Sabemos cómo compilar nuestro código, cómo generar la documentación con `javadoc`, cómo ejecutar nuestro código, cómo empaquetar nuestro código y cómo realizar pruebas sobre nuestro código con JUnit.

Todas estas tareas han necesitado nuestra intervención, para ejecutar el código de nuestra aplicación hemos tenido que pulsar el botón correspondiente en Eclipse, y del mismo modo, hemos tenido que actuar sobre el IDE para ejecutar las pruebas unitarias.

Eclipse ha sido el IDE que hemos elegido, pero como ya hemos comentado, existen otros excelentes entornos de desarrollo, como por ejemplo NetBeans[1]. La elección entre unos y otros acaba siendo cuestión de gustos y adaptación más

[1] Neatbeans se puede descargar para su instalación desde `http://netbeans.org/`

que de la funcionalidad que los entornos proporcionan, ya que todos ellos proporcionan una funcionalidad parecida, al menos, suficiente para lo que nosotros necesitamos en este momento.

En cada uno de estos entorno de desarrollo, el modo de ejecutar una aplicación, o el modo de lanzar la herramienta de documentación cambia. Incluso el modo en que el entorno de desarrollo organiza las carpetas del proyecto puede cambiar, un entorno puede usar el nombre `src` y otro `source`. Estos pequeños cambios hacen necesaria la intervención del desarrollador para migrar proyectos de un entorno a otro.

Sería interesante que, con independencia del entorno de desarrollo, o incluso si no utilizásemos ningún entorno de desarrollo, fuese posible realizar las tareas comunes sobre un proyecto Java de modo estándar.

La herramienta de construcción de proyectos *Ant* nos proporciona precisamente eso, un modo de trabajar con nuestros proyectos con independencia del entorno de desarrollo elegido, o incluso poder trabajar directamente sobre nuestro proyecto desde consola.

10.1. Qué es *Ant*

Ant es una herramienta de construcción de proyectos. Pero su utilidad no se detiene ahí, con *Ant* podemos hacer mucho más que simplemente compilar el código de nuestro proyecto, podemos realizar, de modo automático, otras muchas tareas como la ejecución de prueba sobre nuestro código, la generación de informes a partir de los resultados de las pruebas, la generación de la documentación de nuestro proyecto y un largo, largísimo etcétera. *Ant* es extensible, de modo que incluso podemos definir nuestras propias tareas, ya que *Ant* está escrito en Java.

Ant es un proyecto de la *Fundación Apache* que puedes encontrar en esta dirección `http://ant.apache.org`, donde también se encuentran los sencillos detalles de instalación, basta con descomprimir el fichero que se puede descargar desde la página web de la *Fundación Apache* y establecer las correspondientes variables de entorno.

10.2. Definición del proyecto

La definición de un proyecto *Ant* siempre se hace en un fichero llamado `build.xml`. Como ejemplo de uso de *Ant*, vamos a utilizar el proyecto de conversión de temperaturas presentado en el Capítulo 5 y las pruebas unitarias sobre este proyecto presentadas en el Capítulo 6.

Dentro del proyecto de conversión de temperaturas crea un fichero y llámalo `build.xml`, el fichero de descripción del proyecto es un fichero *xml*. Si este estándar tecnológico no te es familiar, interesa que antes de seguir adelante conozca esta tecnología. En esta dirección `http://www.w3.org/standards/xml/core` encontrarás una introducción a esta tecnología proporcionada por el *World Wide Web Consortium*, el consorcio que se encarga de los estándares Web[2]

[2]Este organismo es el encargado del proceso de estandarización de las tecnologías Web, su página web está repleta de información con respecto a estos estándares tecnológicos. Al igual que la página web de *Sun* sobre Java es la referencia básica en la web sobre el lenguaje de programación Java, la página web del *W3C* es la referencia básica para los estándares web.

La etiqueta raíz bajo la cual se define todo el proyecto es `<project>` que tiene un atributo obligatorio `name` con el que especificamos el nombre del proyecto. El Listado 10.1 muestra la definición básica de un proyecto.

```
1 <project name="ConversionTemperaturas">
2  ...
3 </project>
```

Listado 10.1: Definición de un proyecto *Ant*.

Si la sintaxis *xml* no te es familiar, fíjate que la etiqueta de apertura `<project>` está cerrada con la etiqueta `</project>`[3].

Un proyecto *Ant* está formado, por uno o más objetivos, y cada uno de estos objetivos puede estar formado por una o más tareas. Cada tarea se realizarán en el orden en el que fue especificada dentro del objetivo. Veamos cómo se define un objetivo en *Ant*.

10.2.1. Objetivos

Para definir un objetivo utilizamos la etiqueta `<target>` que de nuevo tiene un atributo obligatorio, el nombre del objetivo. De nuevo, cada objetivo debe ir cerrado con su correspondiente etiqueta de cierre tal y como muestra el Listado 10.2.

```
1 <project name="ConversionTemperaturas">
2  <target name="empezando">
3   ...
4  </target>
5 </project>
```

Listado 10.2: Definición de un objetivo *Ant*.

Cada uno de los objetivos contiene una o más tareas que se ejecutarán en el orden en que fueron definidas cuando ejecutemos el correspondiente objetivo. Veamos cómo se especifican las tareas pertenecientes a un objetivo.

10.2.2. Tareas

En *Ant* existe una enorme cantidad de tareas predefinidas, por ejemplo, existe una tarea para compilar el código fuente de nuestra aplicación, y otra tarea para crear un fichero *jar* a partir del código compilado de nuestra aplicación. A lo largo de este capítulo iremos describiendo las tareas más utilizadas. En este momento, y como ejemplo, vamos a utilizar una tarea muy sencilla cuya función es simplemente mostrar un mensaje de texto en consola, tal y como muestra el Listado 10.3.

```
1 <project name="ConversionTemperaturas">
2  <target name="empezando">
3   <echo>Empezando con Ant.</echo>
4  </target>
5 </project>
```

Listado 10.3: La tarea `<echo>` muestra un mensaje de texto por consola.

[3]En un fichero *xml* bien formado toda etiqueta abierta debe estar cerrada con su correspondiente etiqueta.

Ejecutar un objetivo desde línea de instrucciones es muy sencillo, basta situarse en el directorio donde se encuentre el fichero `build.xml` y teclear `ant empezando`, lo que invocará la ejecución del objetivo `empezando` definido en el fichero `build.xml`. El resultado de la ejecución será parecido a:

```
Rayuela:ant oscar$ ant empezando
Buildfile: build.xml

empezando:
     [echo] Empezando con Ant.

BUILD SUCCESSFUL
Total time: 0 seconds
```

Pasemos a ver cómo utilizar algunas otras tareas más útiles que `<echo>`.

10.3. Compilar el código fuente de un proyecto

La etiqueta `<javac>` nos permite compilar código Java. Para poder usar esta etiqueta debemos indicar el directorio donde está el código fuente mediante el atributo `srcdir`, e indicar donde se escribirán los ficheros de clases compilados mediante el atributo `destdir`. El Listado 10.4 muestra el uso de la etiqueta `<javac>`.

```
1 <project name="ConversionTemperaturas">
2  <target name="compile" description="Compila el proyecto">
3   <mkdir dir="../excepciones/build/classes"/>
4   <javac srcdir="../excepciones"
5    destdir="../excepciones/build/classes" />
6  </target>
7 </project>
```

Listado 10.4: Compilar un proyecto usando *Ant*.

Fíjate que previamente a la tarea de compilación de la línea 3, hemos utilizado la tarea `mkdir` para crear el directorio de destino de las clases compiladas. Ahora ya puedes compilar el código de tu proyecto invocando al objetivo `compile` bien desde consola, bien desde la vista *Ant* de *Eclipse*.

10.4. Propiedades

El objetivo `compile` tal y como lo hemos descrito tiene un inconveniente, y es que si decidimos cambiar el fichero de destino de los ficheros compilados por ejemplo desde el original `../excpeciones/build/classes`, a otro directorio como por ejemplo `../excepciones/build/project/classes`, tendremos que cambiar todas la ocurrencias del destino original. Para solucionar esta situación, podemos utilizar las propiedades que nos permiten asociar a un nombre un valor, y hacer referencia al valor a través de su nombre en cualquier lugar del fichero `build.xml`, tal y como muestra el Listado 10.5.

```
1 <project name="ConversionTemperaturas">
2  <!-- Directorio del código fuente -->
```

```
 3 <property name="src.dir" location="../excepciones"/>
 4 <!-- Directorio de clases compiladas -->
 5 <property name="build.dir" location="build"/>
 6 <!-- Subdirectorio de las clases compiladas del proyecto -->
 7 <property name="build.classes.dir" location="${build.dir}/classes"/>
 8
 9 <target name="compile" description="Compila el proyecto">
10  <mkdir dir="${build.classes.dir}"/>
11  <javac srcdir="${src.dir}"
12   destdir="${build.classes.dir}" />
13 </target>
14 </project>
```

Listado 10.5: Uso de las propiedades.

En este caso, cada una de las propiedades hace referencia a una dirección representada por el atributo `location`.

10.5. Estructuras `path-like`

Con las propiedades podemos definir valores a los que poder hacer referencia por su nombre. Las estructuras `path-like` son aún más potentes, y nos permiten definir grupos de directorios o ficheros. Por ejemplo, en nuestro proyecto de la aplicación de conversión de temperaturas, tenemos programadas una serie de clases de pruebas unitarias que necesitamos compilar antes de ejecutar. Para compilar las clases de pruebas unitarias necesitaremos la biblioteca `junit.jar`, además del directorio donde se encuentran las clases de prueba que queremos compilar.

Para definir grupos de directorios y ficheros *Ant* nos proporciona la etiqueta `<path>`. El Listado 10.6 muestra el uso de esta etiqueta con el objetivo de compilar las clases de pruebas unitarias.

```
 1 <project name="ConversionTemperaturas">
 2  <!-- Directorio del código fuente -->
 3  <property name="src.dir" location="../excepciones"/>
 4  <!-- Directorio de clases compiladas -->
 5  <property name="build.dir" location="build"/>
 6  <!-- Subdirectorio de las clases compiladas del proyecto -->
 7  <property name="build.classes.dir" location="\${build.dir}/classes"/>
 8  <!-- Directorio de las clases de prueba -->
 9  <property name="test.dir" location="../test/test"/>
10  <!-- Subdirectorio de las clases compiladas de prueba -->
11  <property name="test.classes.dir" location="\${build.dir}/test-classes"
        />
12  <!-- Directorio de bibliotecas del proyecto -->
13  <property name="lib" location="../lib"/>
14
15  <path id="test.compile.classpath">
16   <fileset dir="${lib}" includes="*.jar"/>
17   <pathelement location="\${build.classes.dir}"/>
18  </path>
19
20  <target name="compile" description="Compila el proyecto">
21   <mkdir dir="${build.classes.dir}"/>
22   <javac srcdir="\${src.dir}"
23    destdir="${build.classes.dir}" />
24  </target>
25
26  <target name="compile-tests"
27   depends="compile"
28   description="Compila los tests.">
29   <mkdir dir="${test.classes.dir}"/>
30   <javac srcdir="${test.dir}"
31    destdir="${test.classes.dir}">
```

```
32    <classpath refid="test.compile.classpath"/>
33   </javac>
34  </target>
35 </project>
```

Listado 10.6: Uso de estructuras *path-like* para definir la ruta a las bibliotecas del proyecto.

En las líneas 9, 11 y 13 del Listado 10.6 estamos definiendo las propiedades que hacen referencia al directorio con el código fuente de las clases de prueba, al directorio destino de las clases compiladas de prueba y al directorio donde están todas las bibliotecas del proyecto respectivamente.

Por su lado, entre las líneas 15-18, mediante una estructura *path-like*, estamos definiendo donde están las bibliotecas necesarias para compilar las clases de prueba (junit.jar y harmcret.jar) y donde están las clases compiladas del proyecto.

Finalmente, entre las líneas 26-34 estamos definiendo un objetivo para compilar las clases de prueba.

Fíjate en la línea 27, en esa línea estamos indicando que el objetivo `compile-test` depende de la tarea `compile`. Evidentemente, para poder compilar las clases de prueba, las clases a probar deben estar previamente compiladas, mediante el atributo `depends` de la etiqueta `target` se fuerza a cubrir los objetivos especificados en antes de cubrir el objetivo actual.

10.6. Ejecución de las Pruebas Unitarias

El Listado 10.7 muestra cómo ejecutar una batería de pruebas y grabar el resultado a un fichero como un informe con formato de texto.

```
1 <project name="ConversionTemperaturas" default="test">
2  <!-- Directorio del código fuente -->
3  <property name="src.dir" location="../excepciones"/>
4  <!-- Directorio de clases compiladas -->
5  <property name="build.dir" location="build"/>
6  <!-- Subdirectorio de las clases compiladas del proyecto -->
7  <property name="build.classes.dir" location="\${build.dir}/classes"/>
8  <!-- Directorio de las clases de prueba -->
9  <property name="test.dir" location="../test/test"/>
10 <!-- Subdirectorio de las clases compiladas de prueba -->
11 <property name="test.classes.dir" location="\${build.dir}/test-classes"
       />
12 <!-- Directorio de bibliotecas del proyecto -->
13 <property name="lib" location="../lib"/>
14 <!-- Directorio de informes -->
15 <property name="reports.dir" location="reports"/>
16 <!-- Directorio para los informes en formato texto -->
17 <property name="reports.txt.dir" location="${reports.dir}/txt"/>
18
19 <!-- Path para compilar las clases de prueba -->
20 <path id="test.compile.classpath">
21  <fileset dir="${lib}" includes="*.jar"/>
22  <pathelement location="${build.classes.dir}"/>
23 </path>
24
25 <!-- Path para ejecutar las clases de prueba -->
26 <path id="test.classpath">
27  <path refid="test.compile.classpath"/>
28  <pathelement path="${test.classes.dir}"/>
29 </path>
30
31 <target name="compile" description="Compila el proyecto">
```

```
32   <mkdir dir="${build.classes.dir}"/>
33   <javac srcdir="\${src.dir}"
34     destdir="${build.classes.dir}" />
35  </target>
36
37  <target name="compile-tests"
38   depends="compile"
39   description="Compila los tests.">
40   <mkdir dir="${test.classes.dir}"/>
41   <javac srcdir="${test.dir}"
42     destdir="${test.classes.dir}">
43    <classpath refid="test.compile.classpath"/>
44   </javac>
45  </target>
46
47  <target name="test"
48   depends="compile-tests"
49   description="Ejecuta los tests unitarios">
50   <mkdir dir="${reports.dir}"/>
51   <mkdir dir="${reports.txt.dir}"/>
52   <junit printsummary="true"
53     haltonfailure="false"
54     failureproperty="test.failures">
55    <classpath refid="test.classpath"/>
56    <formatter type="plain" />
57    <test name="test.AllTests"
58      todir="${reports.txt.dir}"/>
59   </junit>
60  </target>
61 </project>
```

Listado 10.7: Ejecutar la batería de pruebas unitarias.

En el Listado 10.7 se ha añadido el atributo `default="test"` al proyecto para indicar que el objetivo `test` es el objetivo por defecto, si no se selecciona ningún otro, este es el objetivo que se invoca al ejecutar *Ant*. También se han añadido las propiedades y estructuras *path-like* necesarias para la ejecución de las pruebas. Si únicamente queremos ejecutar algunas de las pruebas y no toda la suite utilizaríamos la variante mostrada en el Listado 10.8, donde se muestran sólo las líneas añadidas al fichero `build.xml`.

```
1 <!-- Directorio para los informes en formato xml -->
2 <property name="reports.xml.dir" location="${reports.dir}/xml"/>
3
4  ...
5
6 <target name="test-xml"
7  depends="compile.tests"
8  description="Ejecuta los tests unitarios">
9  <mkdir dir="${reports.dir}"/>
10 <mkdir dir="${reports.xml.dir}"/>
11 <junit printsummary="true"
12   haltonfailure="false"
13   failureproperty="test.failures">
14   <classpath refid="test.classpath"/>
15   <formatter type="xml" />
16   <batchtest todir="${reports.xml.dir}">
17    <fileset dir="${test.classes.dir}">
18     <include name="**/Test*.class"/>
19    </fileset>
20   </batchtest>
21 </junit>
22 </target>
```

Listado 10.8: Ejecutar sólo algunas de las pruebas.

Fíjate, que en este último caso, además estamos indicando que el formato de los informes sea `xml`, esta posibilidad es interesante ya que a partir de estos

informes podremos hacer sobre ellos una transformación para generarlos en formato `html` tal y como muestra el Listado 10.9, donde, de nuevo, sólo aparecen las líneas añadidas al fichero `build.xml`.

```
<!-- Directorio para los informes en formato html -->
<property name="reports.html.dir" location="${reports.dir}/html"/>

...

<target name="test.reports"
 depends="test"
 description="Genera los informes de los tests en formato xml">
 <junitreport todir="${reports.xml.dir}">
  <fileset dir="${reports.xml.dir}">
   <include name="TEST-*.xml"/>
  </fileset>
  <report format="frames"
   todir="${reports.html.dir}"/>
 </junitreport>
 <fail if="test.failures"
  message="Se han producido errores en los tests."/>
</target>
```

Listado 10.9: Generar informes de las pruebas en formato `xml`.

10.7. Generación de la documentación

Otra tarea que se puede automatizar es la generación de la documentación de nuestro proyecto tal y como muestra el Listado 10.10. La tarea `<javadoc>` tiene algunos atributos interesantes, como por ejemplo `access` que nos permite indicar los métodos y atributos de los que se generará la documentación según su modificador de acceso. En nuestro ejemplo, se generará documentación de todos los métodos y atributos cuya visibilidad sea `private` o mayor. Otros atributos útiles son `author`, si su valor es `true` se añadirá información del autor a la documentación, y `version`, si su valor es `true` se añadirá información de la versión.

```
<!-- Directorio para la documentación -->
<property name="reports.javadoc" location="${reports.dir}/javadoc"/>

...

<target name="javadoc"
 depends="compile"
 description="Genera la documentacio?n del proyecto.">
 <javadoc sourcepath="${src.dir}"
  destdir="${reports.javadoc}"
  author="true" version="true"
  use="true" access="private"
  linksource="true" encoding="ISO-8859-1"
  windowtitle="${ant.project.name}">
  <classpath>
   <pathelement path="${test.classes.dir}"/>
   <pathelement path="${build.classes.dir}"/>
  </classpath>
 </javadoc>
</target>
```

Listado 10.10: Generación de la documentación del proyecto.

10.8. Empaquetado de la aplicación

Ant También nos proporciona la tarea `<jar>` para empaquetar nuestra aplicación, tal y como muestra el Listado 10.11. En este caso, hemos empleado la etiqueta `<property>` con el atributo `<value>` para definir el nombre del fichero empaquetado. La sintaxis de la tarea es bastante autoexplicativa, hay que indicar el nombre del fichero empaquetado con el atributo `destfile` de la etiqueta `jar`. E indicar los ficheros que se van a incluir dentro del fichero empaquetado mediante una estructura *path-like*. Podemos también indicar el contenido del fichero de manifiesto con la etiqueta `<manifest>`.

```
<!-- Directorio para el fichero empaquetado -->
<property name="dist.dir" location="dist" />
<!-- Nombre del fichero empaquetado -->
<property name="dist.name" value="ConversorTemperaturas.jar" />

...

<target name="package"
 depends="compile"
 description="Genera el fichero jar" >
 <mkdir dir="${dist.dir}"/>
 <jar destfile="${dist.dir}/\${dist.name}">
  <fileset dir="${build.classes.dir}"/>
  <manifest>
   <attribute
    name="Main-Class"
    value="conversor.Principal"/>
  </manifest>
 </jar>
</target>
```

Listado 10.11: Empaquetado del proyecto.

10.9. Ejecución y limpieza

También podemos ejecutar nuestra aplicación, como un fichero empaquetado, usando la tarea `<java>` de *Ant*, tal y como se muestra en el Listado 10.12.

```
<target name="execute"
 depends="package"
 description="Ejecuta la aplicacio?n.">
 <java
  jar="${dist.dir}/${dist.name}"
  fork="true"/>
</target>
```

Listado 10.12: Ejecución del proyecto como un fichero empaquetado.

Cabe destacar que para ejecutar la aplicación se debe crear una nueva instancia de la máquina virtual, cosa que indicamos con el valor `true` del atributo `fork` de la tarea `java`.

También podemos borrar los directorios con las clases compiladas y los informes por si nos es de utilidad. El Listado 10.13 muestra el objetivo `clean` formado por un conjunto de tareas `<delete>` que borran los directorios deseados.

```
<target name="clean" description="Limpia el proyecto">
 <delete dir="\${dist.dir}"/>
```

```
3  <delete dir="\${build.dir}"/>
4  <delete dir="\${reports.dir}"/>
5 </target>
```

Listado 10.13: Borrado de directorios.

En el Apéndice A se muestra el listado completo del fichero `build.xml`.

Lecturas recomendadas.

- Sin duda la referencia básica sobre *Ant* se encuentra el la propia página web del proyecto, esta es la dirección directa `http://ant.apache.org/manual/index.html`.
- El capítulo 1 de la referencia [13] presenta la herramienta *Ant* de modo conciso pero muy informativo.
- En español, es interesante el capítulo 3 de la referencia [5].

Capítulo 11

Interfaces gráficas de usuario

Contenidos

11.1. APIs para la programación de interfaces gráficos de usuario en Java: AWT y Swing 154
11.2. Contenedores y Componentes 155
11.3. Gestores de Aspecto (*Layout Managers*) 155
11.4. Detección de eventos: Escuchadores 157
11.5. Algunos componentes Swing 162
11.5.1. `JLabel`, muestra texto o iconos 162
11.5.2. `JButton`, botones que el usuario puede pulsar 162
11.5.3. `JTextField`, campos de introducción de texto 163
11.5.4. `JRadioButton`, botones de opciones 164
11.5.5. `JCheckBox`, botones de selección múltiple 166
11.5.6. `JList`, listas de selección 166
11.6. El patrón de diseño Modelo/Vista/Controlador . 168

Introducción

Hasta ahora, la interacción con nuestras aplicaciones ha sido a través de consola, los datos de entrada los tecleamos en consola, y la respuesta la obteníamos también directamente en consola.

Cada vez es menor el número de aplicaciones con interfaces de usuario basados en consola. Uno de los principales inconvenientes de este tipo de interfaces de usuario es que son poco intuitivos y por lo tanto susceptibles de crear confusión en el usuario y como resultado que los datos de entrada al programa sean erróneos.

Por contrapartida, las aplicaciones basadas en interfaces gráficos de usuario son más intuitivas y la entrada de datos puede estar acotada, evitando que el usuario introduzca valores erróneos.

Java proporciona dos grandes APIs para programar interfaces gráficas de

usuario: Abstract Windows Toolkit (AWT) y Swing. En este capítulo veremos cuales son las características de ambas y justificaremos por qué elegir una sobre otra.

La programación de interfaces gráficos de usuario está basada en la idea de que los componentes gráficos, tales como botones y cajas de edición de texto, son capaces de lazar eventos cuando el usuario interacciona sobre ellos. Por ejemplo, cada vez que el usuario hace click sobre un botón, este lanza un evento como respuesta, y será tarea del programador escribir el código necesario para *escuchar* el tipo de eventos que le interese y actuar en consecuencia cuando se produzca uno.

Los interfaces gráficos de usuario son anteriores a la aparición del lenguaje de programación Java, y en su programación se han descubierto interesantes patrones de diseño, uno de ellos, ampliamente utilizado en otros lenguajes de programación como por ejemplo *Smalltalk*, es el patrón de diseño Modelo/Vista/Controlador. Este patrón de diseño agrupa las clases de una aplicación según su responsabilidad, de modo que una clase sólo puede formar parte bien del Modelo, bien de la Vista o bien del Controlador. Veremos con detalle este patrón de diseño y alguna técnica útil para su implementación.

11.1. APIs para la programación de interfaces gráficos de usuario en Java: AWT y Swing

En Java, existen dos APIs para la programación de interfaces gráficos de usuario AWT (*Abstract Window Toolkit*) y Swing. AWT fue la primera API disponible en Java y sus principales características son:

- La creación de componentes gráficos se delega al Sistema Operativo.
- El Sistema Operativo se encarga de dibujar los componentes gráficos y de la detección de eventos.
- El aspecto de la aplicación es el nativo del Sistema Operativo.

La principal desventaja de AWT es que descansa directamente sobre el Sistema Operativo quien interviene tanto en la creación de componentes gráficos como en la detección de eventos, de modo que la aplicación se puede ralentizar si la interfaz contiene muchos elementos gráficos, y por otro lado no se pueden introducir cambios en el aspecto de los componentes.

El API Swing viene a liberar la creación de interfaces gráficos de la carga que supone la dependencia con respecto al Sistema Operativo. Las principales características de este API son:

- Swing se encarga de dibujar los componentes y de detectar la interacción sobre ellos.
- El conjunto de componentes es más grande que el que proporciona el Sistema Operativo.
- Se tiene control absoluto sobre el aspecto de los componentes.

Por todo ellos, Swing ha ido desplazando a AWT en la creación de interfaces gráficos de usuario en Java.



Figura 11.1: Colocación de *Componentes* con `FlowLayout`.

11.2. Contenedores y Componentes

Dentro de Swing tenemos dos grandes grupos de elementos: los *Contenedores* y los *Componentes*. La diferencia entre ellos es que los *Contenedores* pueden albergar otros *Contenedores* o *Componentes* dentro de ellos, y los *Componentes* son los elementos gráficos con los que el usuario puede interaccionar, como por ejemplo botones, listas, etcétera.

Los tres *Contenedores* que disponibles en Swing son `JFrame` que representa una ventana con marco, `JWindow` que representa una ventana sin marco, y `JPanel` que no tiene un aspecto visual, su cometido es albergar otros *Contenedores* o *Componentes* dentro de él.

La idea básica es que vamos a utilizar los `JPanel` como muñecas rusas, de modo que colocaremos *Componentes* dentro de `JPanel` y esos `JPanel` dentro de otros `JPanel` con más *Componentes* para ir creando el aspecto deseado para el interfaz gráfico de nuestra aplicación.

11.3. Gestores de Aspecto (*Layout Managers*)

Cuando se programan interfaces gráficos de usuario, un aspecto importante es la colocación de los *Componentes* dentro de la ventana de nuestra aplicación. Java nos facilita esta tarea mediante el uso de Gestores de Aspecto (*Layout Managers* en inglés). Estos Gestores de Aspecto son los encargados de colocar los *Componentes* que vamos añadiendo en los *Contenedores*. Cada uno de los Gestores de Aspecto sigue una política de colocación de los componentes, así, por ejemplo, `BoxLayout` coloca cada nuevo componente al la izquierda del último componente añadido, como en el sentido de la escritura del español, de tal modo que si no hay espacio suficiente para insertar un nuevo *Componente* en la línea actual, porque se ha llegado al borde de la ventan, el *Componente* se añadirá al principio de una nueva línea por debajo de la actual. La Figura 11.1 muestra un ejemplo de este comportamiento.

Este Gestor de Aspecto es el que por defecto utiliza `JPanel` cada



Figura 11.2: Colocación de *Componentes* con `BorderLayout`.

vez que añadimos sobre él un *Componente*. `JFrame` posee otro Gestor de Aspecto por defecto `BorderLayout`. Este Gestor de Aspecto define 5 zonas `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, dentro de las cuales sólo podemos añadir un *Componente* o *Contenedor* con el método `add(Component componente, int zona)`, si no indicamos la zona (`add(Component componente`), el nuevo componente será añadido a la región central. La Figura 11.2 muestra una ventana en la que se observa la disposición de las cinco zonas de `BorderLayout`. El código fuente de este sencilla ejemplo aparece en el Listado 11.1.

```
1 package gui;
2
3 import java.awt.BorderLayout;
4 import java.awt.Container;
5
6 import javax.swing.JButton;
7 import javax.swing.JFrame;
8 import javax.swing.SwingUtilities;
9
10 public final class EjemploBorderLayout {
11  private EjemploBorderLayout() {
12   super();
13  }
14
15  private void creaGUI() {
16   JFrame ventana = new JFrame("BorderLayout Manager");
17   Container contenedor = ventana.getContentPane();
18   contenedor.add(new JButton("Centro"));
19   contenedor.add(new JButton("Norte"), BorderLayout.NORTH);
20   contenedor.add(new JButton("Sur"), BorderLayout.SOUTH);
21   contenedor.add(new JButton("Este"), BorderLayout.EAST);
22   contenedor.add(new JButton("Oeste"), BorderLayout.WEST);
23   ventana.setSize(400, 400);
24   ventana.setVisible(true);
25  }
26
27  public static void main(String[] args) {
28   SwingUtilities.invokeLater(new Runnable() {
29    @Override
30    public void run() {
```

```
31      new EjemploBorderLayout().creaGUI();
32    }
33   });
34  }
35 }
```

Listado 11.1: Ejemplo de uso de `BorderLayout`

El Listado 11.1 contiene algunos detalles importante. En la línea 16, estamos almacenando una referencia al contenedor de la ventana principal, que es donde añadimos los botones. Las líneas 17-21 añaden un botón, instancia de `JButton` a cada uno de las cinco regiones que define un `BorderLayout`. En la línea 22 estamos indicando el tamaño de la ventana, y finalmente, en la línea 23 hacemos visible la ventana. En las línea 28-33 estamos utilizando un técnica que quedará definitivamente clara cuando veamos cómo programar hilos en Java en el Capítulo 14, en este momento basta decir que en esas líneas de código se está creando un hilo para atender al interfaz gráfico de usuario de modo que no interfiere con el hilo de la aplicación principal.

Los Gestores de Aspecto por defecto se pueden cambiar con el método `setLayout(java.awt.LayoutManager)` al que se le pasa una instancia del nuevo Gestor de Aspecto que queremos utilizar.

Otros Gestores de Aspecto son `GridLayout`, que permite definir una rejilla con filas y columnas, y podemos añadir un componente dentro de cada una de las posiciones dentro de la rejilla; y `BoxLayout` nos permite disponer los componentes verticalmente, uno encima de otro, u horizontalmente, uno a la izquierda de otro.

11.4. Detección de eventos: Escuchadores

Si has probado a ejecutar el código del Listado 11.1 quizás te hayas dado cuenta del siguiente detalle, al pulsar sobre el botón de cerrar la ventana que aparece en el marco de esta, la ventana se cierra pero la aplicación sigue ejecutándose. No existe ningún error en la aplicación, simplemente, al cerrar la ventana, lo que en realidad está ocurriendo es que se hace invisible, pero con ello no su fuerza que acabe la ejecución de la aplicación.

Para entender lo que está realmente ocurriendo tenemos que conocer de qué modo actúan los *Contenedores* y *Componentes* en Java[1].

La idea básica es que cuando el usuario interacciona con los *Contenedores* o los *Componentes* estos lanzan eventos como respuesta, el programador debe escuchar estos eventos, y cuando los recibe actuar en consecuencia. Dicho de otro modo, el programador escribe código que observa cuando un evento ocurre, y le indica al componente que quiere ser informado cuando el evento ocurra. Por su lado, el componente informa al observador de que el evento ha ocurrido cada vez que este se produce. De nuevo nos encontramos con un patrón de diseño llamado *Observador*. La sección 16.9 presenta este patrón de diseño con detalle. La Figura 11.3 muestra gráficamente la dinámica en la detección de eventos.

De modo resumido, los pasos que debemos dar para responder cuando un evento ocurra son:

[1] Este comportamiento es igualmente seguido en otros muchos lenguajes de programación

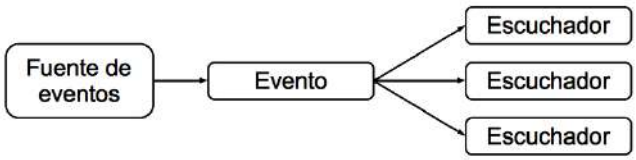

Figura 11.3: Dinámica de la detección de eventos en Swing.



Figura 11.4: Dinámica de la detección de eventos para el caso particular de `WindowEvent`.

1. Conocer cual es el tipo de eventos genera el *Contenedor* o *Componente* que nos interesa.
2. Conocer cual es el `interface` capaz de escuchar a ese tipo de eventos y escribir una clase que lo implemente. A esta clase la llamaremos clase escuchadora.
3. Añadir una instancia de la clase escuchadora a la lista de observadores del *Contenedor* o el *Componente*.

Para ver de modo detallado como los pasos anteriores se concretan en código, supongamos que deseamos cerrar la aplicación cuando el usuario haga click sobre el botón de cierre de la ventana. El primer paso es conocer qué tipo de eventos lanza `JFrame`, que es nuestra ventana, cuando el usuario hace click sobre el botón correspondiente; para ello podemos consultar la página `http://download.oracle.com/docs/cd/E17409_01/javase/tutorial/uiswing/events/eventsandcomponents.html`, donde encontramos que `JFrame` lanza eventos de tipo `WindowEvent`.

El siguiente paso es conocer cual es el `interface` capaz de escuchar este tipo de eventos. En la anterior dirección web encontramos que es `WindowListener`[2]. Este `interface` declara métodos cuyo significado aparece en la Tabla 11.1. La Figura 11.4 muestra la dinámica en la detección de eventos generados por `JFrame`.

La clase que implemente este `interface` debe definir cada uno de los métodos de la Tabla 11.1. Si lo único que queremos hacer es cerrar la aplicación cuando el usuario cierra la ventana, el único método al que añadiremos código es `public void windowClosing(WindowEvent e)`.

Finalmente añadiremos una instancia de la clase que implementa el `interface WindowListener` como escuchador a la ventana, con el método `addWindowListener`[3].

[2]Observa la nomenclatura usada en el nombrado de eventos y sus correspondientes escuchadores: si encontramos un evento de tipo xxxEvent, el `interface` capaz de escucharlo se llamará xxxListener

[3]Fíjate de nueva en la nomenclatura utilizada para nombrar el método que añade el escuchador al *Contenedor*, si el escuchador es xxxListener, el método que lo añade al *Contenedor* o *Componente* es addxxxListener(interface xxxListener)

`windowOpened(WindowEvent e)`	Se invoca cuando la ventana se abre.
`windowClosing(WindowEvent e)`	Se invoca cuando se intenta cerrar la ventana.
`windowClosed(WindowEvent e)`	Se invoca cuando la ventana se ha cerrado definitivamente.
`windowIconified(WindowEvent e)`	Se invoca cuando la ventana se minimiza.
`windowDeiconified(WindowEvent e)`	Se invoca cuando la ventana pasa de estar minimizada a tener su estado normal.
`windowActivated(WindowEvent e)`	Se invoca cuando la ventana pasa a ser la ventana activa.
`windowDeactivated(WindowEvent e)`	Se invoca cuando la ventana deja de ser la ventana activa.

Tabla 11.1: Métodos declarados en la `interface WindowListener` todos ellos son `public void`.

El Listado 11.2 muestra un ejemplo completo de un escuchador para los eventos de la ventana que cierra la aplicación cuando el usuario pulsa el botón de cerrar ventana.

```
1 package gui;
2
3 import java.awt.event.WindowEvent;
4 import java.awt.event.WindowListener;
5
6 import javax.swing.JFrame;
7 import javax.swing.SwingUtilities;
8 // Esta clase implementa WindowListener luego es escuchador de
      WindowEvent
9 public class EjemploWindowListener implements WindowListener {
10  private EjemploWindowListener() {
11   super();
12  }
13
14  private void creaGUI() {
15   // Creamos la ventana
16   JFrame ventana = new JFrame("Aplicación que se cierra con la ventan.")
        ;
17   // Añadimos como escuchador la instancia de esta clase
18   ventana.addWindowListener(this);
19   ventana.setSize(400, 400);
20   ventana.setVisible(true);
21  }
22
23  public static void main(String[] args) {
24   SwingUtilities.invokeLater(new Runnable() {
25    @Override
26    public void run() {
27     new EjemploWindowListener().creaGUI();
28    }
29   });
30  }
31
32  // Los métodos que siguen están declarados en la interface
       WindowListener
33  @Override
34  public void windowOpened(WindowEvent e) {
35  }
36
```

```
37  // Este es el único método con código
38  @Override
39  public void windowClosing(WindowEvent e) {
40   System.out.println("Cerrando la ventana...");
41   System.exit(0);
42  }
43
44  @Override
45  public void windowClosed(WindowEvent e) {
46  }
47
48  @Override
49  public void windowIconified(WindowEvent e) {
50  }
51
52  @Override
53  public void windowDeiconified(WindowEvent e) {
54  }
55
56  @Override
57  public void windowActivated(WindowEvent e) {
58  }
59
60  @Override
61  public void windowDeactivated(WindowEvent e) {
62  }
63 }
```

Listado 11.2: Aplicación que finaliza cuando se cierra la ventana. La clase principal implementa el `interface WindowListener`, por lo que se puede añadir como escuchador de eventos `WindowEvent`.

Si el Listado 11.2 te ha parecido tedioso, ya que hemos tenido que definir todos los métodos dejándolos vacíos excepto el método `windowClosing(WindowEvent e)`, tu sensación es acertada. En los casos en los que una `interface` tiene declarados muchos métodos, de los que usualmente sólo se escribe código para algunos de ellos, Java nos proporciona un clase de conveniencia, llamada adaptadora, que implementa la `interface` definiendo todos los métodos vacíos. ¿Cual es la ventaja de utilizar estas clases adaptadoras?, pues que nuestros escuchadores en vez de implementar el `interface` extienden la clase adaptadora, y sólo sobrescriben los métodos necesarios, la implementación del resto de métodos será la que nos proporcione la clase adaptadora, es decir, serán todos vacíos. El Listado 11.3 muestra un ejemplo cuyo comportamiento es el mismo que el ejemplo del Listado 11.2, pero utilizando una clase interna anónima que extiende la clase adaptadora `WindowAdapter`.

```
1 package gui;
2
3 import java.awt.event.WindowAdapter;
4 import java.awt.event.WindowEvent;
5
6 import javax.swing.JFrame;
7 import javax.swing.SwingUtilities;
8
9 public final class EjemploClaseAdaptadora {
10  private EjemploClaseAdaptadora() {
11   super();
12  }
13
14  private void creaGUI() {
15   // Creamos la ventana
16   JFrame ventana = new JFrame("Escuchador con clase adaptadora.");
17   // Añadimos como escuchador una instancia de una clase interna anónima
18   // que extiende a WindowAdapter y sólo sobrescribe el método
        windowClosing.
```

```
19   ventana.addWindowListener(new WindowAdapter() {
20    @Override
21    public void windowClosing(WindowEvent e) {
22     System.exit(0);
23    }
24   });
25   ventana.setSize(400, 400);
26   ventana.setVisible(true);
27  }
28
29  public static void main(String[] args) {
30   SwingUtilities.invokeLater(new Runnable() {
31    @Override
32    public void run() {
33     new EjemploClaseAdaptadora().creaGUI();
34    }
35   });
36  }
37 }
```

Listado 11.3: Uso de la clase adaptadora `WindowAdapter` para cerrar la aplicación al cerrar la ventana.

Por otro lado, la propia clases `JFrame` nos ofrece un método para definir el comportamiento de la aplicación cuando se cierra la ventana, este método es `setDefaultCloseOperation(int modo)`. El Listado 11.4 muestra un ejemplo de uso de este método.

```
1 package gui;
2
3 import javax.swing.JFrame;
4 import javax.swing.SwingUtilities;
5
6 public class EjemploSetDefaultCloseOperation {
7  private EjemploSetDefaultCloseOperation() {
8   super();
9  }
10
11  private void creaGUI() {
12   // Creamos la ventana
13   JFrame ventana = new JFrame("Escuchador con clase adaptadora.");
14   // Usamos el método de conveniencia setDefaultCloseOperation
15   ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16   ventana.setSize(400, 400);
17   ventana.setVisible(true);
18  }
19
20  public static void main(String[] args) {
21   SwingUtilities.invokeLater(new Runnable() {
22    @Override
23    public void run() {
24     new EjemploSetDefaultCloseOperation().creaGUI();
25    }
26   });
27  }
28 }
```

Listado 11.4: Uso del método `setDefaultCloseOperation(int)` para acabar la aplicación cuando se cierra la ventana.

Esta técnica de escucha de eventos es transversal a todos los *Contenedores* y *Componentes* que forman parte de Swing. Los *Contenedores* y *Componentes* lanzan eventos que seremos capaces de escuchar implementando el `interface` adecuado en una clase, y registrando esa clase como escuchador del *Contenedor* o *Componente*.

En la sección siguiente vamos a ver algunos de los *Componentes* Swing más

comunes y se mostrarán ejemplos de cómo escuchar los eventos que se producen cuando el usuario interacciona sobre ellos.

11.5. Algunos componentes Swing

Esta sección no pretende ser una presentación exhaustiva de *Componentes* Swing, si no una muestra de cómo utilizar la técnica que se ha mostrado en la sección anterior para escuchar los eventos que producen.

11.5.1. `JLabel`, muestra texto o iconos

El primer y más sencillo *Componente* Swing es `JLabel` que se muestra como una cadena de texto o un icono que el usuario no puede modificar, pero sí el programador. Este componente no lanza ningún tipo de evento, ya que el usuario, como hemos dicho, no puede interaccionar sobre él, nos sirve para mostrar texto o iconos.

11.5.2. `JButton`, botones que el usuario puede pulsar

El siguiente componente en la lista es `JButton` con el que podemos crear botones que el usuario puede pulsar. Cada vez que el usuario pulsa un `JButton`, este lanza un evento de tipo `ActionEvent`, si queremos escuchar este tipo de evento, necesitamos implementar la `interface ActionListener` que únicamente declara un método `public void actionPerformed(ActionEvent e)` que será invocado cada vez que el usuario pulse el botón. Finalmente, registraremos el escuchador al botón con el método `addActionListener(ActionListener escuchador)` de la clase `JButton`.

El Listado 11.5 muestra un ejemplo de detección de los eventos `ActionEvent`, donde también se ha incluido un `JLabel`. En la línea 21 creamos el botón como una instancia de `JButton`, en las líneas 22-27 añadimos al botón un escuchador como una clase interna anónima que implementa el `interface ActionListener` y definimos el método que declara esta interfaz `public void actionPerformed (ActionEvent e)`. Otro detalle interesante es que hemos utilizado un `JPanel` para añadir sobre él los componentes `JLabel` y `JButton` y aprovechar que el Gestor de Aspecto de un `JPanel` es **FlowLayout** y coloca los *Componentes* en el sentido de la escritura. Otro detalle nuevo es el uso del método `public void pack();` este método calcula el tamaño óptimo de la ventana para contener todos los componentes que se le han añadido, de este modo estamos delegando en Swing el cálculo del tamaño de la ventana.

```
1 package gui;
2
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5
6 import javax.swing.JButton;
7 import javax.swing.JFrame;
8 import javax.swing.JLabel;
9 import javax.swing.JPanel;
10 import javax.swing.SwingUtilities;
11
12 public class EjemploJButton {
13  private EjemploJButton() {
14   super();
```

```
15  }
16
17  private void creaGUI() {
18   JFrame ventana = new JFrame("Un JLabel y un JButton");
19   JPanel contenedor = new JPanel();
20   contenedor.add(new JLabel("Pulsa el botón:"));
21   JButton jbBoton = new JButton("Púlsame");
22   jbBoton.addActionListener(new ActionListener() {
23    @Override
24    public void actionPerformed(ActionEvent e) {
25     System.out.println("Botón pulsado");
26    }
27   });
28   contenedor.add(jbBoton);
29   ventana.getContentPane().add(contenedor);
30   ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31   ventana.pack();
32   ventana.setVisible(true);
33  }
34
35  public static void main(String[] args) {
36   SwingUtilities.invokeLater(new Runnable() {
37    @Override
38    public void run() {
39     new EjemploJButton().creaGUI();
40    }
41   });
42  }
43
44 }
```

Listado 11.5: Ejemplo de uso de `JButton`

11.5.3. `JTextField`, campos de introducción de texto

La clase `JTextField` dibuja una caja de edición de texto de una única línea donde el usuario puede introducir texto. En el momento de la instanciación de `JTextField` podemos indicar el número de columnas.

Un `JTextField` lanza eventos de tipo `ActionEvent` cada vez que el usuario pulsa el botón *Enter* y este componente tiene el foco. Este es el mismo evento que lanza un botón cuando el usuario lo pulsa, luego el procedimiento para escuchar los eventos es el mismo que en el caso de `JButton`. El Listado 11.6 muestra un ejemplo de uso de `JTextField`. Fíjate que esta vez en la línea 22 hemos optado por cambiar el Gestor de Aspecto del `JFrame` es vez de utilizar un `JPanel` intermedio. En la línea 29 hemos utilizado el método `String getText()` para obtener el texto introducido por el usuario en el `JTextField`.

```
1  package gui;
2
3  import java.awt.Container;
4  import java.awt.FlowLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JTextField;
11 import javax.swing.SwingUtilities;
12
13 public class EjemploJTextField {
14  private JTextField jtfTexto;
15
16  private EjemploJTextField() {
17   super();
18  }
```

```
19
20  public void creaGUI() {
21   JFrame ventana = new JFrame();
22   ventana.setLayout(new FlowLayout());
23   Container contenedor = ventana.getContentPane();
24   contenedor.add(new JLabel("Introduce un texto: "));
25   jtfTexto = new JTextField(50);
26   jtfTexto.addActionListener(new ActionListener() {
27    @Override
28    public void actionPerformed(ActionEvent e) {
29     System.out.println("El texto escrito es: " + jtfTexto.getText());
30    }
31   });
32   ventana.add(jtfTexto);
33   ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34   ventana.pack();
35   ventana.setVisible(true);
36  }
37
38  public static void main(String[] args) {
39   SwingUtilities.invokeLater(new Runnable() {
40    @Override
41    public void run() {
42     new EjemploJTextField().creaGUI();
43    }
44   });
45  }
46 }
```

Listado 11.6: Ejemplo de uso de `JTextField`

11.5.4. `JRadioButton`, botones de opciones

El siguiente *Componente* en complejidad es el botón de radio `JRadioButton` que dibuja un botón asociado a una opción, como por ejemplo para elegir la forma de pago Tarjeta/Transferencia/Cheque. Muchas veces las opciones presentadas al usuario son excluyentes, como en el caso anterior, de modo que seleccionar una de ellas implica que se de-selecciona la anterior si hubiese alguna.

`JRadioButton` puede lanzar dos tipos de eventos interesante, nuestro conocido `ActionEvent` y el nuevo `ItemEvent`. El evento `ItemEvent` nos da más información sobre lo que ha ocurrido que `ActionEvent`, ya que nos dice si lo que ha ocurrido es una selección o una des-selección del botón.

Ya sabemos que los eventos de tipo `ActionEvent` los podemos escuchar con una clase que implemente el `interface ActionListener` añadida al *Componente* que queremos escuchar con el método `addActionListener(ActionListener escuchador)`, y que este `interface` sólo declara un método `public void actionPerformed(ActionEvent e)` que se invoca cada vez que el botón se pulsa.

En el caso de un evento de tipo `ItemEvent` lo podemos escuchar con una clase que implemente el `interface ItemListener` siempre que añadamos la instancia al *Componente* que queremos escuchar con `addItemListener(ItemListener escuchador)`. Este `interface` sólo declara un método `public void itemStateChanged(ItemEvent e)` que se invoca cada vez que el usuario selecciona o des-selecciona un `JRadioButton`. Para conocer si lo que ha ocurrido es una selección o una de-selección, podemos utilizar el método `getStateChange()` de la clase `ItemEvent` que nos devolverá `ItemEvent.SELECTED` si lo que ha ocurrido es una selección, o `ItemEvent.DESELECTED` si lo que ha ocurrido es una de-selección.

```
1 package gui;
2
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import java.awt.event.ItemEvent;
7 import java.awt.event.ItemListener;
8
9 import javax.swing.BoxLayout;
10 import javax.swing.ButtonGroup;
11 import javax.swing.JFrame;
12 import javax.swing.JPanel;
13 import javax.swing.JRadioButton;
14 import javax.swing.SwingUtilities;
15
16 public final class EjemploJRadioButton {
17  private EjemploJRadioButton() {
18   super();
19  }
20
21  private JPanel creaContenedor(String posicion) {
22   JPanel contenedor = new JPanel();
23   contenedor.setLayout(new BoxLayout(contenedor, BoxLayout.Y_AXIS));
24   Escuchador escuchador = new Escuchador();
25   JRadioButton jrbMucho = new JRadioButton("Mucho");
26   jrbMucho.addActionListener(escuchador);
27   jrbMucho.addItemListener(escuchador);
28   contenedor.add(jrbMucho);
29   JRadioButton jrbNormal = new JRadioButton("Normal");
30   jrbNormal.addActionListener(escuchador);
31   jrbNormal.addItemListener(escuchador);
32   contenedor.add(jrbNormal);
33   JRadioButton jrbPoco = new JRadioButton("Poco");
34   jrbPoco.addActionListener(escuchador);
35   jrbPoco.addItemListener(escuchador);
36   contenedor.add(jrbPoco);
37   if(posicion == BorderLayout.EAST) {
38    ButtonGroup grupo = new ButtonGroup();
39    grupo.add(jrbMucho);
40    grupo.add(jrbNormal);
41    grupo.add(jrbPoco);
42   }
43   return contenedor;
44  }
45
46  private void creaGUI() {
47   JFrame ventana = new JFrame("Ejemplo JRadioButton");
48   ventana.add(creaContenedor(BorderLayout.WEST), BorderLayout.WEST);
49   ventana.add(creaContenedor(BorderLayout.EAST), BorderLayout.EAST);
50   ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51   ventana.pack();
52   ventana.setVisible(true);
53  }
54
55  private class Escuchador implements ActionListener, ItemListener {
56   public Escuchador() {
57    super();
58   }
59
60   @Override
61   public void actionPerformed(ActionEvent e) {
62    System.out.println("Botón pulsado");
63   }
64
65   @Override
66   public void itemStateChanged(ItemEvent e) {
67    String texto = ((JRadioButton)e.getSource()).getText();
68    if(e.getStateChange() == ItemEvent.DESELECTED)
69     System.out.format("Botón %s deseleccionado.\n", texto);
70    else if(e.getStateChange() == ItemEvent.SELECTED)
71     System.out.format("Botón %s seleccionado.\n", texto);
72   }
73  }
```

```
74
75  public static void main(String[] args) {
76   SwingUtilities.invokeLater(new Runnable() {
77    @Override
78    public void run() {
79     new EjemploJRadioButton().creaGUI();
80    }
81   });
82  }
83
84 }
```

Listado 11.7: Ejemplo de uso de `JRadioButton`

El Listado 11.7 muestra un ejemplo con este nuevo *Componente*. Además de lo ya comentado, fíjate que en las líneas 38-41 estamos utilizando una nueva clase `ButtonGroup`, esta clase agrupa los botones de manera excluyente, de modo que cuando alguno de los botones se selecciona, si hay algún otro botón previamente seleccionado este último se des-seleccionará. La clase `ButtonGroup` crea un grupo lógico, y no tiene ninguna representación gráfica. En la línea 67 puedes ver como hemos recuperado el texto escrito a la derecha de cada uno de los `JRadioButton`. Y finalmente, en la línea 23 hemos utilizado un nuevo Gestor de Aspecto, `BoxLayout` que nos permite disponer los *Componentes* verticalmente dentro de la ventana.

11.5.5. `JCheckBox`, botones de selección múltiple

Usualmente los botones de tipo `JRadioButton` se utilizan cuando las opciones presentadas al usuario son mutuamente excluyentes entre sí, y se han añadido a un `ButtonGroup` para comportarse de este modo. Si lo que queremos presentar al usuario son opciones no excluyentes solemos utilizar botones de tipo `JCheckBox`. Estos botones se dibujan como una pequeña caja que al seleccionarlo aparece marcada con un *tick*.

Los `JCheckBox` lanzan los mismos tipos de eventos que los `JRadioButton`, es decir eventos `ActionEvent` y eventos `ItemEvent` para indicar, estos últimos, si lo que ha ocurrido es una selección o una de-selección. Por lo tanto todo lo comentado en la sección 11.5.4 sobre los `JRadioButton` es válido para los `JCheckBox`.

11.5.6. `JList`, listas de selección

La clase `JList` presentan al usuario varias opciones en forma de lista. El usuario puede seleccionar una o más opciones dependiendo del modo de selección de la lista.

Los eventos que un `JList` puede lazar cada vez que el usuario selecciona una opción de la lista son nuestro conocido `ActionEvent` y el nuevo `ListSelectionEvent`. Este evento nos indica si la selección se está efectuando (por ejemplo, el usuario pulsa sobre un elemento de la lista y, sin soltar el botón del ratón, se desplaza sobre los elementos de la lista), o es la acción final, cuando el usuario suelta el botón del ratón.

Para escuchar los eventos de tipo `ItemSelectionEvent` debemos implementar la `interface ItemSelectionListener` que declara un único método `public void valueChanged(ListSelectionEvent e)`. Para consultar si

la selección está en marcha o es definitiva podemos usar el método `getValueIsAdjusting()` de la clase `ItemSelectionEvent`.

El Listado 11.8 muestra un ejemplo de uso de este componente. Otros detalles interesantes de este ejemplo son el uso del método `setVisibleRowCount(int)` de la línea 25 para indicar cuantos elementos son visibles en la lista. En la línea 23 activamos el modo de selección de los elementos de la lista a `ListSelectionModel.SINGLE_SELECTION`, de modo que sólo se podrá seleccionar un elemento único de la lista (otros modos posible son `SINGLE_INTERVAL_SELECTION` y `MULTIPLE_INTERVAL_SELECTION`).

```
1 package gui;
2
3 import java.awt.Container;
4
5 import javax.swing.JFrame;
6 import javax.swing.JList;
7 import javax.swing.JScrollPane;
8 import javax.swing.ListSelectionModel;
9 import javax.swing.SwingUtilities;
10 import javax.swing.event.ListSelectionEvent;
11 import javax.swing.event.ListSelectionListener;
12
13 public final class EjemploJList {
14  private EjemploJList() {
15   super();
16  }
17
18  private void creaGUI() {
19   JFrame ventana = new JFrame("Ejemplo JList");
20   Container contenedor = ventana.getContentPane();
21   JList opciones = new JList(new String[]{"Lunes", "Martes", "Miércoles"
        , "Jueves", "Viernes", "Sábado", "Domingo"});
22   opciones.setVisibleRowCount(5);
23   opciones.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
24   opciones.addListSelectionListener(new Escuchador());
25   JScrollPane scroll = new JScrollPane(opciones);
26   contenedor.add(scroll);
27   ventana.pack();
28   ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29   ventana.setVisible(true);
30  }
31
32  private class Escuchador implements ListSelectionListener {
33   @Override
34   public void valueChanged(ListSelectionEvent e) {
35    if(e.getValueIsAdjusting() == true)
36     System.out.println("Item en curso: " + ((JList)e.getSource()).
          getSelectedValue());
37    else if(e.getValueIsAdjusting() == false)
38     System.out.println("Item definitivo: " + ((JList)e.getSource()).
          getSelectedValue());
39   }
40  }
41
42  public static void main(String[] args) {
43   SwingUtilities.invokeLater(new Runnable() {
44    @Override
45    public void run() {
46     new EjemploJList().creaGUI();
47    }
48   });
49  }
50
51 }
```

Listado 11.8: Ejemplo de uso de `JList`

Por defecto, el *Componente* `JList` no tiene una barra de desplazamiento para poder visualizar los elementos de la lista, si queremos que la lista posea uno, lo tenemos que añadir tal y como se muestra en las líneas 25 y 26. Fíjate que indicamos el *Componente* al que asociaremos la barra de desplazamiento en el momento de crear esta. Finalmente añadimos la barra de desplazamiento a la ventana y no la lista original.

Otros métodos interesantes del *Componente* `JList` son `Object getSelectedValue()` el elementos actualmente seleccionado, si la lista es de selección única; y `Object [] getSelectedValues()` si se lista es de selección múltiple.

Con el componente `JList` acabamos la breve muestra de las posibilidades de creación de interfaces gráficos de usuario *Swing*. Esta sección no pretende ser un exposición exhaustiva de todas las posibilidades que nos proporciona *Swing*, lo que pretende mostrar es la técnica de cómo programar interfaces gráficos de usuario con el patrón de diseño *Observable*.

11.6. El patrón de diseño Modelo/Vista/Controlador

Quizás el patrón de diseño Modelo/Vista/Controlador sea uno de los más utilizados en el desarrollo de proyectos informáticos, tanto es así que incluso existe una adaptación al mundo de aplicaciones web de este patrón de diseño.

Este patrón de diseño define tres actores con las siguientes responsabilidades:

Modelo es el responsable de mantener y gestionar los datos de la aplicación.

Vista es la responsable del interfaz gráfico de usuario y la detección de eventos sobre los componentes.

Controlador es quien hace corresponder la interacción del usuario con posible cambios en el Modelo.

Veamos , con un ejemplo, el papel de cada uno de estos actores. En la Figura 11.5 se muestra el interfaz gráfico de una aplicación que calcula la cuota mensual de una hipoteca. El usuario puede introducir los tres datos que se necesita para el cálculo en tres cajas de edición de texto, y cuando pulsa el botón *Calcula* aparece la nueva cuota en la parte inferior de la ventana.

En este caso, el *Modelo* contiene los datos de la hipoteca: cantidad hipotecada, duración de la hipoteca, interés del préstamo y la cuota mensual. La *Vista* es la encargada de crear el interfaz gráfico y la detección de los eventos sobre el interfaz. El *Controlador* sabe que cuando el usuario pulsa el botón *Calcula*, debe leer los datos de la hipoteca y enviárselos al *Modelo* para que este haga el cálculo.

En la Figura 11.6 se muestra la dinámica de este patrón de diseño, que se detalla en los siguiente pasos:

1. El usuario interacciona sobre la *Vista*.

2. La *Vista* informa al *Controlador* de lo ocurrido.



Figura 11.5: Un interfaz gráfico para el cálculo de la cuota mensual de una hipoteca.

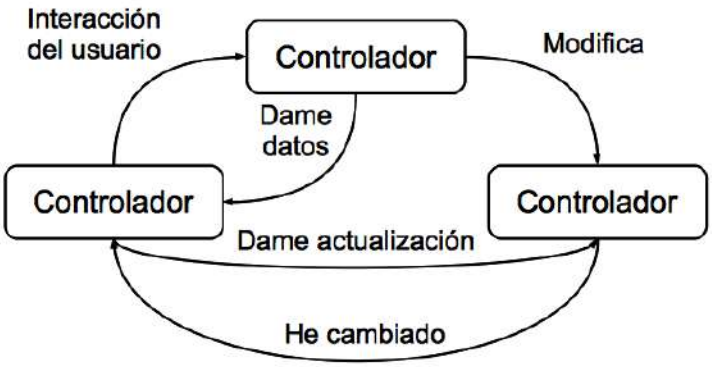

Figura 11.6: Dinámica del modelo MVC.

3. El *Controlador* decide que datos necesita de la *Vista* para llevar a cabo la tarea como respuesta a la interacción del usuario.

4. El *Controlador* actualiza el *Modelo*.

5. El *Modelo* informa a la *Vista* de que se ha actualizado.

6. La *Vista* pide los datos de su interés para visualizarlos.

En el ejemplo del cálculo de la cuota mensual de una hipoteca esta dinámica se concretaría del siguiente modo:

1. El usuario introduce la cantidad, el tiempo y el interés de la hipoteca y pulsa el botón *Calcula*.

2. La *Vista* informa al *Controlador* de que el usuario ha pulsado el botón *Calcula*.

3. La lógica del negocio programada en el *Controlador* indica que si el usuario pulsa el botón *Calcula* se debe recuperar la cantidad, el tiempo y el interés de la hipoteca que están en la *Vista*.

4. El *Controlador* envía estos datos al *Modelo* para que calcule la nueva cuota.

5. El *Modelo* calcula la nueva cuota e informa de ello a la *Vista*

6. La *Vista* pide la nueva cuota y la visualiza.

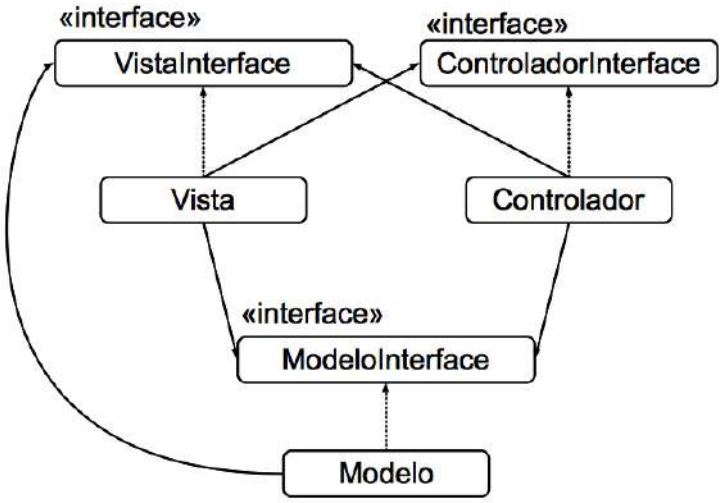

Figura 11.7: Diagrama UML para el patrón de diseño MVC.

Aunque en un primer momento, este patrón puede resultar farragoso, o parecernos que hay muchas idas y venidas entre el código de los actores, es todo lo contrario, gracias a esta división de responsabilidades los posibles cambios en la implementación de uno de los actores es completamente transparente al resto. Imagina por ejemplo que hay un cambio en la *Vista*, si después de cambiar, la *Vista* sigue proporcionándonos los datos sobre los que efectuar el cálculo y le podemos seguir informando de los cambios en el *Modelo* para que se actualice, este cambio será totalmente transparente tanto para el *Controlador* como para el *Modelo*.

Fíjate que para poder implementar este patrón de diseño, la *Vista* debe conocer tanto al *Controlador* como al *Modelo*, y por su parte el *Controlador* debe conocer tanto a la *Vista* como al *Modelo*. Por su lado, el único actor que necesita conocer el *Modelo* es a la *Vista*, de hecho, en la Figura 11.6 no hay ninguna flecha que salga desde el *Modelo* hacia el *Controlador*. Que un actor tenga conocimiento de los otros implica que tendrá una referencia hacia el actor con el que necesita intercambiar mensajes. Para dejar aún más clara la potencia de este patrón, vamos a implementar las referencias a una `interface` y no a una clase concreta, es decir, la funcionalidad que un actor ofrece la vamos a recoger en un `interface` y además tendremos la clase concreta que implementará el `interface`, la Figura 11.7 muestra el esquema de clases citado.

En el Apéndice B se encuentra el código fuente de la aplicación del cálculo de la cuota mensual de una hipoteca.

Cuestiones.

1. ¿Cuales son las ventajas y desventajas de utilizar el método de conveniencia `setDefaultCoseOperation(JFrame.EXIT_ON_CLOSE)` para cerrar la aplicación cuando el usuario cierra la ventana?.

2. El código del cálculo de la cuota mensual de una hipoteca mostrado en el Apéndice B sigue la estrategia de no enviar datos desde la *Vista* al *Controlador* cuando se produce el evento de pulsar el botón *Calcula*, es decir, la *Vista* no envía la cantidad hipoteca, el tiempo ni el interés, es el *Controlador* quien pide estos datos una vez que la *Vista* le informa de que el usuario pulsó el botón *Calcula*. Lo mismo ocurre cuando el *Modelo* cambia su estado, al informar a la *Vista* no le envía el nuevo valor de la cuota mensual de la hipoteca, simplemente le informa de que hay un nuevo valor disponible y es finalmente la *Vista* quien pide el nuevo valor al *Modelo*.

   ¿Cuales son las ventajas de esta aproximación?. ¿Cuales son las desventajas?.

Ejercicios.

1. Recupera el ejercicio de la agenda telefónica y crea un interfaz gráfico de usuario para ella.

Lecturas recomendadas.

- El capítulo 1 de la referencia [8] presenta el patrón de diseño MVC haciendo referencia al resto de patrones que utiliza.
- La referencia [3] también presenta de modo riguroso y muy ameno el patrón de diseño MVC.

Capítulo 12

Applets

Contenidos

12.1. ¿Qué son los Applets? **173**
12.2. Ciclo de vida de un Applet **174**
12.3. Código HTML para contener un Applet **175**
12.4. Lectura de parámetros de la página HTML **176**
12.5. Convertir una aplicación Swing en un Applet . . **176**
12.6. Comunicación entre Applets **177**

Introducción

Para muchos de nosotros, el primer contacto con Java fue a través de los Applets, esos pequeños programas que se ejecutan dentro de una página web y con los que se puede interaccionar. Algunos ejemplos se pueden encontrar en las siguientes direcciones: `http://www.falstad.com/mathphysics.html`, `http://openastexviewer.net/web/thinlet.html`.

En este capítulo se va a presentar la programación de Apples, sus características, particularidades, y cómo, con poco trabajo extra podemos convertir nuestras aplicaciones Swing en Applets siempre que se cumpla una serie de restricciones.

12.1. ¿Qué son los Applets?

Los Applets son aplicaciones Java que se ejecutan en el contexto de un navegador web. A través de código HTML reservamos una zona de la página web para visualizar el Applet, y es el navegador web el encargado de descargar las clases del Applet desde la url especificada, iniciar una instancia de la máquina virtual de Java y ejecutar el Applet.

La seguridad, en el caso de los Applets, es un importante factor a tener en cuenta. Para ejecutar un Applet es necesario descargar desde la web las clases que se ejecutarán en nuestra máquina. Fíjate el riesgo que en principio se corre si no hubiese restricciones de seguridad, descargas un programa que no sabes

quien ha escrito ni con qué propósito y lo ejecutas en tu máquina. Si no hubiesen restricciones de seguridad un programa malicioso podría acceder a tu disco duro para leer información personal, o podría borrar o modificar ficheros, o escribir en tu disco duro.

Por todo lo anterior, un Applet tiene las siguientes restricciones de seguridad:

- Un Applet no puede leer del disco duro del cliente.
- Un Applet no puede escribir al disco duro del cliente.
- Un Applet no puede abrir conexiones de red a ningún otro servidor salvo aquel desde el que se descargó.
- Un Applet no puede ejecutar aplicaciones en el cliente.
- Un Applet no puede acceder a la información privada del usuario.

Estas restricciones de seguridad y el hecho de que finalmente el Applet se ejecutará en un Máquina Virtual Java los hacen muy seguros.

12.2. Ciclo de vida de un Applet

Un Applet se ejecuta en el contexto de un navegador web y tiene fuertes restricciones de seguridad, tal y como hemos visto. El hecho de que los Applets se ejecuten en el contexto de un navegador web implica que su ciclo de vida no es el de una aplicación de escritorio, como las que ya hemos aprendido a programar. El ciclo de vida de un Applet está directamente relacionado con las llamadas que el navegador web hace a métodos del Applet.

Para que una de nuestras clases sea un Applet debe extender a la clase `JApplet` que define los siguientes métodos relacionados con su ciclo de vida:

- `public void init()`, el navegador web llama a este método cuando el Applet ha sido efectivamente cargado. Este método es el primero que se invoca en un Applet.
- `public void start()`, el navegador web llama a este método para indicarle que debe empezar su ejecución.
- `public void paint(Graphics g)`, el navegador web llama a este método cada vez que se debe dibujar el contenido del Applet, y nos permite el acceso al contexto gráfico de bajo nivel `Graphics`. Un detalle muy importante es que desde nuestro código nunca llamaremos directamente a este método, para forzar su llamada utilizaremos el método `public void repaint()`.
- `public void stop()`, el navegador web llama a este método para indicar que el Applet debe detener su ejecución, por ejemplo, cuando se abandona la página web que contiene el Applet.
- `public void destroy()`, el navegador web llama a este método antes de eliminar el Applet de memoria, en cuyo caso se llamará previamente al método `stop()`.

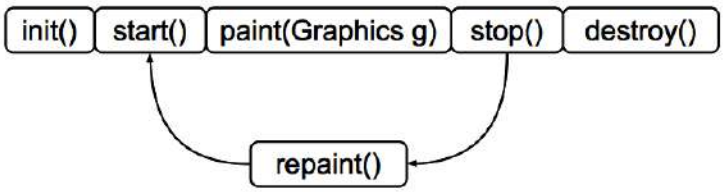

Figura 12.1: Llamada a los métodos de un Applet durante su ciclo de vida.

La Figura 12.1 muestra gráficamente el orden de las llamadas entre los métodos que constituyen el ciclo de vida de un Applet.

Para programar de modo eficiente un Applet debemos seguir estas reglas: los recursos que el Applet necesita para su ejecución deben ser creados en su método `init()`, y esos mismos recursos deben ser liberados en su método `destroy()`. Nunca llamaremos, desde nuestro código, al método `paint(Graphics g)` para forzar el dibujado del Applet, para ellos utilizaremos el método `repaint()`.

12.3. Código HTML para contener un Applet

Para poder visualizar un Applet al cargar una página HTML debemos utilizar la etiqueta `<html>` como se muestra en el Listado 12.1. El atributo `achive` nos sirve para indicar el fichero empaquetado de nuestra aplicación; con la etiqueta `code` especificamos la clase que implementa el Applet, como veremos en la siguiente sección; y mediante las etiquetas `width` y `height` especificamos el ancho y alto dentro de la página web reservado para la visualización del Applet.

```
<html>
 <head>
  <title>El primer applet</title>
 </head>
 <body>
  <applet
   archive="hipoteca.jar"
   code="applets.hipoteca.HipotecaApplet"
   width=519
   height=65>
   Si puedes ver esto tu navegador no soporta Java.
  </applet>
 </body>
</html>
```

Listado 12.1: Código HTML que muestra la aplicación de la hipoteca dentro de una página web.

Otros atributos de la etiqueta `<applet>` que nos pueden ser de utilidad son:

- `alt`, muestra un texto alternativo si el navegador no puede visualizar el Applet.
- `align`, el alineado del Applet dentro de la página web.
- `hspace`, el margen a la izquierda y derecha del Applet en unidades de píxeles.

- `vspace`, el margen superior e inferior del Applet en unidades de píxeles.
- `name`, el nombre del Applet. Etiqueta importante en la comunicación entre Applets que residen en la misma página web, como veremos en la Sección 12.6.

12.4. Lectura de parámetros de la página HTML

La etiqueta `<applet>` puede contener otras etiquetas de interés además de las que ya hemos visto. Con la etiqueta `<param>` especificamos un parámetro con su valor, por ejemplo `<param name="saludo"value="Hola/>`. Desde el código de nuestro Applet podremos leer los parámetros definidos dentro de la etiqueta Applet con el método `String getParameter(String nombreParametro)` que recibe como argumento el nombre del parámetro que queremos leer (`"saludo"` en el ejemplo anterior). Esto nos permite definir parámetros de entrada a nuestro Applet sin necesidad de modificar el código de nuestro Applet, en vez de ello, los definiremos en el código HTML que contiene al Applet.

En la Sección 12.6 se mostrará cómo hacer uso de esta técnica.

12.5. Convertir una aplicación Swing en un Applet

Un Applet, a efectos prácticos, es una aplicación Java con la particularidad de que se ejecuta en el contexto de un navegador web. Un Applet tiene una zona, dentro de la página web, donde se va a visualizar, y lo que podemos visualizar es, entre otras cosas un interfaz gráfico de usuario. Dicho de otro modo, si una aplicación Swing cumple con las restricciones de seguridad impuestas a los Applets, podremos, con pocas modificaciones, transformarla en un Applet.

Este es un sencillo recetario para convertir una aplicación Swing en un Applet:

1. No podemos hacer uso de `JFrame`, en vez de esta clase utilizaremos `JApplet`.
2. Un Applet no tiene constructores, el código dentro del constructor en la aplicación Swing lo escribiremos dentro del método `public void init()` del Applet.
3. No se pueden utilizar métodos de `JFrame` relativos al tamaño de la ventana (`setSize(...)`), al título de esta (`setTitle(String titulo)`, o su posición (`setLocation(int, int`), ya que la posición y el tamaño del Applet se especifican dentro del código HTML.
4. Un Applet no puede tener escuchadores de tipo `WindowListener`.

Siguiendo estos sencillos pasos, el Listado 12.2 muestra cómo convertir la aplicación Swing del Listado B.7 del Apéndice B del cálculo de la cuota mensual de una hipoteca en un Applet.

```
1 package applets.hipoteca;
2
```

```
3 import javax.swing.JApplet;
4
5 import gui.hipoteca.controlador.Controlador;
6 import gui.hipoteca.controlador.ControladorImpl;
7 import gui.hipoteca.modelo.Modelo;
8 import gui.hipoteca.modelo.ModeloImpl;
9 import gui.hipoteca.vista.Vista;
10 import gui.hipoteca.vista.VistaImpl;
11
12 public class HipotecaApplet extends JApplet {
13  private static final long serialVersionUID = 1L;
14
15  @Override
16  public void init() {
17   Vista vista = new VistaImpl();
18   Modelo modelo = new ModeloImpl();
19   Controlador controlador = new ControladorImpl();
20   modelo.setVista(vista);
21   vista.setControlador(controlador);
22   vista.setModelo(modelo);
23   controlador.setModelo(modelo);
24   controlador.setVista(vista);
25
26   setContentPane(vista.getContenedor());
27  }
28 }
```

Listado 12.2: Applet con la aplicación del cálculo de la cuota mensual de una hipoteca

Como tuvimos cuidado de aislar todo lo relativo al actor *Vista* siguiendo el patrón de diseño MVC, la transformación de aplicación Swing a Applet ha sido muy sencilla, de hecho, hemos podido aprovechar todas las clases dentro los paquetes `gui.hipoteca.modelo`, `gui.hipoteca.vista` y `gui.hipoteca.controlador`.

12.6. Comunicación entre Applets

Applets que residen dentro de la misma página web pueden obtener referencias a los otros Applets dentro de la misma página y a través de estas referencias un Applet puede llamar a los métodos de otro Applet dentro de la misma página web. Para ello utilizaremos el método `Applet getApplet(String nombre)` que recibe como argumento el nombre del Applet del que queremos obtener una referencia. Recuerda que el nombre de un Applet lo podemos definir con el atributo `name` de la etiqueta `<applet>`. Este método pertenece al contexto donde se está ejecutando el Applet, que no es otra cosa que la propia página web. Para obtener este contexto desde un Applet utilizamos el método `public AppletContext getAppletContext()`.

El Listado 12.3 muestra un sencillo ejemplo de comunicación entre Applets residentes en la misma página web mostrado en el Listado 12.4. La Figura 12.2 muestra cómo se visualiza la página web en un navegador.

```
1 package applets.comunicacion;
2
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6
7 import javax.swing.JApplet;
8 import javax.swing.JButton;
9 import javax.swing.JLabel;
```

```
10 import javax.swing.JPanel;
11 import javax.swing.JTextField;
12
13 public class ComunicacionApplets extends JApplet {
14  private static final long serialVersionUID = 1L;
15  private JTextField jtfTuDices;
16  private JLabel jlElDice;
17  private ComunicacionApplets elOtro = null;
18  JPanel contenedor = new JPanel(new BorderLayout());
19
20  @Override
21  public void init() {
22   JPanel miEntrada = new JPanel();
23   miEntrada.add(new JLabel("Tú dices:"));
24   jtfTuDices = new JTextField(50);
25   miEntrada.add(jtfTuDices);
26   JButton jbEnviar = new JButton("Enviar");
27   jbEnviar.addActionListener(new Escuchador());
28   miEntrada.add(jbEnviar);
29   JPanel suEntrada = new JPanel();
30   jlElDice = new JLabel("Escuchando....");
31   suEntrada.add(jlElDice);
32   contenedor.add(suEntrada, BorderLayout.SOUTH);
33   contenedor.add(miEntrada, BorderLayout.NORTH);
34   setContentPane(contenedor);
35  }
36
37  public void recibeMensaje(String mensaje) {
38   jlElDice.setText("Mensaje: " + mensaje);
39   repaint();
40  }
41
42  private class Escuchador implements ActionListener {
43   @Override
44   public void actionPerformed(ActionEvent e) {
45    if(elOtro == null)
46     elOtro = (ComunicacionApplets)getAppletContext().getApplet(
           getParameter("ElOtro"));
47    elOtro.recibeMensaje(jtfTuDices.getText());
48   }
49  }
50
51 }
```

Listado 12.3: Código HTML de la página visualizada en la Figura 12.2

```
1 <html>
2  <head>
3   <title>Conversación entre Applets</title>
4  </head>
5
6  <body>
7   <applet
8    archive="comunicacionApplets.jar"
9    code="applets.comunicacion.ComunicacionApplets"
10   width=800
11   height=70
12   name="Superior">
13   <param
14    name="ElOtro"
15    value="Inferior" />
16   Si puedes ver esto tu navegador no soporta Java.
17  </applet>
18  <br />
19  <applet
20   archive="comunicacionApplets.jar"
21   code="applets.comunicacion.ComunicacionApplets"
22   width=800
23   height=70
24   name="Inferior">
25   <param
26    name="ElOtro"
```

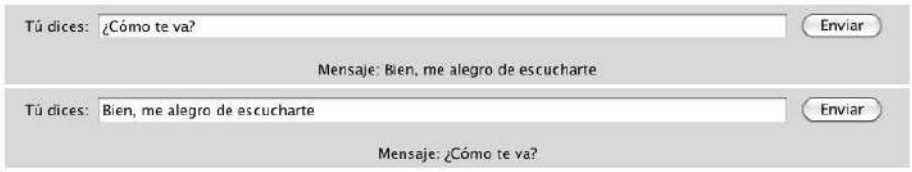

Figura 12.2: Llamada a los métodos de un Applet durante su ciclo de vida.

```
27      value="Superior"/>
28     Si puedes ver esto tu navegador no soporta Java.
29    </applet>
30  </body>
31 </html>
```

Listado 12.4: Applet con la aplicación del cálculo de la cuota mensual de una hipoteca

Ejercicios.

1. Convierte la aplicación de la Agenda en un Applet para que puedas interaccionar con ella a través de un navegador web.

Lecturas recomendadas.

- En esta dirección web `http://download.oracle.com/javase/tutorial/deployment/applet/index.html` podrás encontrar toda la información necesaria para programar Applets.

Capítulo 13

Control de errores con *MyLyn* y *Bugzilla*

Contenidos

13.1. Sistema de control de tareas *MyLyn* 182
13.1.1. Cual es el objetivo de *MyLyn* 182
13.1.2. Trabajar con *MyLyn* 182
13.1.2.1. Trabajar con tareas 182
13.1.2.2. Trabajar con categorías 186
13.1.2.3. Trabajar con *Working Sets* 187
13.2. Sistema de gestión de errores *Bugzilla* 188
13.2.1. Cual es el objetivo de *Bugzilla* 188
13.2.2. Instalación de *Bugzilla* 188
13.2.3. Trabajar con *Bugzilla* 195
13.3. Acceso a *Bugzilla* desde *MyLyn* y *Eclipse* 199
13.3.1. Beneficios de la combinación de *Bugzilla* y *MyLyn* desde *Eclipse* 201
13.3.2. Trabajo con *MyLyn* y *Bugzilla* desde *Eclipse* 201
13.3.2.1. Añadir errores a *Bugzilla* desde *Eclipse* . . 201
13.3.2.2. Recuperar errores desde *Bugzilla* como tareas en *MyLyn* 201

Introducción

Los *Entornos de Desarrollo Integrado* son una excelente herramienta para la creación y mantenimiento de código. Sin embargo, cuando se trabaja en proyectos cuyo código está organizado en un gran número de clases dentro de sus correspondiente paquetes, navegar por le código para encontrar un método o una clase puede llegar a consumir mucho tiempo. *MyLyn* es una herramienta, integrada en *Eclipse* a través de un *plug-in*, que oculta el código que no es relevante a la tarea que estamos llevando a cabo. Aunque *MyLyn* es mucho más que eso.

Por otro lado, una tarea importante en todo proyecto informática es el seguimiento y gestión de errores en el código. *Bugzilla* es una excelente herramienta que cumple esta función.

Quizás, lo mejor de ambas herramientas es la posibilidad de combinarlas dentro de *Eclipse* de tal manera que podemos inicial el seguimiento de un error en *Bugzilla* desde el propio *Eclipse* y darlo de alta como una tarea en *MyLyn* para poder centrar nuestra atención sólo en el código relativo a ese error.

13.1. Sistema de control de tareas *MyLyn*

MyLyn es una potente herramienta que se encuentra en el paquete básico de *Eclipse*. Al descargarnos *Eclipse* nos estamos descargando también esta herramienta.

13.1.1. Cual es el objetivo de *MyLyn*

Cuando trabajamos en equipo y en grandes proyectos es usual que el código fuente del proyecto esté organizado en un gran número de paquetes, dentro de los cuales encontramos un gran número de clases. El número total de ficheros con código fuente puede ser muy elevado. Generalmente, cuando desarrollamos una tarea dentro de un proyecto de gran envergadura, sólo son relevantes a esta tarea un número reducido de ficheros frente al total. Si el entorno de desarrollo que estamos utilizando nos presenta todos los ficheros del proyecto, podemos perder bastante tiempo navegando por ellos y cribando los que sí son relevantes a nuestra tarea.

En esencia, el objetivo de *MyLyn* es permitir concentrarnos sólo en el código de un proyecto sobre el que estamos trabajando, ocultando el resto del código del proyecto que no es relevante a la tarea que estamos llevando a cabo. Pero *MyLyn* no es sólo eso, si no un sistema de control de trabajo que podemos utilizar bien de forma local o bien en conexión con un sistema de control de errores. Y es en este último caso cuando *Bugzilla* se convierte en una herramienta imprescindible.

En *MyLyn* la unidad básica de trabajo es la tarea. Las tareas se pueden organizar en categorías y estas a su vez se pueden agrupar en grupos de trabajo. Veamos cómo trabajar con cada uno de estos nuevos conceptos.

13.1.2. Trabajar con *MyLyn*

MyLyn dispone de una vista propia en *Eclipse*. El nombre de esta vista es **Task List**. Si no aparece esta vista por defecto, puedes hacerla visible seleccionando la opción que se muestra en la Figura 13.1.

La nueva vista que se abrirá en *Eclipse* se muestra en la Figura 13.2.

13.1.2.1. Trabajar con tareas

Lo primero que vamos a hacer es crear una nueva tarea, para ello, despliega el icono que se muestra en la Figura 13.3 y selecciona la opción *New Task...*.

A continuación se nos abrirá la ventana que se muestra en la Figura 13.4 donde debemos seleccionar la opción por defecto *Local*. Lo que estamos haciendo en este punto es indicarle a *MyLyn* que la tarea que estamos creando es local a

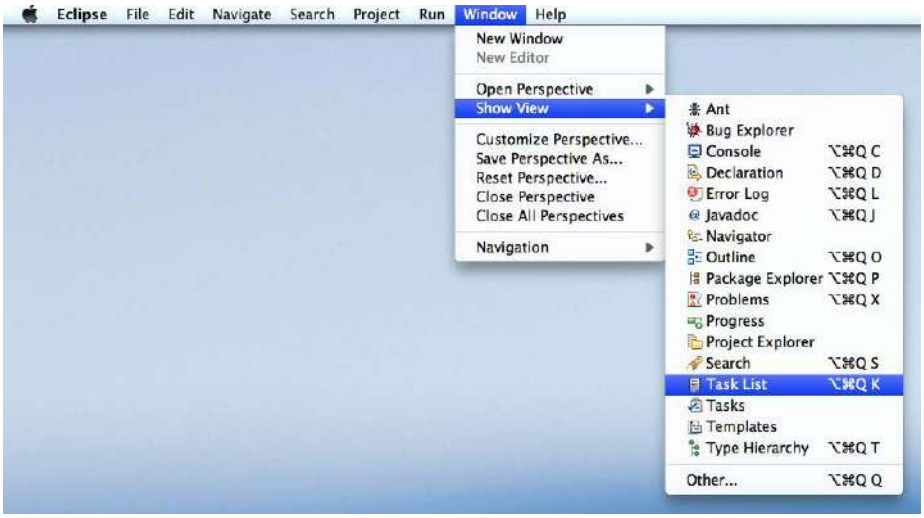

Figura 13.1: Opción de menú para abrir la vista de *MyLyn*.

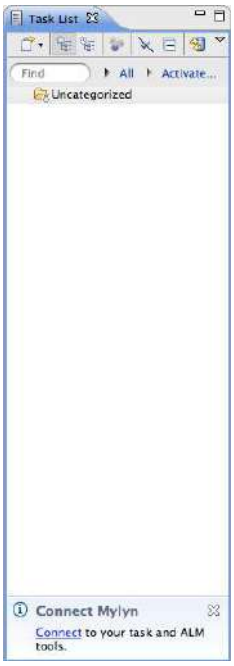

Figura 13.2: Aspecto de la vista **Task List** de *MyLyn*.

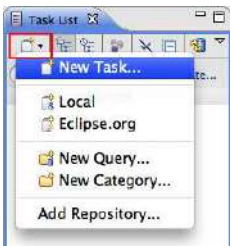

Figura 13.3: Creación de una nueva tarea en *MyLyn*.

nuestra máquina de desarrollo, no estamos utilizando ningún repositorio externo. La creación de tareas en repositorios externos la cubriremos en la sección 13.3.

Tras seleccionar el repositorio local, se nos abrirá la ventana mostrada en la Figura 13.5 donde podemos introducir las propiedades de la nueva tarea.

En esta ventana podemos definir las siguiente propiedades para la nueva tarea:

- Nombre de la tarea.
- *Status*: el estado de la tarea, que puede ser *Completo* o *Incompleto*.
- *Scheduled*: cuando está planificado que trabajaremos en la tarea.
- *Due*: en que fecha debe estar completada la tarea.
- *Estimate*: número de horas estimadas de dedicación a la tarea.
- Campo de comentarios.

Una vez definidas estas propiedades de la nueva tarea, podemos guardarla, y al hacerlo, veremos que se actualiza la vista *Task List* con la nueva tarea, que aparecerá en la carpeta *Uncategorized*. En esta carpeta se añaden, de modo automático, todas las tareas que no se han asignado a ninguna categoría.

Antes de ver cómo crear nuevas categorías veamos cual es el trabajo básico con las tareas. *MyLyn* nos permite concentrarnos sólo en el código que necesitamos para desarrollar una determinada tarea. Para activar una tarea pulsa el icono con aspecto de círculo, con relleno blanco la primera vez, que se encuentra a la izquierda del nombre de la tarea. Verás que ocurren varios cambios en *Eclipse*, por un lado, el aspecto del icono a la izquierda de la tarea cambia de aspecto para mostrar una esfera sombreada en gris. Y lo que es más importante, la vista *Package Explorer* ha ocultado toda la información relativa a la organización de los paquetes y las clases en ella contenida. Observa también que el botón de tarea activa de *MyLyn* mostrado en la Figura 13.6 está pulsado.

¿Qué ha ocurrido al activar la tarea? ¿Por qué han desaparecido todas las clases de la vista *Package Explorer*? Como hemos comentado, *MyLyn* nos permite concentrarnos sólo en el código relacionado con la tarea que activa en cada momento. ¿Cómo lo consigue? Ocultando el código que no hemos modificado o consultado en el desarrollo de la tarea. Como es la primera vez que activamos la tarea, y no hemos modificado o consultado ningún código, *MyLyn* oculta toda la jerarquía de paquetes. Para desactivar el filtro, pulsa sobre el icono mostrado en la Figura 13.6, se mostrará de nuevo la jerarquía completa de clases. Abre

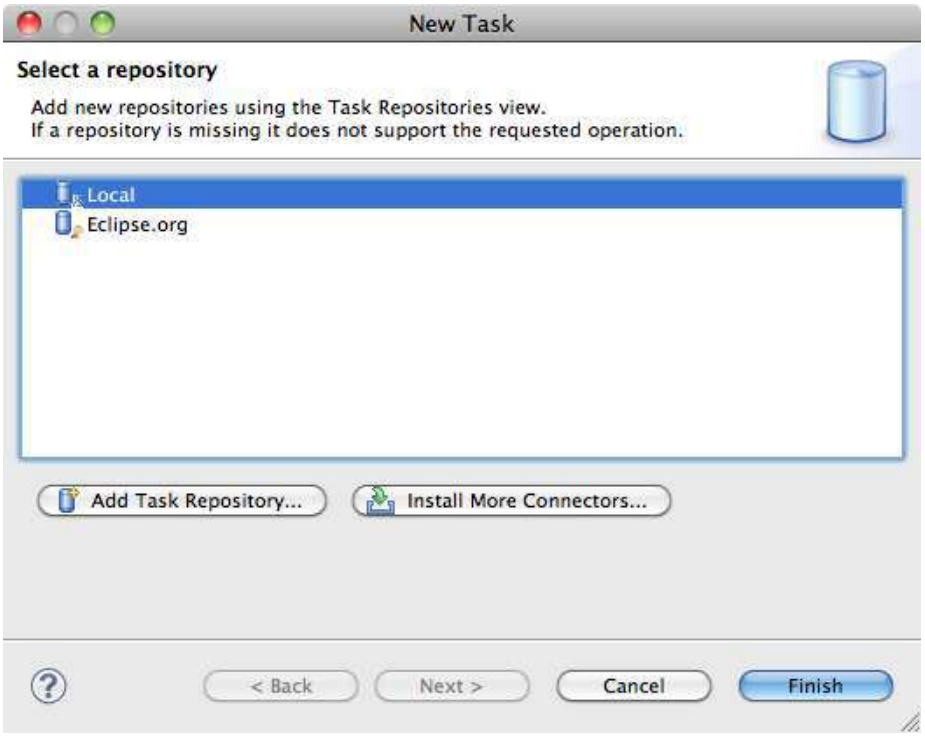

Figura 13.4: Selección del repositorio donde se añadirá la nueva tarea.

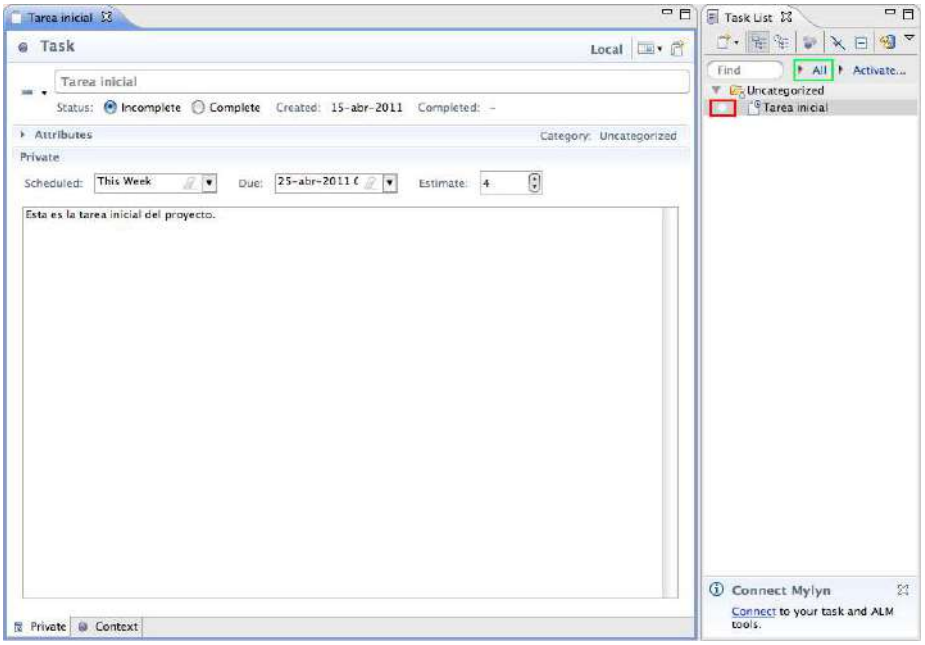

Figura 13.5: Definición de las propiedades de una nueva tarea en *MyLyn*.



Figura 13.6: Tarea activa en MyLyn *MyLyn*.

el código de una clase y activa de nuevo el filtro. Verás que todas las clases se ocultan excepto aquella que aparece en el editor. Realiza algún cambio en el código de la clase, como añadir algún comentario, verás que en la vista *Package Explorer* aparecen sólo los métodos en los que has añadido comentarios, el resto de métodos de la clase permanece oculto. Esta es la base de la potencia de *MyLyn* presentar sólo el código de las clases que hemos modificado o consultado al desarrollar una determinada tarea.

Observa que en el editor también aparece el icono de filtro de *MyLyn*, si lo activas, los métodos sobre los que no has trabajado al desarrollar la tarea se colapsarán, sólo se mostrará el código de aquellos métodos que has modificado o consultado.

Finalmente, si desactivas la tarea, pulsando en el icono a la izquierda de la tarea en la vista *Task List*, *Eclipse* te mostrará todos los proyectos, paquetes y clases en la vista *Package Explorer*. Si activas de nuevo la tarea, *Eclipse* te mostrará sólo el código relacionado con ella. *MyLyn* recuerda el estado en que quedó la información de trabajo relacionada con la tarea en el momento de su desactivación.

13.1.2.2. Trabajar con categorías

Las categorías son agrupaciones de tareas. Una tarea sólo puede pertenecer a una categoría. Las categorías nos sirven para estructurar las tareas. Para un proyecto concreto podemos, por ejemplo, crear una categoría relativa a las tareas relacionadas con la creación del interface gráfico de usuario, otra categoría relativa a las tareas relacionadas con el modelo de datos de nuestra aplicación y así sucesivamente.

Para crear una nueva categoría despliega la opción de creación de nuevos elementos en la vista *Task List* y selecciona la opción *New Cathegory* tal y como muestra la Figura 13.7.

Como nombre de la categoría introduzcamos, por ejemplo, *Modelo de datos*. Al pulsar el botón *Ok* observarás que se ha creado una carpeta con ese nombre en la vista *Task List*. Para añadir, o cambiar de ubicación, una tarea ya existente, simplemente pincha sobre la tarea y arrástrala hasta la categoría deseada. Verás que la descripción de la tarea se actualiza para dar cuenta de la categoría a la que pertenece.

Para crear una tarea dentro de una determinada categoría, simplemente selecciona la categoría antes de crear la tarea. Por defecto, la tarea se creará en la categoría actualmente seleccionada.

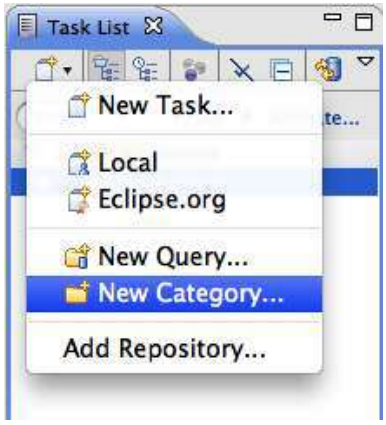

Figura 13.7: Creación de una nueva categoría en *MyLyn*.

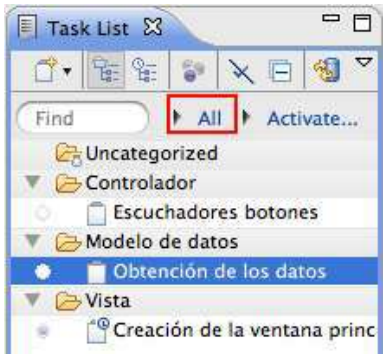

Figura 13.8: Creación de un *Workin Set* en *MyLyn*.

13.1.2.3. Trabajar con *Working Sets*

Por encima de las categorías existe un nuevo escalón que nos permite organizarlas en nuevos grupos. Fíjate que las tareas sólo pueden pertenecer a una única categoría, una tarea no puede estar en más de una categoría. Los *Working Sets* nos permiten agrupar categorías, y una categoría puede pertenecer a más de un *Working Set*. La pertenencia de una categoría a un *Working Set* no es exclusiva.

Supongamos que tenemos una jerarquía de tareas y categorías como la mostrada en la Figura 13.8.

Pulsa el triángulo negro a la izquierda del enlace *All* para desplegar las opciones y selecciona *Edit*. Se abrirá la ventana de la Figura 13.9.

Pulsa el botón *New* para crear un nuevo *Working Set*, se abrirá la ventana mostrada en la Figura 13.10. Introduce un nombre para este nuevo *Working Set* y selecciona las categorías *Vista* y *Modelo de datos*. Para acabar pulsa el botón *Finish*. Volverás a la ventana de la Figura 13.8 pero esta vez podrá ver el nuevo *Working Set* recién creado, pulsa el botón *Ok*. Ahora, si despliegas de nuevo la lista de *Working Sets* (triángulo a la izquierda del enlace *All* de la Figura 13.8 verás el nombre del nuevo *Working Set*, si lo seleccionas sólo te aparecerán las categorías *Modelo de datos* y *Vista* que son las que has incluido en este *Working*

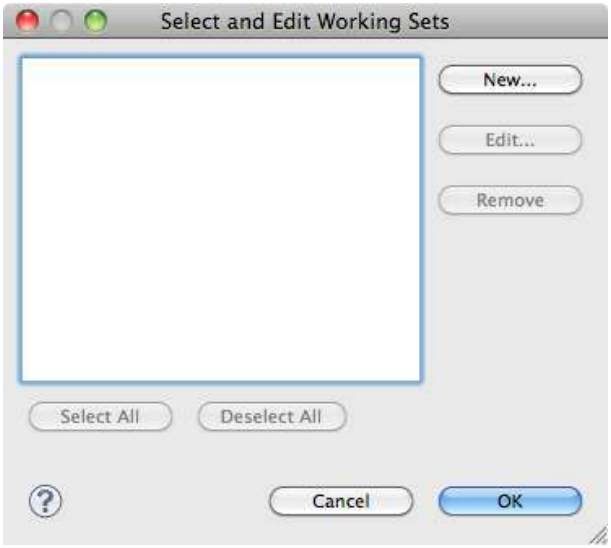

Figura 13.9: Selección y edición de un *Workin Set* en *MyLyn*.

Set, el resto de categorías no se visualiza.

Puedes probar a crear un nuevo *Working Set*, llámalo *Relación entre Modelo y Controlador* y añade a él sólo las categorías *Modelo* y *Controlador*. Como ves, la categoría *Modelo* está dentro de dos *Working Sets*.

13.2. Sistema de gestión de errores *Bugzilla*

Bugzilla es una herramienta libre y de código abierto utilizada por grandes compañías, como *Mozilla*, *Apache* o *Eclipse*. Se puede trabajar con *Bugzilla* desde un navegador web o bien desde un cliente como *MyLyn* si se dispone del conector adecuado. Afortunadamente, y no es de extrañar, *Eclipse* proporciona por defecto este conector.

13.2.1. Cual es el objetivo de *Bugzilla*

Bugzilla es un sistema de seguimiento de errores. Durante el ciclo de vida del software, una de las etapas básicas es la detección y solución de errores. *Bugzilla* nos permite gestionar y automatizar el seguimiento de errores hasta su resolución final.

13.2.2. Instalación de *Bugzilla*

En el momento de escribir estas página, la última versión estable de *Bugzilla* es la 4.0, que se puede descargar desde `http://www.bugzilla.org`.

En esta sección se muestra cómo instalar *Bugzilla* en Ubuntu 10.10. Para otros sistemas operativos o versiones de Linux consultar los detalles en la página web de *Bugzilla*.

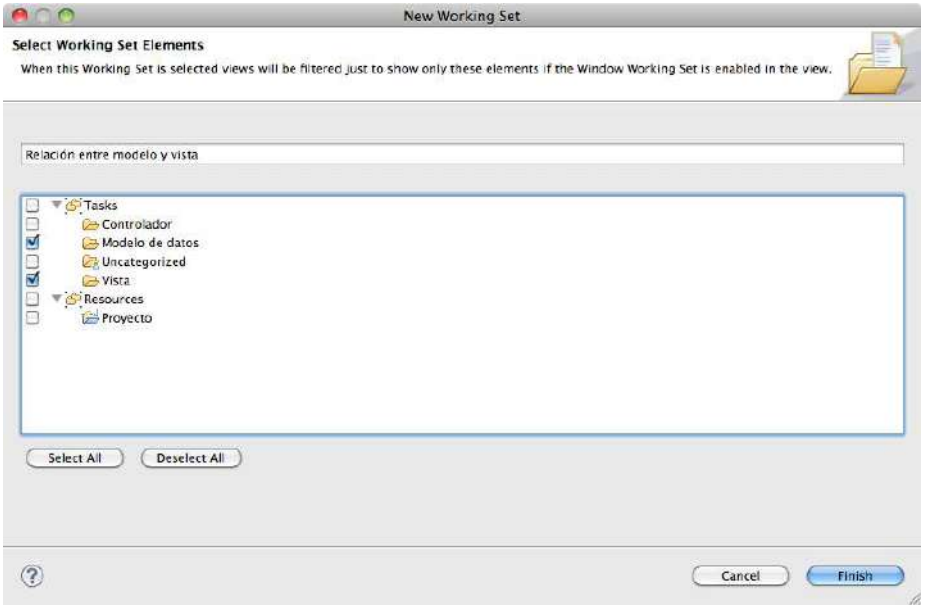

Figura 13.10: Propiedades de un *Workin Set* en *MyLyn*.

Antes de instalar *Bugzilla* necesitamos instalar algunos paquetes de los que depende *Bugzilla*. El primero de ellos es **perl**, para comprobar si tenemos `perl` instalado abrimos un terminal y escribimos:

```
$perl -v

This is perl, v5.10.1 (*) built for i486-linux-gnu-thread-multi

Copyright 1987-2009, Larry Wall

Perl may be copied only under the terms of either the Artistic
License or the GNU General Public License, which may be found
in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be
found on this system using "man perl" or "perldoc perl".  If you
have access to the Internet, point your browser at
http://www.perl.org/, the Perl Home Page.
```

Si podemos ver lo anterior es que tenemos la versión 5.10.1 de `perl` instalada, *Bugzilla* necesita al menos la versión 5.8. Si `perl` no está instalado lo podemos instalar tecleando en el terminal:

```
$sudo apt-get install perl
```

Fíjate que para instalar nuevos paquetes en linux necesitamos permiso de superusuario, de ahí que utilicemos `sudo`. Antes de proceder a la instalación se nos preguntará por la clave de superusuario.

Lo siguiente que debemos instalar, si no lo tenemos ya instalado, es un gestor de bases de datos. Vamos a utilizar *MySQL* aunque también es posible utilizar *PostgreSQL*. Para comprobar si *MySQL* está instalado escribimos de nuevo en un terminal:

```
$mysql --version
mysql  Ver 14.14 Distrib 5.1.41, for debian-linux-gnu (i486)
using readline 6.1
```

Lo anterior nos indica que tenemos instalada la versión 5.1.41 de este gestor de bases de datos. Si no lo tenemos, lo podemos instalar tecleando:

```
apt-get install mysql-admin mysql-client mysql-server
```

Lo anterior nos instalará tanto el gesto de bases de datos como el cliente y las herramientas de administración. Durante la instalación de *MySQL* se nos pedirá que definamos la clave de acceso del administrador. Esta clave de acceso la utilizaremos más tarde en la configuración de *Bugzilla*.

El siguiente paso es comprobar si tenemos instalado el servidor web *Apache*. Para ello, de nuevo, escribe en un terminal:

```
$apache2 -v
Server version: Apache/2.2.16 (Ubuntu)
Server built:   Nov 18 2010 21:17:43
```

En este caso, el mensaje nos indica que tenemos instalado el servidor web *Apache* en nuestro sistema en su versión 2.2.16. Si no fuese el caso lo puedes instalar tecleando en el terminal

```
$sudo apt-get install apache2
```

En este punto, ya tenemos instalados todos los paquetes necesarios para poder trabajar con *Bugzilla*. Lo siguiente es descargar y configurar *Bugzilla*.

El directorio por defecto donde el servidor web *Apache* busca los ficheros solicitados es, en el caso de la distribución de Linux Ubuntu 10.10, `/var/www`[1]. Sitúate en ese directorio y descarga *Bugzilla* escribiendo en un terminal lo siguiente:

```
$sudo wget http://ftp.mozilla.org/pub/mozilla.org/
webtools/bugzilla-4.0.tar.gz
```

Al acabar la descarga verás que tienes un fichero comprimido con nombre bugzilla-4.0.tar.gz. Para descomprimir este fichero escribe en el terminal:

```
$ tar -xvf bugzilla-4.0.tar.gz
```

Observarás que se ha creado el directorio bugzilla-4.0. El siguiente paso para tener *Bugzilla* a punto es instalar los módulos *perl* que necesita *Bugzilla*, para ello sitúate en el directorio `/var/www/bugzilla-4.0` y escribe en el terminal:

```
$sudo ./checksetup.pl ?check-modules
```

[1]Para otras distribuciones de Linux y sistemas operativos consultar la documentación.

Te aparecerá una lista con los módulos necesarios y los opcionales que necesita *Bugzilla*. Tal y como se indica al final de la lista de módulos, puedes instalar todos los necesarios escribiendo en el terminal:

```
$sudo /usr/bin/perl install-module.pl ?all
```

Para comprobar que tenemos todos los módulos necesarios instalados escribe de nuevo en el terminal:

```
$sudo ./checksetup.pl ?check-modules
```

Si alguno de los módulos necesarios no se ha instalado escribe de nuevo en un terminal:

```
$sudo /usr/bin/perl install-module.pl ?all
```

En este punto necesitamos instalar los módulos *perl* necesarios para *Apache*. Los instalamos escribiendo en el terminal:

```
$sudo apt-get install libapache2-mod-perl2
libapache2-mod-perl2-dev
libapache2-mod-perl2-doc
$ sudo /usr/bin/perl install-module.pl Apache2::SizeLimit
```

Con todo esto ya podemos continuar con la instalación de *Bugzilla*, escribe en el terminal:

```
$sudo ./checksetup.pl
```

Con ello, entre otras cosas, se habrá creado el fichero `localconfig` en el directorio `/var/www/bugzilla-4.0`. Edita este fichero y modifica las siguiente variables de *Bugzilla*:

```
$webservergroup = ?www-data?;
$db_pass = ?clave de administrado de MySQL?;
```

El siguiente paso es crear la base de datos que manejará *Bugzilla*. Para crear esta base de datos utilizaremos el cliente de *MySQL* que ya hemos instalado. Escribe en un terminal:

```
$ sudo mysql -u root -p
```

Se te solicitará la clave de administrador que has introducido en la instalación de *MySQL*. Una vez introducida con éxito nos encontraremos en el ambiente del cliente de *MySQL*. Para crear la base de datos que manejará *Bugzilla* escribe:

```
mysql> create database bugs;
Query OK, 1 row affected (0.00 sec)
```

Para ver todas las bases de datos que gestiona *MySQL* escribe:

```
mysql> show databases;
+-----------+
| Database |
+-----------+
| information_schema |
| bugs |
| mysql |
+-----------+
3 rows in set (0.00 sec)
```

Verás que efectivamente tenemos la base de datos llamada `bugs` creada. Lo siguiente es asignar todos los privilegios para trabajar con esta base de datos al administrador `root`, para ello escribe:

```
mysql> GRANT ALL PRIVILEGES ON bugs.* TO root@localhost
IDENTIFIED BY ?clave de administrador?;
mysql> FLUSH PRIVILEGES;
```

Para salir del cliente de *MySQL* escribe:

```
mysql> quit;
```

De regreso al terminal vamos a continuar con la instalación de *Bugzilla*, escribe lo siguiente en el terminal:

```
$ ./checksetup.pl
```

Este script de *perl* creará, entre otras cosas, todas las tablas necesarias en la base de datos `bugs` y configurará *Bugzilla*. También se nos pedirá que introduzcamos el correo electrónico y clave de acceso del Administrador de *Bugzilla*. El nombre de usuario en *Bugzilla* debe ser un correo electrónico.

El siguiente paso es configurar *Apache* para que reconozca el directorio donde tenemos instalado *Bugzilla*, para ello sitúate en el directorio `/etc/apache2`[2] y edita el fichero `httpd.conf` y escribe en él lo siguiente:

```
Alias /bugzilla "/var/www/bugzilla-4.0"
<Directory /var/www/bugzilla-4.0>
    AddHandler cgi-script .cgi
    Options +Indexes +ExecCGI
    DirectoryIndex index.cgi
    AllowOverride Limit
</Directory>
```

En este punto sólo nos queda reiniciar el servidor web *Apache* escribiendo lo siguiente en el terminal:

```
$sudo /etc/init.d/apache2 restart
```

Y si no tenemos ningún problema ya podemos abrir un navegador web para conectarnos a *Bugzilla* escribiendo como url la siguiente `http://localhost/`



Figura 13.11: Página de inicio de *Bugzilla*.



Figura 13.12: Página de introducción de usuario y clave de *Bugzilla*.

`bugzilla`. Deberemos ver la página de inicio de *Bugzilla* tal y como se muestra en la siguiente Figura 13.11:

Una vez ingresado en *Bugzilla* utilizando el correo electrónico del administrador como usuario y la clave que hemos definido en la instalación de de *Bugzilla*, se presentará una pantalla como la mostrada en la Figura 13.12, seleccionamos el enlace *Administration* para acabar de configurar *Bugzilla*.

En la nueva pantalla mostrada en la Figura 13.13 seleccionamos el enlace *Parameters* lo que nos dará paso a la pantalla mostrada en la Figura 13.14

En esta pantalla, los únicos parámetros que es necesario configurar son **urlbase** y **cookiepath**. Como valor de **urlbas** escribiremos la url de nuestro servidor, en el ejemplo se muestra `http://localhost/bugzilla/` lo que indica que estamos utilizando *Bugzilla* en el servidor local, no tendremos acceso a este servidor desde otra máquina. Si *Bugzilla* estuviese instalado en

[2]El directorio donde se encuentra el fichero `httpd.conf` puede ser distinto dependiendo de la distribución de Linux o del sistema operativo.

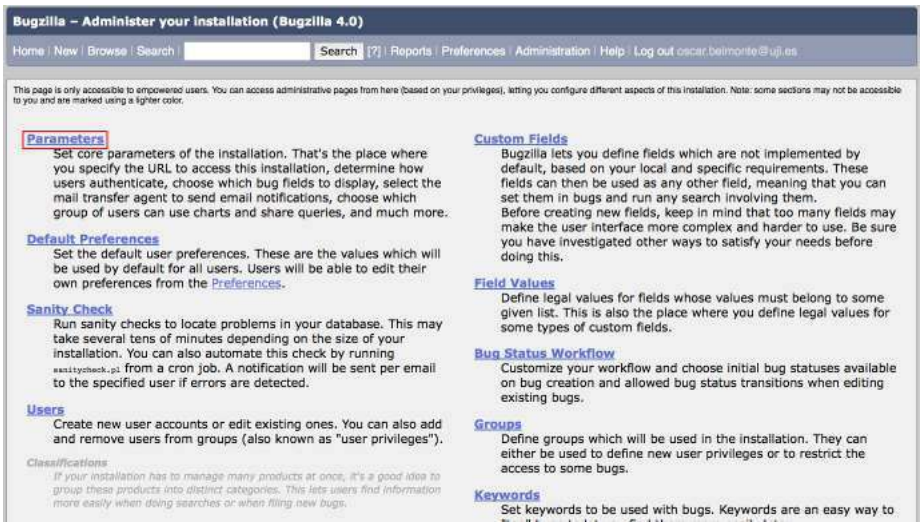

Figura 13.13: Página de administración de *Bugzilla*.

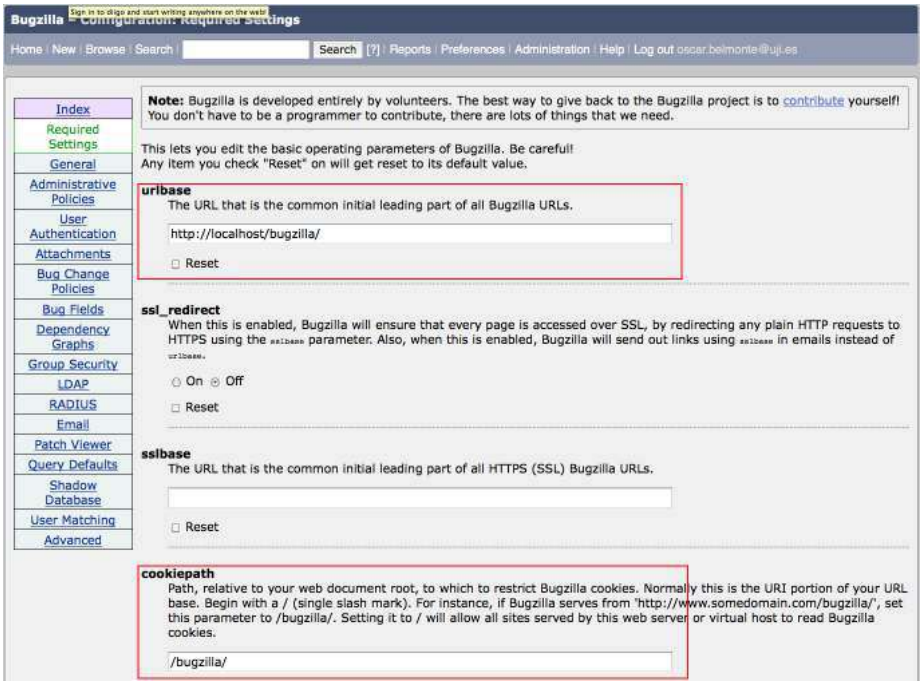

Figura 13.14: Página de definición de parámetros de *Bugzilla*.



Figura 13.15: Lista de usuarios en *Bugzilla*.

un servidor con nombre mi.servidor.com el valor del parámetro **urlbase** sería `http://mi.servidor.com/bugzilla/`. Como valor de **cookiepath** escribiremos `/bugzilla/`. Finalmente pulsamos el botón *Sava changes*.

13.2.3. Trabajar con *Bugzilla*

Como ya se hemos comentado, *Bugzilla* es un sistema de gestión de errores con interfaz web. El núcleo de trabajo con *Bugzilla* es el proyecto. En *Bugzilla* se pueden crear tantos proyectos como sea necesario. Una vez creado un proyecto, todos los usuarios que estén autorizados a dar de alta los errores encontrados en el proyecto pueden empezara a hacerlo.

Por lo tanto, lo primero que vamos a hacer es crear un nuevo proyecto. Después crearemos un usuario y le asignaremos privilegios para dar de alta los errores encontrados en el proyecto recién creado.

Todo este proceso lo vamos a hacer con la cuenta de Administrador de *Bugzilla*. Ingresa en *Bugzilla* pulsando sobre el enlace *Log In* que se muestra en la Figura 13.11, se te solicitará un nombre de usuario y su clave de acceso. Como ya hemos comentado, los nombre de usuario en *Bugzilla* deben ser direcciones de correo electrónico, debemos introducir la dirección de correo electrónico de Administrador que introdujimos en la configuración de *Bugzilla* y su clave.

Como en el caso de la configuración de los parámetros de *Bugzilla*, seleccionamos en enlace *Administration* y esta vez, dentro de esta pantalla, seleccionamos el enlace *Users*, lo que nos dará paso a la pantalla mostrada en la Figura 13.15, donde seleccionamos el enlace *add a new user*, se nos mostrará la pantalla de la Figura 13.16 donde introduciremos los datos del nuevo usuario. Fíjate que como nombre de usuario se debe introducir una dirección de correo electrónico. Finalmente pulsamos el botón *Add*.

Lo siguiente que vamos a hacer es crear un nuevo producto. Como ya se ha comentado, los errores se dan de alta en un determinado producto, puedes pensar que los productos son tus proyectos. Para crear un nuevo producto, vuelve a la página *Administration* y selecciona el enlace *Products*, verás una página como la de la Figura 13.17, por defecto hay un producto de prueba creado, llamado *TestProduct*. Vamos a añadir un nuevo producto selecciona el enlace *Add* lo que te llevará a la nueva página mostrada en la Figura 13.18, introduce los parámetros del nuevo producto y pulsa el botón *Add*, esto te llevará a la nueva página mostrada en la Figura 13.19, en esta página se nos indica que debemos crear al menos un componente dentro del nuevo producto, puedes pensar que los componentes de los productos son como las tareas de tu proyecto. Selecciona

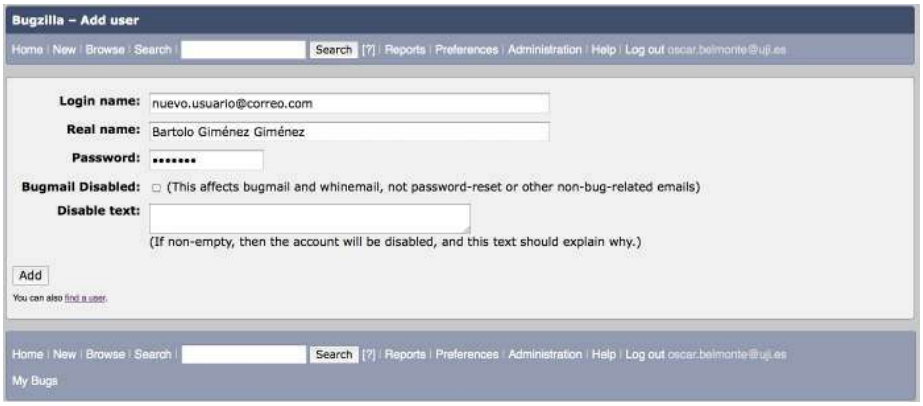

Figura 13.16: Página de creación de un nuevo usuario en *Bugzilla*.



Figura 13.17: Lista de productos de *Bugzilla*.

el enlace *Edit components*, lo que te llevará a la página mostrada en la Figura 13.20. Selecciona el enlace *Add* con lo que te llevará a la página de definición del nuevo componente mostrada en la Figura 13.21, introduce en ella los datos del nuevo componente y pulsa el botón *Add* lo que te dará entrada de nuevo a la página donde se muestra los componentes de un producto. Ahora verás que el componente *ComponentePrueba* se ha añadido a la lista.

Resumiendo lo que he hemos hecho hasta este momento:

1. Hemos configurado *Bugzilla*.
2. Hemos creado un nuevo usuario.



Figura 13.18: Propiedades de un producto en *Bugzilla*.

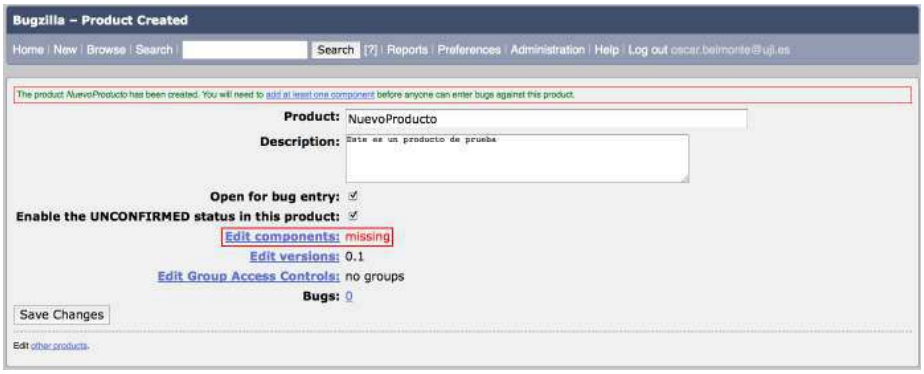

Figura 13.19: Producto recién creado en *Bugzilla*. Debemos crear al menos un componente en este producto.



Figura 13.20: Lista de los componentes de un producto en *Bugzilla*.

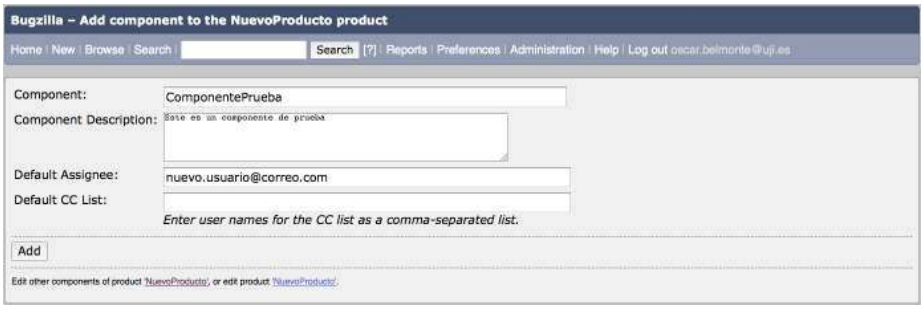

Figura 13.21: Propiedades de un nuevo componente en *Bugzilla*.



Figura 13.22: Lista de los productos en *Bugzilla* en los que podemos dar de alta nuevos errores.

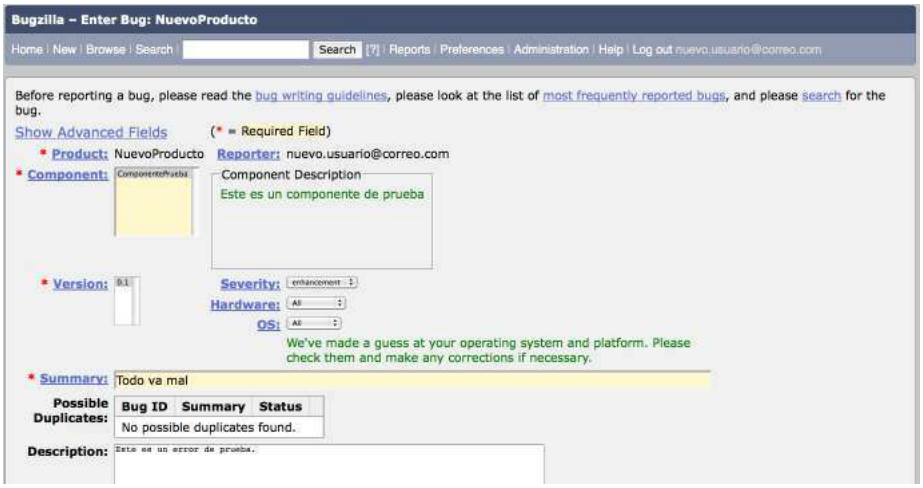

Figura 13.23: Definición de los parámetros del error que se está dando de alta en *Bugzilla*.

3. Hemos creado un nuevo producto (proyecto).

4. Hemos añadido un nuevo componente (tarea) al producto (proyecto).

Ahora ya podemos ingresar a *Bugzilla* con nuestro nuevo usuario y dar de alta un error en el componente recién creado. Para ello sal de *Bugzilla* ya que hemos ingresado como administradores, e ingresa de nuevo pero esta vez con el usuario y clave recién creados. En la página de inicio de *Bugzilla* selecciona el icono con leyenda *File a Bug*, lo que te llevará a la nueva página mostrada en la Figura 13.22 donde seleccionarás el producto donde quieres dar de alta el error. En nuestro caso, seleccionamos el enlace *NuevoProducto*, lo que nos llevará a la página mostrada en la Figura 13.23 y pulsa el botón *Submit Bug*, lo que te dará paso a la página mostrada en la Figura 13.24 donde se muestran todos los detalles del error recién creado. Esta última contiene todos los detalles del error, y a medida que trabajemos sobre este error y añadamos nueva información, toda ella aparecerá en esta pantalla.

Para completar esta breve introducción al uso de *Bugzilla* realicemos una búsqueda sobre la base de datos de errores. Para ello vuelve a la página inicial de *Bugzilla* y selecciona el icono que tiene como leyenda *Search*, y en la nueva página seleccionemos el enlace *Advanced Search*, en este momento se nos mostrará la página de la Figura 13.25. En esta página seleccionemos el producto

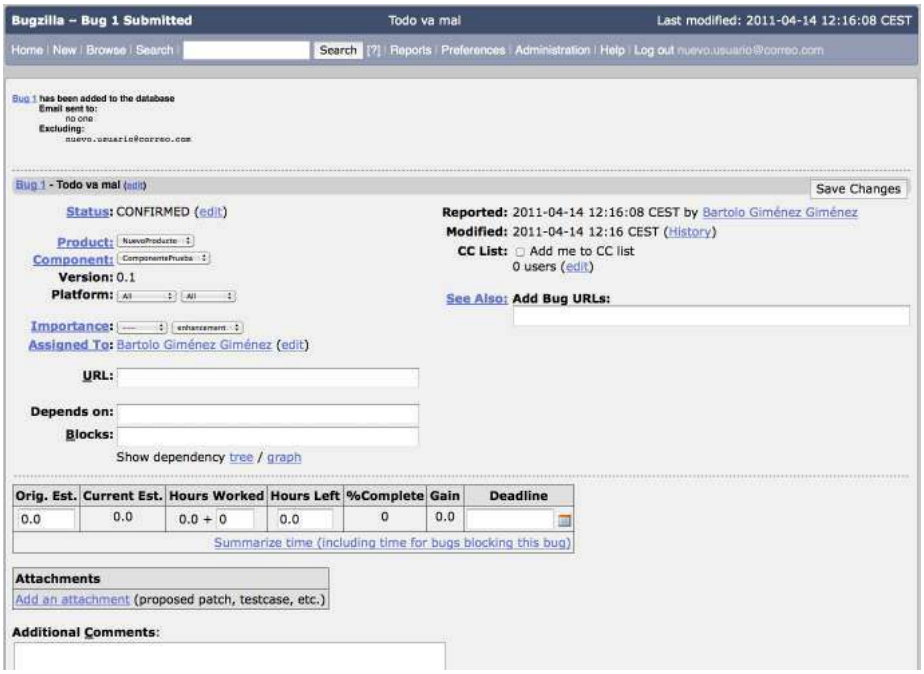

Figura 13.24: Información sobre el error recién creado en *Bugzilla*.

NuevoProducto y dejemos el resto de campos tal y como están, esta consulta se interpreta como «Busca todos los errores dados de alta en el producto Nuevo-Producto», pulsa el botón *Search* y verá la página mostrada en la Figura 13.26 con los resultados de la búsqueda. Sólo aparece un único error y al pulsar sobre el identificador de este error volveremos a la página de información detallada sobre él.

Hasta aquí hemos visto el trabajo básico con *Bugzilla* como una potentísima herramienta de control y gestión de errores, pero sin duda, la potencia de esta herramienta se magnifica en combinación con *MyLyn*, tal y como se muestra en la siguiente sección.

13.3. Acceso a *Bugzilla* desde *MyLyn* y *Eclipse*

Tanto *MyLyn* como *Bugzilla* son herramientas muy útiles en el desarrollo de software utilizadas de modo independiente. Pero sin duda, la posibilidad de combinar ambas soluciones en *Eclipse* nos facilita enormemente el desarrollo de software y el seguimiento de errores.

Hemos visto cómo *MyLyn* nos permite definir tareas y concentrar nuestra atención únicamente en el código relacionado con ellas. Por otro lado, *Bugzilla* es un potente sistema de seguimiento de errores. Como desarrolladores de software, algunas de nuestras tareas consisten en solucionar errores en las aplicaciones que desarrollamos, podemos ver la resolución de cada error como una tarea específica. Sería muy interesante poder combinar la potencia de *MyLyn* con la de *Bugzilla* para trabajar en la resolución de errores como si se tratase de otra

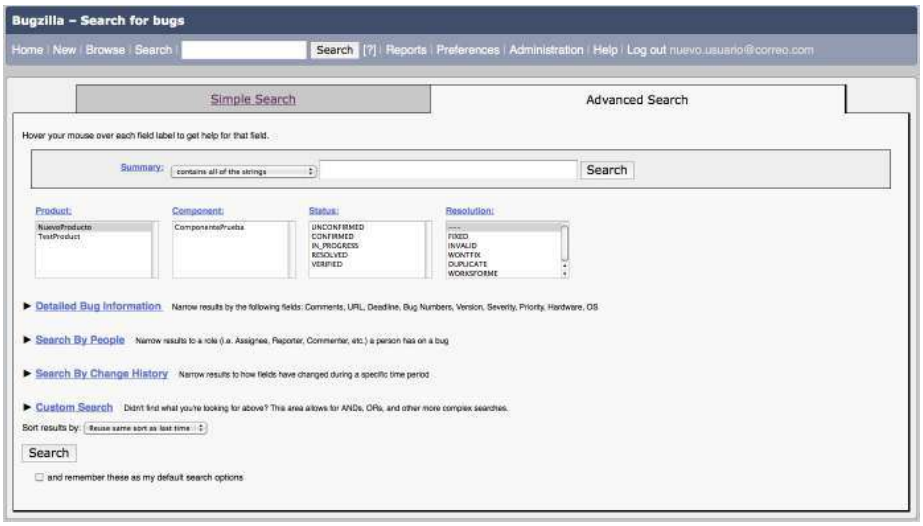

Figura 13.25: Búsqueda de errores en *Bugzilla*.



Figura 13.26: Resultado de la búsqueda de errores en *Bugzilla*.

tarea cualquiera.

13.3.1. Beneficios de la combinación de *Bugzilla* y *MyLyn* desde *Eclipse*

La principal ventaja de poder trabajar en *Eclipse* con *Bugzilla* a través de *MyLyn* es que podemos realizar el seguimiento de los errores como si de otra tarea cualquiera se tratase.

En *MyLyn* podemos definir una consulta ha *Bugzilla* de tal modo que al dar de alta un nuevo error este nos aparezca como una nueva tarea en *MyLyn*.

Otra ventaja importante es que podemos trabajar con *Bugzilla* directamente desde *Eclipse* sin necesidad de abandonar el entorno de desarrollo para realizar el seguimiento de los errores.

13.3.2. Trabajo con *MyLyn* y *Bugzilla* desde *Eclipse*

En esta sección vamos a presentar los fundamentos del trabajo conjunto desde *Eclipse* con *MyLyn* y *Bugzilla*.

13.3.2.1. Añadir errores a *Bugzilla* desde *Eclipse*

Para crear un nuevo error en *Bugzilla* desde *Eclipse* crea una nueva tarea en la vista *Task List* tal y como se muestra en la Figura 13.3. Cuando se abra la ventana mostrada en la Figura 13.4 pulsa esta vez sobre el botón *Add Task Repository*. Lo que vamos a hacer esta vez, es crear un repositorio hacia *Bugzilla*. Al pulsar sobre el botón *Add Task Repository* se abrirá la ventana mostrada en la Figura 13.27, selecciona *Bugzilla* y pulsa el botón *Next*[3].

En la nueva ventana que se abrirá, introduce los datos de conexión al repositorio de *Bugzilla* como se muestra en la Figura 13.28.

Una vez rellenados los datos de conexión es útil pulsar el botón *Validate Settings* para comprobar que se puede establecer la conexión con el repositorio. Si la conexión se establece podemos pulsar el botón *Finish*, de lo contrario, corrige los datos de conexión y vuelve a pulsar el botón *Finish*, verás la ventana de la Figura 13.29 donde aparecerá el repositorio recién creado **Errores**.

Pulsa de nuevo el botón *Finish* con lo que se abrirá una solapa en *Eclipse* para definir las propiedades del nuevo error. Rellénala con los datos de un error tal y como muestra la Figura 13.30.

Finalmente pulsa el botón *Submit* con lo que se enviará el nuevo error al repositorio *Bugzilla*. Como ejercicio puedes probar a buscar el nuevo error desde un navegador web en *Bugzilla*.

13.3.2.2. Recuperar errores desde *Bugzilla* como tareas en *MyLyn*

Por último, veamos cómo podemos crear una consulta desde *MyLyn* para que nos aparezcan como tareas en la vista *Task List* los errores presentes en el repositorio de *Bugzilla*. Para ello seleccionemos el botón *New Query* que se muestra en la Figura 13.3 y en la ventana que se abrirá seleccionemos el repositorio creado en la sección anterior **Errores**, y pulsemos el botón *Next*.

[3]Es posible que tengas que actualizar *MyLyn* para trabajar con la versión 4.0 de *Bugzilla*, consulta la dirección `http://www.eclipse.org/mylyn/new/` para conocer los detalles.



Figura 13.27: Creación de un nuevo repositorio en *MyLyn*.

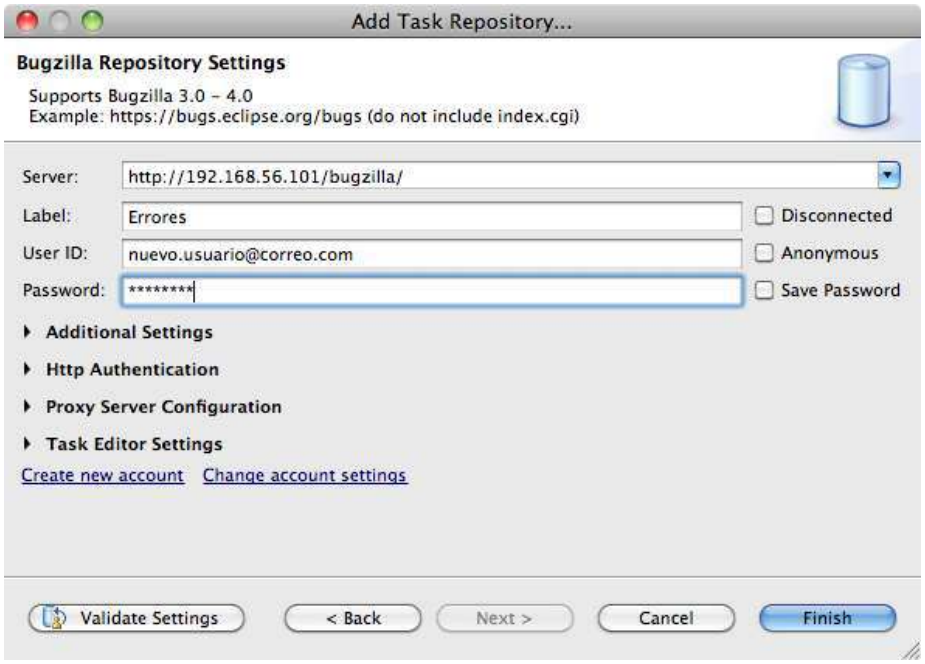

Figura 13.28: Datos del repositorio *Bugzilla*.

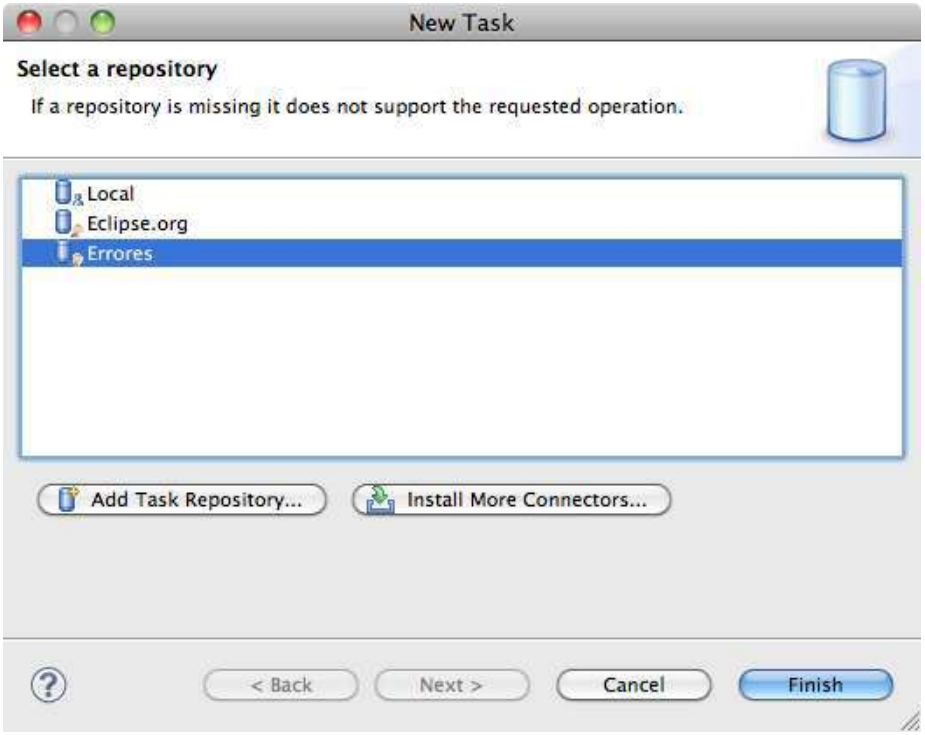

Figura 13.29: Repositorio *Errores* recién creado.

En la nueva ventana que se abrirá tenemos dos opciones:

- *Create query using form.*
- *Create query from existing URL.*

La opción que nos interesa es la primera, que aparece seleccionada por defecto, y pulsemos el botón *Next*.

En la nueva ventana que se abrirá introduce los parámetros de la consulta tal y como muestra la Figura 13.31, y finalmente pulsa el botón *Finish*, en la vista *Task List* podrás ver los errores importados a *MyLyn* desde *Bugzilla*, tal y como se muestra en la Figura 13.32.

Ahora podrás trabajar con ellos como con cualquier otra tarea local, además de disponer de todos los campos que ofrece el conector a *Bugzilla*.

Lecturas recomendadas.

- El capítulo 25 de la referencia [13] está dedicado enteramente a *MyLyn*.
- En la dirección `http://wiki.eclipse.org/index.php/Mylyn/User_Guide` se puede encontrar un manual en línea completo para el trabajo con *MyLyn*. En la dirección `http://www.eclipse.org/mylyn/start/` se pueden encontrar otros enlaces de gran interés como vídeo tutoriales sobre *MyLyn*, muy aconsejables de ver.

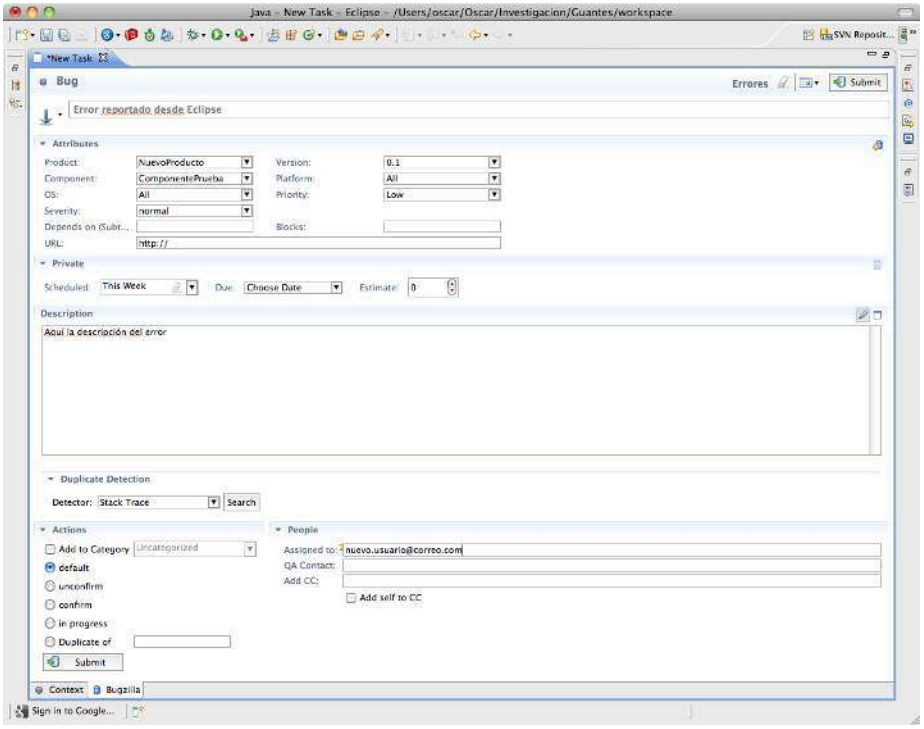

Figura 13.30: Datos del repositorio *Bugzilla*.

- El capítulo 27 de la referencia [13] está dedicado enteramente a *Bugzilla*
- La dirección web de *Bugzilla* es `http://www.bugzilla.org/`, allí podrás encontrar gran cantidad de información sobre esta herramienta.

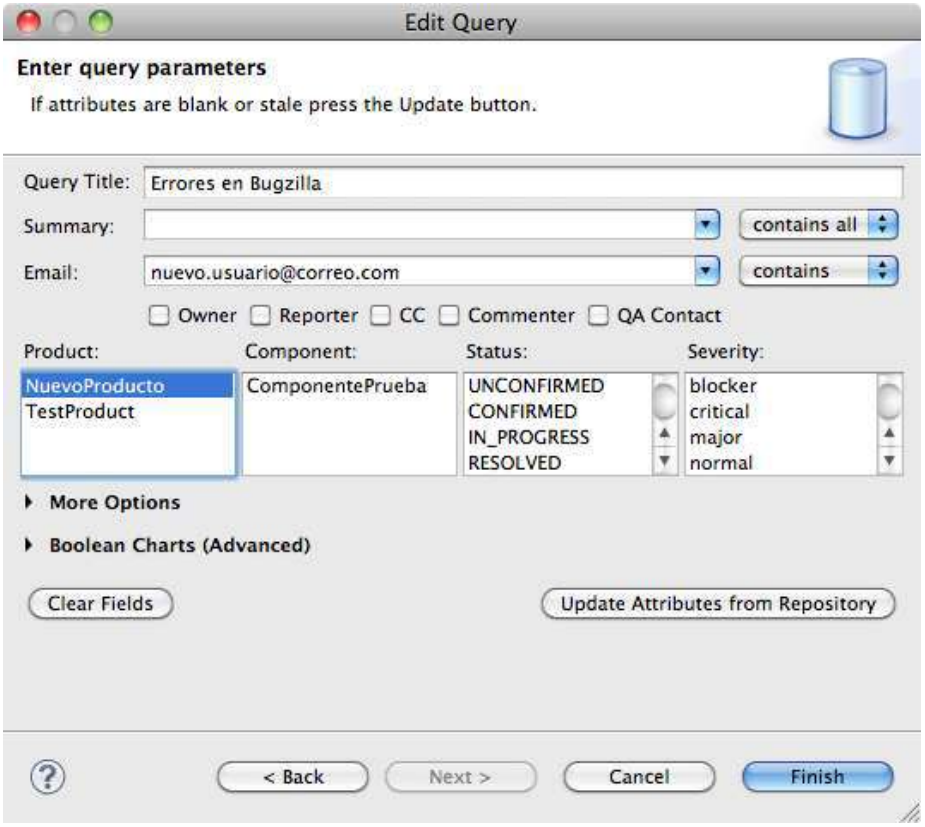

Figura 13.31: Datos del repositorio *Bugzilla*.

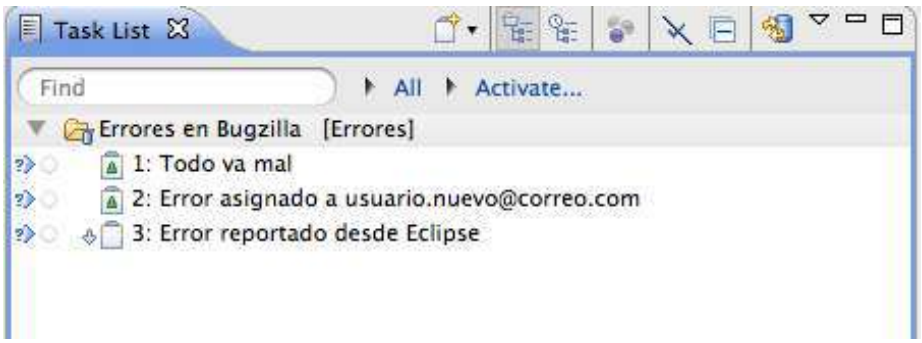

Figura 13.32: Datos del repositorio *Bugzilla*.

Capítulo 14

Programación concurrente con Hilos

Contenidos

14.1. ¿Qué es un hilo? Utilidades. Consideraciones sobre el uso de hilos 208
14.2. Creación de hilos en Java 209
14.2.1. Creación de un Hilo extendiendo a la clase `Thread` . 209
14.2.2. Creación de un Hilo mediante una clase interna . . . 210
14.2.3. Creación de un Hilo mediante una clase interna anónima 211
14.3. Ciclo de vida de un hilo 212
14.4. Control de hilos 213
14.5. Sincronización 215
14.5.1. Sincronizacón utilizando los cerrojos intrínsecos . . . 215
14.5.2. Sincronización utilizando el `interface Lock` 218

Introducción

La programación concurrente es nativa en Java, esto quiere decir que no necesitamos ninguna biblioteca adicional para poder trabajar con Hilos, los Hilos son nativos en Java.

Un buen entendimiento de cómo trabajar con Hilos es fundamental cuando en nuestras aplicaciones se ejecutan más de una tarea de manera concurrente. Y precisamente las aplicaciones con interfaces gráficos son un excelente ejemplo de las ventajas de trabajar con Hilos. Imagina que has escrito una aplicación, con un interfaz gráfico en la que al pulsar un botón se lleva a cabo un complejo cálculo que consume mucho tiempo de CPU. Si tu aplicación tiene un único Hilo de ejecución, el interfaz gráfico se quedará *congelado* cuando el usuario inicie el cálculo, ya que hasta que este no se acabe, no volverá el control al interfaz gráfico. Obviamente, ninguna aplicación se comporta así, o al menos no debería.

Todos los Hilos de una aplicación en Java comparten el mismo espacio de

memoria, lo que implica que varios Hilos pueden estar accediendo a la misma zona de memoria (por ejemplo una variable) para modificar su valor. En este caso se pueden producir colisiones e inconsistencias por el uso compartido de memoria. De algún modo hay que establecer un mecanismo de sincronización para que el acceso a zonas de memoria compartida no den lugar a este tipo de comportamiento erróneo.

En este capítulo vas a conocer cómo trabajar con Hilos en Java, vas a conocer cómo se crean Hilos en Java y como sincronizar su ejecución cuando acceden a recursos compartidos para que no haya inconsistencias en tus aplicaciones.

14.1. ¿Qué es un hilo? Utilidades. Consideraciones sobre el uso de hilos

De un modo muy resumido podemos decir que un Hilo es un flujo de control independiente dentro de un programa. Como tal tiene su propia pila de llamadas, pero todos los Hilos creados en el mismo programa comparten el mismo espacio de direcciones, lo que implica que comparten todas las instancias del programa excepto las locales. Nuestros programas pueden tener un gran número de Hilos ejecutándose al mismo tiempo y todos ellos comparten el uso de la CPU.

Esto proporciona grandes ventajas, ya que si disponemos, como empieza a ser la habitual, de CPUs con más de un núcleo, varios hilos pueden estar ejecutándose de modo paralelo en cada uno de los núcleos de la CPU.

También podemos pensar que cada uno de los Hilos es un programa secuencial en sí mismo, con lo que la programación de soluciones complejas, cuando se piensan como un todo, se simplifican enormemente al ser divididas en partes que se ejecutan de modo independiente pero coordinado.

A cambio tenemos que pagar un precio, ya que si varios Hilos intentan acceder al mismo recurso compartido y al mismo tiempo, el resultado, si el acceso no está sincronizado, puede no ser el deseado. Para ver lo que queremos decir con claridad, supongamos dos hilos que se están ejecutando sobre una única CPU, el código que ejecuta el primero de ellos aparece en el Listado 14.1 y el que ejecuta el segundo aparece en el Listado 14.2. Supongamos que el primer Hilo tiene el disfrute de la única CPU, y que el valor de la variable `cantidad` en ese momento es 9, la condición de la línea 1 se cumple, y antes de que se ejecute la segunda línea, se cede el disfrute de la CPU al segundo hilo, que comprueba que la condición de su línea 1 se cumple, por lo que incrementa en dos unidades la variable `cantidad` que pasa a valer 11, y después de modificar el valor de `cantidad`, de nuevo, se cede el disfrute de la CPU al primer hilo. En este momento, la condición que comprobó el primer Hilo ya no es válida y aún así, se incrementará el valor de la variable `suma` que de 11 pasará a valer 12. Evidentemente se ha producido un error ya que en este caso el acceso al recurso compartido `suma` no está sincronizado.

```
1 if(cantidad < 10)
2  cantidad++;
3 System.out.println("Cantidad: " + cantidad);
```

Listado 14.1: Código que ejecuta el primer Hilo.

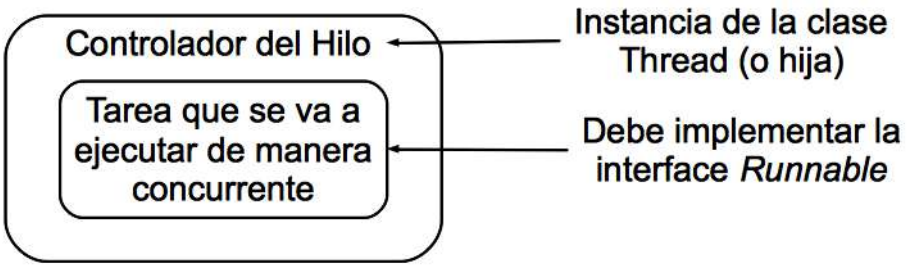

Figura 14.1: La idea de cómo crear un Hilo en Java gráficamente. Necesitamos dos ingredientes, la tarea cuyo código se va a ejecutar concurrentemente, y el controlador del hilo (clase `Thread`).

```
1 if(cantidad < 20)
2  cantidad += 2;
3 System.out.println("Cantidad: " + cantidad);
```

Listado 14.2: Código que ejecuta el segundo Hilo.

14.2. Creación de hilos en Java

Lo primero que hay que conocer de la creación de Hilos en Java es que se componen de dos partes, por un lado definiremos la tarea que queremos que se ejecute de manera concurrente con otras tareas, y por otro lado tenemos el controlador de esa tarea, que nos permitirá sincronizarla con el resto de tareas.

La clase que representa una tarea, y que controlará un Hilo, ha de implementar el `interface Runnable`. Este `interface` declara un único método `public void run()`, el código del cual se ejecutará de manera concurrente con otros Hilos cuando se inicie. Por otro lado, el controlador del Hilo será una instancia de la clase `Thread`, a quien en el momento de su creación debemos pasarle como argumento una instancia de la clase que implementa el `interface Runnable`. La Figura 14.1 muestra gráficamente este concepto.

Podemos utilizar tres técnicas distintas para crear un hilo en Java:

1. Extender a la clase `Thread`.
2. Definir una clase interna que implemente el `interface Runnable`.
3. Definir una clase interna anónima que implemente el `interface Runnable`.

Veamos cada una de estas técnicas por separado, y cuales son sus ventajas y desventajas.

14.2.1. Creación de un Hilo extendiendo a la clase `Thread`

La clase `Thread` implementa la `interface Runnable`, luego simplemente extendiéndola y sobrescribiendo su método `public void run()` tendremos un nuevo Hilo. El Listado 14.3 muestra el uso de esta primera técnica. Fíjate que al crear

el Hilo, este no empieza su ejecución, para indicar que el Hilo debe empezar a ejecutarse debemos llamar a su método `start`. El Hilo permanece vivo siempre que se esté ejecutando su método `public void run()`. El resultado de la ejecución de este Hilo es que se muestran 10 mensajes de texto por consola.

```
1 package hilos;
2
3 public final class ThreadExtendido extends Thread{
4  public ThreadExtendido() {
5   super();
6  }
7
8  @Override
9  public void run() {
10  for(int i = 0; i < 10; i++)
11   System.out.println("Estamos en: " + i);
12 }
13
14 public static void main(String[] args) {
15  new ThreadExtendido().start();
16 }
17 }
```

Listado 14.3: Creación de un Hilo extendiendo a la clase `Thread`

Fíjate que el método `public void run()` es un método público, luego, mientras el Hilo se está ejecutando, cualquier otra clase podría llamar al método `run()` de esta clase, provocando con ello inconsistencias durante la ejecución de nuestra aplicación. Por ello, esta técnica de creación de Hilos está desaconsejada.

14.2.2. Creación de un Hilo mediante una clase interna

El Listado 14.4 muestra un ejemplo de esta segunda técnica. Hemos definido la clase `ClaseInterna` que implementa el `interface Runnable`, dentro de la clase `HiloClaseInterna`. Y en la línea 20 creamos una instancia de la clase `Thread` con un argumento que es una instancia de la clase `ClaseInterna` que define en su método `public void run()` la tarea que se ejecuta concurrentemente. Igual que en el ejemplo anterior, el Hilo permanecerá vivo siempre que se encuentre ejecutando su método `run()`.

```
1 package hilos;
2
3 public final class HiloClaseInterna {
4  private HiloClaseInterna() {
5   super();
6  }
7
8  private class ClaseInterna implements Runnable {
9   private ClaseInterna() {
10   super();
11  }
12
13  public void run() {
14   for(int i = 0; i < 10; i++)
15    System.out.println("Estamos en: " + i);
16  }
17 }
18
19 private void ejecuta() {
20  new Thread(new ClaseInterna()).start();
21 }
22
23 public static void main(String[] args) {
```

```
24   new HiloClaseInterna().ejecuta();
25  }
26 }
```

Listado 14.4: Creación de un Hilo como una clase interna que implementa el `interface Runnable`

Al contrario que en la creación de hilos como extensiones de la clase `Thread`, en este caso la clase interna `ClaseInterna` es `private`, luego sólo se podrá acceder a su método `publi void run()` desde dentro de la clase en la que está definida, hemos evitado que otra clase llame de modo accidental al método `run()` previniendo con ello inconsistencias en nuestra aplicación.

Este método está recomendado cuando la tarea que se debe ejecutar de manera concurrente es compleja, y la clase interna implementa el algoritmo de ejecución de la tarea. Además, ya que la tarea concurrente está implementada dentro de una clase, podremos crear instancias de esta clase en cualquier otro lugar de la clase que la contiene.

14.2.3. Creación de un Hilo mediante una clase interna anónima

El Listado 14.5 muestra un ejemplo del uso de clases internas anónimas para la creación de Hilos. Este técnica es muy parecida a la técnica de la sección anterior salvo que se ha utilizado una clase interna anónima para implementar la tarea que se ejecuta de manera concurrente.

```
1 package hilos;
2
3 public final class HiloClaseInternaAnonima {
4  private HiloClaseInternaAnonima() {
5   super();
6  }
7
8  private void ejecuta() {
9   new Thread(new Runnable() {
10    @Override
11    public void run() {
12     for(int i = 0; i < 10; i++)
13      System.out.println("Estamos en: " + i);
14    }
15   }).start();
16  }
17
18  public static void main(String[] args) {
19   new HiloClaseInternaAnonima().ejecuta();
20  }
21 }
```

Listado 14.5: Creación de un Hilo como una clase interna anónima que implementa el `interface Runnable`

Esta técnica está recomendada cuando el código de la tarea concurrente es sólo de algunas líneas y no se va a reutilizar en ningún otro caso, ya que al ser una clase anónima no tienen nombre y no podríamos crear una instancia de una clase sin nombre en ninguna otra parte de nuestro código.

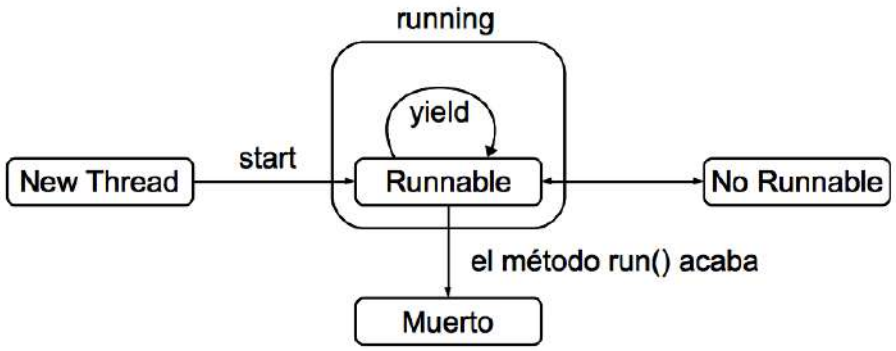

Figura 14.2: Estados en los que se puede encontrar un Hilo y las posibles transiciones entre ellos.

14.3. Ciclo de vida de un hilo

Los Hilos pueden encontrarse en más estados que únicamente en el estado de ejecución cuando están vivos. La Figura 14.2 muestra los estados en los que se puede encontrar un hilo y las transiciones entre los estados.

Un hilo inicia su ejecución cuando se llama a su método `public void start()`, y en este momento se dice que está en *Ejecución*. Durante su ejecución, un Hilo puede pasar al estado *Pausado* o al estado *Planificado*, en este estado el Hilo no se ejecuta, pero siempre podrá pasar de nuevo al estado en *Ejecución*. Un Hilo puede pasar al estado *Terminado* desde el estado en *Ejecución*, se dice que el Hilo se ha terminado, y un hilo *Terminado* nunca más puede pasar de nuevo al estado en *Ejecución*, *Pausado* o *Planificado*.

La diferencia entre los estados *Pausado* o *Planificado* es que de un estado *Planificado* conocemos el momento en que de nuevo pasará a *Ejecución*. De un estado *Pausado* no sabemos cuando volverá al estado *Ejecución*. Dependiendo del método de la clase `Thread` que utilicemos, podremos llevar un Hilo en *Ejecución* a uno de estos estados.

Un hilo pasa al estado *Terminado* cuando ocurre alguna de estas tres cosas:

1. Sale de su método `public void run()`.
2. Se produce una excepción dentro del método `run()` no gestionada.
3. Se llama al método `public void stop()` del Hilo.

Buena práctica

El uso del método `stop()` para detener la ejecución de un Hilo está fuertemente desaconsejado ya que puede provocar inconsistencias y de hecho está marcado como obsoleto en el API Java. Para acabar un Hilo limpiamente se aconseja salir de su método `run()`.

Esta buena práctica es la que se sigue en los ejemplo mostrados, cuando se ha iterado un cierto número de veces, se sale del método `public void run()`.

Otra técnica muy extendida es utilizar dentro del método `public void run()` un bucle `while(condicion)` cuya condición inicialmente es `true` y se cambia a `false` cuando se quiere acabar el Hilo limpiamente.

14.4. Control de hilos

El control de hilos se puede llevar a cabo desde una perspectiva *gruesa* tratando cada Hilo como un bloque, o desde una perspectiva más *fina* mediante el uso de la *sincronización*. En esta sección veremos como coordinar la ejecución de Hilos desde una perspectiva *gruesa* dejando para la siguiente sección la sincronización de Hilos desde una perspectiva más *fina*.

Podemos suspender temporalmente la ejecución de un Hilo mediante el uso del método `public static void sleep(long milisegundos)`[1]. Pasado el tiempo especificado, el Hilo volverá al estado en *Ejecución*. Es decir, con el método `sleep(long milisegundos)` *planificamos* que el Hilo vuelva a su estado *Ejecución* pasados los milisegundos especificados.

El Listado 14.6 muestra un ejemplo que muestra un mensaje cada 1000 milisegundos, es decir, cada segundo. El método `sleep(long)` puede lanzar una excepción de tipo `InterruptedException` que debemos gestionar.

```
1 package hilos;
2
3 public final class EjemploSleep {
4  private EjemploSleep() {
5   super();
6  }
7
8  private void ejecuta() {
9   new Thread(new Runnable() {
10    @Override
11    public void run() {
12     try {
13      for(int i = 0; i < 10; i++) {
14       System.out.println("Estamos en: " + i);
15       Thread.sleep(1000);
16      }
17     } catch(InterruptedException e) {
18      e.printStackTrace();
19     }
20    }
21   }).start();
22  }
23
24  public static void main(String[] args) {
25   new EjemploSleep().ejecuta();
26  }
27
28 }
```

Listado 14.6: Uso del método `sleep(long milisegundos)` para pausar la ejecución del Hilo durante el intervalo de tiempo espedificado.

Si un Hilo está en estado *Pausado* o *Planificado* siempre lo podremos sacar de este estado con el método `void interrupt()`. Este método lanza una excepción de tipo `InterruptedException`.

Existen casos en los que un Hilo debe esperar a que otro acabe su ejecución para que él pueda continuar. Piensa por ejemplo en un Hilo que cuenta el número

[1] También existe una versión de mayor precisión `public static void sleep(long milisegundos, int nanosegundos`

de caracteres que se leen desde un fichero. Si la lectura del fichero se hace en un Hilo y la cuenta del número de caracteres en otro, el Hilo que cuenta el número de caracteres debe esperar a que el Hilo que carga el fichero acabe su tarea antes de empezar con la cuenta.

Para que un Hilo espere a que otro acabe su ejecución llamaremos al método `public final void join()`[2] del segundo Hilo en el código del primero, tal y como muestra el ejemplo del Listado 14.7.

```
1 package hilos;
2
3 public final class EjemploJoin {
4  private Thread hilo;
5
6  private EjemploJoin() {
7   super();
8  }
9
10 private void ejecuta() {
11  hilo = new Thread(new Tarea());
12  hilo.start();
13  new Thread(new TareaQueEspera()).start();
14 }
15
16 private class Tarea implements Runnable {
17  public void run() {
18   try {
19    for(int i = 0; i < 10; i++) {
20     System.out.println("Cuenta: " + i);
21     Thread.sleep(500);
22    }
23   } catch(InterruptedException e) {
24    e.printStackTrace();
25   }
26  }
27 }
28
29 private class TareaQueEspera implements Runnable {
30  public void run() {
31   try {
32    for(int i = 0; i < 5; i++) {
33     System.out.println("Cuenta y espera: " + i);
34     Thread.sleep(500);
35    }
36
37    hilo.join();
38
39    for(int i = 5; i < 10; i++) {
40     System.out.println("Salgo de la espera y cuenta: " + i);
41     Thread.sleep(500);
42    }
43   } catch(InterruptedException e) {
44    e.printStackTrace();
45   }
46  }
47 }
48
49 public static void main(String[] args) {
50  new EjemploJoin().ejecuta();
51 }
52 }
```

Listado 14.7: Uso del método `join()` para pausar la ejecución del Hilo hasta que acabe la ejecución de otro.

[2]Existen dos versiones de este método `public final void join(long milisegundo` y `public final void sleep(long milisegundos, int nanosegundos` en los que se espera, como máximo, el tiempo especificado

14.5. Sincronización

En los ejemplos anteriores el control sobre Hilos no implicaba acceso concurrente a recursos. Cuando varios Hilos que se ejecutan de modo concurrente intentan acceder al mismo recurso debemos sincronizar los accesos al recurso para que no se produzcan inconsistencias en nuestras aplicaciones.

La sincronización en Java se base en el uso de cerrojos, varios Hilos que intentan acceder a un recurso compartido sincronizan su acceso a través de un cerrojo. Antes de acceder al recurso compartido un Hilo intentará adquirir un cerrojo, si el cerrojo no está en posesión de ningún otro Hilo lo adquirirá, de lo contrario tendrá que esperar a que el cerrojo se libere. Una vez en posesión del cerrojo puede trabajar con el recurso compartido con la seguridad de que ningún otro Hilo accederá a este recurso hasta que él no libere el cerrojo que indica la propiedad de uso del recurso.

¿Y donde se encuentran esos cerrojos que nos sirven para sincronizar el acceso a recursos compartidos?. Todo objeto tiene un *cerrojo intrínseco*, recuerda que los Hilos y su sincronización son nativos en Java.

En las secciones siguientes vamos a ver cómo sincronizar el acceso a recursos compartidos con dos técnicas: mediante el uso de cerrojos intrínsecos y mediante la `interface Lock` y la `interface Condition` introducidas ambas en la versión 5.0 de Java.

14.5.1. Sincronizacón utilizando los cerrojos intrínsecos

En Java todo objeto posee un cerrojo intrínseco que podemos utilizar para sincronizar el acceso a los recursos que nos proporciona el objeto. Como ya sabemos, un método es un recurso o servicio que proporciona un objeto, si queremos sincronizar el acceso a un método de un objeto basta marcarlo con la palabra reservada `synchronized`, como en el Listado 14.8.

```
1 class A {
2  synchronized void metodo1() {}
3  synchronized void metodo2() {}
4  ...
5 }
```

Listado 14.8: Para indica que el acceso a un método está sincronizado utilizamos la palabra reservada `synchronized`.

Definir un método como `synchronized` implica que antes de que un Hilo pueda ejecutar el método, debe adquirir el *cerrojo intrínseco* del objeto para poder ejecutarlo. Si el *cerrojo intrínseco* está en posesión de otro Hilo, el Hilo que intenta adquirir el cerrojo tendrá que esperar a que el otro lo libere. El *cerrojo intrínseco* pertenece al objeto y sincroniza el acceso a todos los métodos definidos como `synchronized`, si un Hilo adquiere el *cerrojo intrínseco* accediendo al `metodo1()` del ejemplo anterior, y otro Hilo intenta acceder al `metodo2()` no podrá hacerlo ya que no podrá adquirir el *cerrojo intrínseco* del objeto. Un Hilo libera el cerrojo cuando acaba la ejecución del método sincronizado, y en ese momento cualquier otro Hilo que se encuentre a la espera podrá adquirir el cerrojo.

Como el *cerrojo intrínseco* pertenece al objeto, la sincronización en el ejemplo del Listado 14.8 es relativa a todo el objeto, si un Hilo está en posesión del *cerrojo*

intrínseco, cualquier otro Hilo no podrá acceder a ningún otro método definido como `synchronized`.

Hay casos en los que no nos interesa sincronizar el acceso a los método del objeto, si no sincronizar sólo bloques de código. Incluso a veces nos interesa tener varios bloques de código sincronizados pero que el acceso a distintos bloques de código no esté sincronizado, si no que la sincronización sea relativa al bloque y no al objeto completo.

Java nos permite sincronizar el acceso a bloque de código también mediante el uso de la palabra reservada `synchronized`, tal y como se muestra en el Listado 14.9. Un Hilo puede estar ejecutando el bloque sincronizado por el `cerrojo1` mientras que otro Hilo puede estar ejecutando el bloque sincronizado por el `cerrojo2`; pero ningún otro Hilo podrá acceder a estos bloques sincronizados hasta que los Hilos en posesión de los cerrojos no abandonen el bloque sincronizado.

```
1 class A {
2  Object cerrojo1 = new Object();
3  Object cerrojo2 = new Object();
4
5  void metodo1() {
6   ...
7   synchronized(cerrojo1) {
8    // El acceso a este bloque está sincronizado
9    // por cerrojo1.
10  }
11  ...
12 }
13
14 void metodo2() {}
15  ...
16  synchronized(cerrojo2) {
17   // El acceso a este bloque está sincronizado
18   // por cerrojo2.
19  }
20 }
21 }
```

Listado 14.9: Cada uno de los bloques de código está sincronizado por un *cerrojo intrínseco* de un objeto distinto. El acceso sólo está sincronizado si el acceso es al mismo bloque de código.

Fíjate que, por otra parte, esta técnica nos permite sincronizar con más detalle el código conflictivo, no sincronizamos todo un método si no sólo las líneas de código que acceden al recurso compartido. Incluso podemos utilizar esta técnica para sincronizar bloques de código en distintos métodos con el mismo cerrojo.

Ya hemos visto que cuando un Hilo no puede adquirir el cerrojo para acceder a un método o bloque sincronizado tiene que esperar a que el Hilo que tiene en posesión el cerrojo lo libere, y que el cerrojo se libera cuando el Hilo que lo tiene en posesión acaba la ejecución del método o bloque sincronizado. También podemos forzar que un Hilo entre en espera mediante el método `public final void wait() throws InterruptedException`[3]. Cuando un Hilo entra en espera, libera el *cerrojo intrínseco* que tenga en posesión, y esta es la diferencia con respecto al método `sleep(int milisegundos)` que provoca una espera del

[3]Existen otras dos versiones de este método `public final void wait(long timeout) throws InterruptedException` y `public final void wait(long timeout, int nanos) throws InterruptedException`

Hilo, pero el Hilo no libera el *cerrojo intrínseco* que tenga en posesión. El uso del método `wait()` lleva un Hilo desde el estado *Ejecución* al estado *Pausado* ya que no conocemos cuando el Hilo volverá al estado *Ejecución*.

Un Hilo que ha entrado en espera mediante una llamada a `wait()` saldrá de este estado si ocurre alguna de estas cuatro cosas:

- Algún otro Hilo llama al método `public final void notify()`.
- Algún otro Hilo llama al método `public final void notifyAll()`.
- Algún otro Hilo llama al método `public void interrupt()`.

Mientras un Hilo permanece en estado de espera nunca será elegido para competir por la adquisición del cerrojo.

Existe la posibilidad de que el Hilo en estado de espera sufra un *despertar espúreo*, lo que implica que el Hilo se despierte aunque no ocurra ninguna circunstancia de la indicada anteriormente, de modo que, si detuvimos el Hilo porque no se comprobaba cierta condición, si el Hilo se despierta de modo espúreo puede que la condición siga sin satisfacerse, por lo que siempre debemos comprobar que la condición se satisface cuando un Hilo sale del estado de espera.

El Listado 14.10 muestra una posible implementación de un *Buffer* de tamaño fijo con sus operaciones `getDato()` y `setDato(T dato)` sincronizadas y de modo que si un Hilo intenta tomar un dato y el *Buffer* está vacío el Hilo debe esperar (`wait()`), y si el Hilo intenta poner un nuevo dato y el *Buffer* está lleno también debe esperar.

```
1 package buffer;
2
3 public class BufferSinLock<T> {
4  private int cabeza = 0;
5  private int capacidad;
6  private Object datos[];
7  private int ocupacion = 0;
8
9  public BufferSinLock(int capacidad) {
10  super();
11  this.capacidad = capacidad;
12  datos = new Object[capacidad];
13 }
14
15 @SuppressWarnings("unchecked")
16 public synchronized T getDato() throws InterruptedException {
17  T dato;
18  while(ocupacion == 0)
19   wait();
20  dato = (T)datos[cabeza];
21  ocupacion--;
22  cabeza++;
23  cabeza %= capacidad;
24  System.out.format("-%s [%d]\n", dato, ocupacion);
25  notifyAll();
26  return dato;
27 }
28
29 public synchronized void setDato(T dato) throws InterruptedException {
30  while(ocupacion == capacidad)
31   wait();
32  datos[(cabeza + ocupacion)%capacidad] = dato;
33  ocupacion++;
34  System.out.format("+%s[%d]\n", dato, ocupacion);
35  notifyAll();
```

```
36  }
37 }
```

Listado 14.10: Implementación de un *Buffer* de tamaño fijo utilizando *cerrojos intrínsecos*

En el Apéndice C se muestra un ejemplo de uso de este *Buffer*.

14.5.2. Sincronización utilizando el `interface Lock`

Desde la versión Java 5, se ha enriquecido el API de Java con nuevos paquetes para facilitar la programación con Hilos. Estos nuevos paquetes son:

`java.util.concurrent` Que proporciona nuevas estructuras de datos y clases de utilidad de acceso concurrente.

`java.util.concurrent.atomic` Conjunto de clases para acceso atómico a tipos de datos primitivos y referencia.

`java.util.concurrent.locks` Conjunto de clases para facilitar la sincronización y acceso concurrente a recursos compartidos.

El último paquete `java.util.concurrent.locks` de la lista anterior proporciona clases que facilitan la sincronización concurrente a recursos compartidos. En particular la `interface Lock` y las clases que lo implementan `ReentrantLock`, `ReentrantReadWriteLock.ReadLock` y `ReentrantReadWriteLock.WriteLock` son de gran ayuda. La idea básica es que un Hilo debe obtener el cerrojo que representa alguna de estas clases antes de poder ejecutar el código que nos interesa de manera concurrente, y una vez que se ha terminado la ejecución del código crítico, debemos liberar el cerrojo. La técnica recomendada para hacerlo se muestra en el Listado 14.11

```
1 Lock l = ...; // creamos alguna instancia que implemente a Lock
2 l.lock();
3 try {
4     // acceso a los recursos compartidos
5 } finally {
6     l.unlock();
7 }
```

Listado 14.11: Modo aconsejado de trabajar con los objetos que implementan la `interface Lock`

Otra `interface` de gran ayuda es `Condition` con la que podemos definir varias condiciones sobre un mismo `Lock` de modo que el hilo que adquiere el `Lock` puede pausarse, liberando el `Lock`, utilizando un `Condition` si la condición de ocupación del Hilo no es válida, y se le informará a través de esa misma `Condition` de que la condición de ocupación del Hilo puede ser válida de nuevo.

Como ejemplo de funcionamiento de estas interfaces, el Listado 14.12 muestra una implementación de un *Buffer* de tamaño fijo y acceso sincronizado, como el del Listado 14.10 pero esta vez haciendo uso de estas interfaces. En el Apéndice C se muestra un ejemplo de uso de este *Buffer*.

```
1 package buffer;
2
```

```
3 import java.util.concurrent.locks.Condition;
4 import java.util.concurrent.locks.Lock;
5 import java.util.concurrent.locks.ReentrantLock;
6
7 public class BufferConLock<T> {
8  private int cabeza = 0;
9  private int capacidad;
10  private Object datos[];
11  private int ocupacion = 0;
12  private Lock cerrojo;
13  private Condition condicion;
14
15  public BufferConLock(int capacidad) {
16   super();
17   this.capacidad = capacidad;
18   datos = new Object[capacidad];
19   cerrojo = new ReentrantLock();
20   condicion = cerrojo.newCondition();
21  }
22
23  @SuppressWarnings("unchecked")
24  public T getDato() throws InterruptedException {
25   T dato;
26   cerrojo.lock();
27   try {
28    while(ocupacion == 0)
29     condicion.await();
30    dato = (T)datos[cabeza];
31    ocupacion--;
32    cabeza++;
33    cabeza %= capacidad;
34    System.out.format("-%s[%d]\n", dato, ocupacion);
35   } finally {
36    condicion.signalAll();
37    cerrojo.unlock();
38   }
39   return dato;
40  }
41
42  public void setDato(T dato) throws InterruptedException {
43   cerrojo.lock();
44   try {
45    while(ocupacion == capacidad)
46     condicion.await();
47    datos[(cabeza + ocupacion)%capacidad] = dato;
48    ocupacion++;
49    System.out.format("+%s[%d]\n", dato, ocupacion);
50   } finally {
51    condicion.signalAll();
52    cerrojo.unlock();
53   }
54  }
55 }
```

Listado 14.12: Implementación de un *Buffer* de tamaño fijo utilizando las clases `ReentrantLock` y `Condition`

¿Cuando utilizar los *cerrojos intrínsecos* y cuando el `interface Lock`?. La sección 13.2 de la referencia [1] clarifica mucho cuando utilizar una técnica u otra.

Ejercicios.

1. Escribe una aplicación en la que varios Hilos intenten acceder a un recurso compartido que muestre un mensaje por consola. El acceso a este recurso compartido debe hacerse de modo sincronizado.

Lecturas recomendadas.

- El Capítulo 10 de la referencia [2] es sin duda una muy buena exposición de las técnicas de concurrencia en Java utilizando *cerrojos intrínsecos*.
- La referencia [1] es obligada de principio a fin si la concurrencia es capital en nuestras aplicaciones.
- El Capítulo 9 de la referencia [4] expone criterios muy interesante a seguir en el uso de Hilos para que no se vea reducida la potencia de nuestras aplicaciones por un mal uso de los Hilos.

Capítulo 15

Programación para la Red

Contenidos

15.1. Trabajo con URLs 222
15.1.1. ¿Qué es una URL? 222
15.1.2. Leer desde una URL 223
15.1.3. Escribir a una URL 223
15.2. Trabajo con Sockets 225
15.2.1. ¿Qué es un Socket? 225
15.2.2. Sockets bajo el protocolo TCP 225
15.2.2.1. Sockets TCP en el lado del servidor 225
15.2.2.2. Sockets TCP en el lado del cliente 226
15.2.3. Sockets bajo el protocolo UDP 227
15.2.3.1. Sockets UDP en el lado del servidor 228
15.2.3.2. Sockets UDP en el lado del cliente 228

Introducción

Java nació como un lenguaje de propósito general con grandes capacidades para trabajar en red. En el Capítulo 12 vimos cómo programar Applets, que son aplicaciones Java que se ejecutan en el contexto de un navegador web. Java nos permite trabajar en red de un modo muy sencillo gracias a la gran cantidad de clases que nos proporciona el paquete `java.net`.

En este capítulo veremos como trabajar con *URLs*, pilar básico para referenciar recursos en el protocolo HTTP, y como trabajar con *Sockets* tanto bajo el protocolo *TCP* como bajo el protocolo *UDP*.

Como verás, tanto si trabajamos con URLs como si trabajamos con Sockets, la lectura y escritura sobre estos canales la haremos a través de los *Flujos* que vimos en el Capítulo 7. Recuerda que todo proceso de lectura/escritura es independiente del dispositivo de entrada/salida y se realiza a través de *Flujos*, de ahí la importancia de conocerlos a fondo.

15.1. Trabajo con URLs

En esta sección vamos a recordar qué es una URL. Veremos que en el paquete `java.net` tenemos una clase para crear URLs. Esta clase nos permite tanto leer el contenido de la URL como escribir hacia la URL.

15.1.1. ¿Qué es una URL?

Una URL representa un recurso en Internet. Este recurso viene especificado por cuatro partes, por ejemplo en la siguiente URL `http://www.google.es:80/index.html`, se está especificando:

1. El protocolo `http`.
2. El host de destino `www.google.es`.
3. El puerto de conexión `80`.
4. El fichero solicitado `index.html`.

Los servidores que atienden peticiones HTTP generalmente utilizan el puerto de escucha 80, y si no se indica la contrario, el fichero al que por defecto se accede es `index.html`[1]; por lo que la anterior URL la podemos escribir de forma abreviada de modo `http://www.google.es`.

En esta sección vamos a utilizar como ejemplo siempre el protocolo HTTP, aunque, como el lector sabrá, existe otra gran cantidad de protocolos sobre TCP.

Java nos proporciona la clase `URL` para poder especificar recursos en Internet como muestra el Listado 15.1. En el primer caso especificamos la URL como una única cadena. En el segundo caso lo especificamos como cuatro cadenas indicando el protocolo, dirección de Internet, puerto y recurso respectivamente. Si el protocolo tiene un puerto bien conocido no hace falta especificarlo, como en el tercer caso. Todos estos constructores pueden lanzar una excepción de tipo `MalformedURLException` para indicar que la URL que se intenta especificar no es válida.

```
1 URL url = new URL("http://www.google.es:80/index.html");
2 URL url = new URL("http", "www.uji.es", 80, "index.html");
3 URL url = new URL("http", "www.uji.es", "index.html");
```

Listado 15.1: Algunos constructores de la clase `URL`

La clase `URL` nos proporciona métodos para recuperar la información contenida en una URL:

- `public String getProtocol()`, devuelve el protocolo.
- `public String getHost()`, devuelve el host.
- `public int getPort()`, devuelve el puerto de conexión.
- `public String getFile()`, devuelve el recurso que se solicita.

Pasemos ahora a ver cómo leer desde una URL.

[1] La página de inicio de un sitio web también puede ser `index.js`, o `index.cgi`

15.1.2. Leer desde una URL

Una vez que tenemos construida una URL válida, podemos recuperar a partir de ella un `InputStream` para leer el contenido del recurso al que apunta la URL, mediante el método `public final InputStream openStream() throws IOException`, y como ya vimos en el Capítulo 7 a partir de una referencia de tipo `InputStream` podemos recuperar un `BufferedReader` con el que leer línea a línea desde el recurso. El Listado 15.2 muestra un ejemplo completo de cómo leer el contenido de un fichero de esto apuntado por una URL.

```
1 package red;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStream;
6 import java.io.InputStreamReader;
7 import java.net.MalformedURLException;
8 import java.net.URL;
9
10 public final class LeerDesdeURL {
11  private LeerDesdeURL() {
12   super();
13  }
14
15  private void ejecuta(String direccion) {
16   try {
17    URL url = new URL(direccion);
18    InputStream is = url.openStream();
19    BufferedReader br = new BufferedReader(new InputStreamReader(is));
20    try {
21     String cadena;
22     while((cadena = br.readLine()) != null)
23      System.out.println(cadena);
24    } finally {
25     br.close();
26    }
27   } catch (MalformedURLException e) {
28    e.printStackTrace();
29   } catch (IOException e) {
30    e.printStackTrace();
31   }
32  }
33
34  public static void main(String[] args) {
35   new LeerDesdeURL().ejecuta(args[0]);
36  }
37 }
```

Listado 15.2: Lectura desde una URL

15.1.3. Escribir a una URL

Escribir a una URL exige un poco más de trabajo, ya que no podemos escribir directamente a una URL. Para escribir a una URL necesitamos una referencia a la clase `URLConnection`. La referencia necesaria nos la devuelve el método `public URLConnection openConnection() throws IOException` de la clase `URL`. Una vez obtenida esta referencia, debemos indicar que queremos escribir sobre ella, para lo que utilizaremos el método `public void setDoOutput(boolean dooutput)` de la clase `URLConnection` con un argumento `true`. Ahora ya podemos recuperar un `OutputStream` con el método `public OutputStream getOutputStream() throws IOException` de la clase `URConnection`

El Listado 15.3 muestra todos los pasos necesarios para escribir a una URL y obtener una respuesta de ella. En este caso hemos utilizado la dirección `http://rubi.dlsi.uji.es/~oscar/PHP/nombre.php` que espera un parámetro de entrada llamado *nombre* y devuelve un saludo[2].

```
1 package red;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.MalformedURLException;
8 import java.net.URL;
9 import java.net.URLConnection;
10
11 public final class EscribirAURL {
12  private EscribirAURL() {
13   super();
14  }
15
16  private void ejecuta(String sUrl) {
17   try {
18    URL url = new URL(sUrl);
19    URLConnection conexion = url.openConnection();
20    conexion.setDoOutput(true);
21    PrintWriter pw = new PrintWriter(conexion.getOutputStream(), true);
22    pw.println("nombre=oscar");
23    BufferedReader br = new BufferedReader(new InputStreamReader(conexion
          .getInputStream()));
24    try {
25     String respuesta;
26     while((respuesta = br.readLine()) != null)
27      System.out.println(respuesta);
28    } finally {
29     pw.close();
30     br.close();
31    }
32   } catch(MalformedURLException e) {
33    e.printStackTrace();
34   } catch(IOException e) {
35    e.printStackTrace();
36   }
37  }
38
39  public static void main(String[] args) {
40   new EscribirAURL().ejecuta(args[0]);
41  }
42 }
```

Listado 15.3: Escritura a una URL

En el Listado 15.3, presta atención al segundo parámetro del constructor de `PrintWriter` que es `true` para indicar que se haga *auto-flush* cada vez que escribimos en el *stream*.

Si el protocolo que utilizamos en la conexión es HTTP, podemos modelar la referencia de respuesta del método `openConnection()` al tipo `HttpURLConnection`. Esta nueva clase tiene métodos que nos permiten especificar el tipo de petición que se realiza `public void setRequestMethod(String method) throws ProtocolException` para indicar si la petición es GET, POST, HEAD, etc.; y otros métodos interesantes para saber el código de estado de la petición (`public int getResponseCode() throws IOException`).

[2]Para ver el resultado en un navegador, teclea la dirección http://rubi.dlsi.uji.es/ oscar/PHP/nombre.php?nombre=oscar y obtendrás un saludo como respuesta

15.2. Trabajo con Sockets

Los *Sockets* son el ladrillo fundamental en comunicaciones a través del protocolo TCP o UDP. Como en el caso del trabajos con URLs veremos que una vez establecida la conexión entre dos Sockets (cliente/servidor) todas las tareas de lectura y escritura entre los Sockets se realizarán sobre los *Flujos* que la clase `Socket` nos proporciona.

15.2.1. ¿Qué es un Socket?

Un Socket es cada uno de los extremos que se establece en una comunicación, en toda comunicación tendremos un Socket en el lado del emisor y un Socket en el lado del receptor.

En el paquete `java.net` encontramos clases para trabajar con Sockets tanto bajo el protocolo TCP como bajo el protocolo UDP. Como recordatorio, el protocolo TCP está orientado a la conexión, mientras que el protocolo UDP está orientado al mensaje.

15.2.2. Sockets bajo el protocolo TCP

En las conexiones bajo el protocolo TCP existe un canal de conexión entre el cliente y el servidor. Entre otras cosas el protocolo TCP garantiza la recepción de los datos en el mismo orden en que se emiten y sin pérdidas. El símil que se utiliza para visualizar las conexiones bajo el protocolo TCP es el de una llamada de teléfono, en la que se reserva un canal por el que la información circula sin pérdidas y de manera ordenada.

15.2.2.1. Sockets TCP en el lado del servidor

Para crear un Socket que acepte conexiones en un determinado puerto, Java nos proporciona la clase `java.net.ServerSocket`, en el momento de crear una instancia de esta clase indicamos el puerto de escucha. Para aceptar conexiones utilizamos el método `public Socket accept() throws IOException`, que es bloqueante, es decir, la ejecución se detiene en este método hasta que un cliente se conecta. Cada vez que un cliente se conecta al Socket servidor, el método `accept()` nos devolverá una referencia al Socket cliente (instancia de la clase `Socket` esta vez), a partir de la cual podremos obtener *Flujos* de lectura o escritura.

El Listado 15.4 muestra un ejemplo de un sencillo Servidor que envía un saludo a cada cliente que se le conecta. En la línea 41 puedes ver cómo la clase `Socket` nos devuelve un *Flujo* de escritura para comunicarnos con el cliente.

```
1 package red;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import java.net.ServerSocket;
6 import java.net.Socket;
7
8 public final class ServidorSencillo {
9  private ServidorSencillo() {
10   super();
11  }
12
```

```
13  private void ejecuta(int puerto) {
14   try {
15    System.out.println("Servidor a la escucha");
16    // Creamos un Socket servidor a la escucha en el puerto indicado
17    ServerSocket servidor = new ServerSocket(puerto);
18    Socket cliente;
19    try {
20     // Cada vez que se conecta un cliente le enviamos un saludo
21     while((cliente = servidor.accept()) != null)
22      new Thread(new Saludo(cliente)).start();
23    } finally {
24     servidor.close();
25    }
26   } catch (IOException e) {
27    e.printStackTrace();
28   }
29
30  }
31
32  private class Saludo implements Runnable {
33   private Socket cliente;
34
35   public Saludo(Socket cliente) {
36    this.cliente = cliente;
37   }
38
39   @Override
40   public void run() {
41    System.out.println("Cliente conectado");
42    try {
43     // Obtenemos un stream de escritura a partir del Socket del cliente
44     PrintWriter pw = new PrintWriter(cliente.getOutputStream(), true);
45     pw.println("Hola desde el servidor");
46     pw.close();
47    } catch (IOException e) {
48     e.printStackTrace();
49    }
50   }
51  }
52
53  public static void main(String args[]) {
54   new ServidorSencillo().ejecuta(Integer.parseInt(args[0]));
55  }
56 }
```

Listado 15.4: Un Socket servidor que envía un saludo a cada cliente que se le conecta

Si, por ejemplo, inicias el servidor en el puerto 1234, pasando este entero por la línea de argumentos, podrás conectarte al servidor desde un navegador web en la dirección `http://localhost:1234`. Y obtendrás como resultado el mensaje de saludo. También puedes conectarte al servidor mediante `telnet` escribiendo en una consola `telnet localhost 1234`.

15.2.2.2. Sockets TCP en el lado del cliente

Para conectarnos a un Socket servidor desde Java disponemos de la clase `java.net.Socket`. Al crear una instancia de esta clase le pasamos la dirección a la que nos queremos conectar y en que puerto, y una vez establecida la conexión podremos abrir *Flujos* de lectura y escritura.

El Listado 15.5 muestra un ejemplo de un cliente que se conecta al servidor del Listado 15.4 para obtener un saludo como respuesta.

```
1 package red;
2
```

```
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.net.Socket;
7 import java.net.UnknownHostException;
8
9 public final class ClienteSencillo {
10  private ClienteSencillo() {
11   super();
12  }
13
14  private void ejecuta(int puerto) {
15   try {
16    // Me conecto al servidor local que escuha es este puerto
17    Socket cliente = new Socket("localhost", puerto);
18    try {
19     // Recupero un stream de lectura
20     BufferedReader br = new BufferedReader(new InputStreamReader(cliente
           .getInputStream()));
21     String saludo;
22     while((saludo = br.readLine()) != null)
23      System.out.println(saludo);
24    } finally {
25     if(cliente != null) cliente.close();
26    }
27   } catch (UnknownHostException e) {
28    e.printStackTrace();
29   } catch (IOException e) {
30    e.printStackTrace();
31   }
32  }
33
34  public static void main(String[] args) {
35   new ClienteSencillo().ejecuta(Integer.parseInt(args[0]));
36  }
37 }
```

Listado 15.5: Un Socket clinet que conecta al anterior servidor para recibir un saludo.

15.2.3. Sockets bajo el protocolo UDP

En el caso del Protocolo UDP no se garantiza que los mensajes recibidos lo hagan en el mismo orden que los enviados, y tampoco se garantiza que al receptor lleguen todos los mensajes del emisor, existe la posibilidad de que alguno se pierda por el camino. En este caso el símil que se utiliza es el de comunicaciones a través de un servicio postal.

En el caso del protocolo UDP, al contrario que en el caso del protocolo TCP, no disponemos de dos clases diferenciadas dependiendo de si lo que queremos crear es un Socket servidor o un Socket cliente.

En el caso del protocolo UDP disponemos de dos clases para realizar la comunicación. La clase `DatagramPacket` la utilizaremos para especificar los datos que queremos enviar o recibir. La clase `DatagramSocket` es la que se encarga de enviar o recibir cada uno de los `DatagramPacket`.

Dependiendo del constructor que utilicemos para instanciar la clase `DatagramPacket` estaremos creando un datagrama para enviar datos, o un datagrama para recibir datos.

De modo análogo, dependiendo del constructor que utilicemos para instanciar la clase `DatagramSocket` estaremos creando un Socket UDP capaz de recibir datos en un cierto puerto, o un Socket UDP capaz de enviar datos a un determinado servidor.

15.2.3.1. Sockets UDP en el lado del servidor

Para crear un Socket UDP en el lado del servidor debemos utilizar el constructor de la clase `DatagramSocket` que recibe como parámetro el puerto de conexión `public DatagramSocket(int puerto)`. Una vez creado el Socket UDP, la clase `DatagramSocket` nos permitirá tanto enviar como recibir paquetes en el puerto indicado.

Para recibir paquetes necesitamos crear una instancia de la clase `DatagramPacket` indicando el buffer sobre el que se leerán los datos y su tamaño, por ejemplo `new DatagramPacket(new byte[100], 100)` nos permitirá leer hasta 100 bytes del paquete recibido, si el paquete recibido contiene más de 100 bytes, el resto será ignorado. El Listado 15.6 muestra un ejemplo de lectura de un paquete UDP.

```
1 byte[] buffer = new byte[8*1024];
2 DatagramPacket paquete = new DatagramPacket (buffer, buffer.length);
3 DatagramSocket servidor = new DatagramSocket(12345); // Puerto de
      escucha 12345
4 servidor.receive(paquete); // El método receive es bloqueante
5 InetSocketAddress direccionCliente = paquete.getSocketAddress();
6 System.out.println("Host cliente: " + direccionCliente.getHostName());
```

Listado 15.6: Lectura de un paquete mediante el protocolo UDP.

Para enviar un paquete como contestación al cliente del que se acaba de recibir un paquete crearíamos una nueva instancia de la clase `DatagramPacket` pero esta vez utilizando el constructor que incluye un tercer parámetro con la dirección de destino del paquete `public DatagramPacket(byte[] buffer, int tamBuffer, SocketAddress direccion)`. El listado 15.7 es una continuación del listado 15.6 donde el servidor envía un paquete de respuesta al cliente.

```
7 bytes[] mensaje = "Hola".getBytes();
8 DatagramPacket paqueteRespuesta = new DatagramPacket(mensaje, mensaje.
      length, direccionCliente);
9 servidor.send(paqueteRespuesta);
```

Listado 15.7: Escritura de un paquete mediante el protocolo UDP.

Como puedes observar en el Listado 15.7, es posible utilizar la misma instancia de la clase `DatagramSocket` para enviar datos. La dirección a la cual se debe enviar el paquete la obtiene la clase `DatagramSocket` a partir de la clase `DatagramPacket`.

15.2.3.2. Sockets UDP en el lado del cliente

Para escribir un cliente utilizando el protocolo UDP utilizamos de nuevo las clases `DatagramPacket` y `DatagramSocket`, pero esta vez, al crear una instancia de `DatagramPacket`, indicamos la dirección y el puerto del servidor al que queremos hacer llegar el paquete, por ejemplo `new DatagramPacket(new byte[1], 1, inetAddress, 12345)`. Para crear la instancia de `DatagramSocket` utilizaremos el constructor por defecto `new DatagramSocket()`. Finalmente, para enviar el paquete utilizaremos de nuevo el método `send(paquete)` de la clase `DatagramSocket`. En el Listado 15.8 se muestra un ejemplo de envío de un paquete hacia un servidor UDP.

```
1 DatagramPacket paquete = new DatagramPacket(new byte[1], 1, inetAddress,
      12345);
2 DatagramSocket servidor = new DatagramSocket();
3 servidor.send(paquete);
```

Listado 15.8: Envío de un paquete mediante el protocolo UDP desde un cliente a un servidor.

Igual que en el caso del servidor, el cliente puede quedar a la escucha de los paquetes que reciba desde el servidor, como se muestra en el Listado 15.9 que es continuación del Listado 15.8.

```
4 byte[] buffer = new byte[8*1024];
5 paquete = new DatagramPacket(buffer, buffer.length);
6 servidor.receive(paquete);
```

Listado 15.9: Recepción de paquetes por parte de un cliente.

Cuestiones.

1. Tanto el método `accept()` de la clase `ServerSocket` como el método `receive()` de la clase `DatagramSocket` son bloqueantes. ¿Qué debes utilizar para que tu aplicación no se quede «congelada» esperando a que se retorne de estos dos métodos?.
2. El protocolo UDP no comprueba que un paquete ha llegado efectivamente a su destino. ¿Cómo podrías asegurarte de que sí que llega a su destino?
3. El protocolo UDP tampoco tiene en cuenta que el orden de los paquetes recibidos es el mismo que el de los paquetes enviados. ¿Cómo podrías reordenar los paquetes que recibe tu aplicación para garantizar que siguen el mismo orden con el que se enviaron?
4. Al utilizar Sockets bajo el protocolo TCP, ¿Cómo puedes darte cuenta de que la conexión entre el cliente y el servidor se ha cortado? ¿Y en el caso del protocolo UDP?

Ejercicios.

1. Itenta crear una aplicación de chat. En la aplicación hay una parte de servidor, que atiende la conexión de nuevos clientes. Cada vez que un cliente se conecta al servidor de chat queda registrado de modo que recibe todos los mensajes que envían el resto de clientes.

Lecturas recomendadas.

- Una excelente referencia para casi cualquier tema relacionado con la programación para la red es el libro de Elliotte Rusty Harold [12].

Capítulo 16

Patrones de diseño

Contenidos

16.1. Principios de POO 232
16.2. ¿Qué son los patrones de diseño? 233
16.3. ¿Qué es el acoplamiento entre clases y por qué hay que evitarlo? 233
16.4. Grupos de patrones de diseño 233
16.5. El patrón de diseño *Singleton* 233
16.5.1. Situación que intenta resolver 234
16.5.2. Ejemplo de implementación 234
16.6. El patrón de diseño *Factory Method* 235
16.6.1. Situación que intenta resolver 235
16.6.2. Ejemplo de implementación 236
16.7. El patrón de diseño *Abstract Factory* 238
16.7.1. Situación que intenta resolver 238
16.7.2. Ejemplo de implementación 238
16.8. El patrón de diseño *Strategy* 244
16.8.1. Situación que intenta resolver 245
16.8.2. Ejemplo de implementación 245
16.9. El patrón de diseño *Observer* 247
16.9.1. Situación que intenta resolver 247
16.9.2. Ejemplo de implementación 248
16.10El patrón de diseño *Decorator* 249
16.10.1Situación que intenta resolver 250
16.10.2Ejemplo de implementación 250

Introducción

En este capítulo se presenta, en primer lugar y a modo de resumen, los *Principios de Programación Orientada a Objetos*.

A continuación se define lo que son los patrones de diseño software y por

qué son útiles.

El grueso del capítulo lo forma la presentación de algunos de los patrones de diseño más utilizados en el desarrollo de proyectos informáticos. Los patrones de diseño no están ligados a ningún lenguaje de programación en particular, son *directrices* que nos pueden ayudar en la escritura de código.

16.1. Principios de POO

A lo largo de este libro se ha intentado presentar no sólo el lenguaje de programación Java, y las herramientas de ayuda en el desarrollo de proyectos, si no también las buenas prácticas a seguir en la codificación y en el diseño de nuestras aplicaciones. Las buenas prácticas aplicadas al diseño de software orientado a objetos constituyen sus principios, unas normas de carácter general que conviene seguir en la construcción de software. Siguiendo las expuestas en las referencias [8] y [9] y de modo resumido y son:

- Encapsular lo que varía.
- Favorecer la composición frente a la herencia.
- Programar orientado a la interface no a la implementación.
- Evitar el acoplamiento entre clases.
- Reducir la responsabilidad de cada clase.

Los principios SOLID son otro grupo de principios a tener en cuenta en el diseño de software. Estos principios establecen:

Single responsability Una clase debe tener una única responsabilidad que justifique su existencia.

Open close principle La definición de una clase debe ser abierta para su extensión pero cerrada para su modificación.

Liskov substitution Siempre debe ser posible sustituir una clase padre por otra hija sin que cambie el comportamiento de la aplicación.

Interface segregation Una clase sólo debe implementar un interface si es necesario que ofrezca todos los métodos que declara el interface, y no sólo unos cuantos.

Dependency inversion Las clases no deben crear instancias de otras clases con las que trabajen, la dependencia de una clase con respecto de otra debe inyectarse desde fuera de la clase.

Estos principios generales pueden concretarse, a veces, es soluciones bien conocidas a problemas recurrentes en el diseño del software. Y precisamente estas soluciones son lo que se conoce como *Patrones de diseño*.

16.2. ¿Qué son los patrones de diseño?

Es usual que, durante el desarrollo de un proyecto informático, nos encontremos de modo recurrente con el mismo tipo de problemas. Por ejemplo cómo garantizar la existencia de una única instancia para poder acceder a un determinado dispositivo. O cómo estructurar una aplicación basada en un iterface gráfico para que me permita múltiples representaciones de los mismos datos. En este último caso ya has visto un patrón de diseño muy potente, MVC, en el capítulo 11.

Los patrones de diseño son soluciones bien conocidas y ampliamente empleadas para resolver problemas recurrentes en el diseño de aplicaciones informáticas. Cada uno de los patrones de diseño tiene, o suele tener, un nombre estandarizado, lo que define un vocabulario común que facilita el intercambio de ideas, y una plantilla de aplicación que muestra cuales son sus componentes y cómo se relacionan entre si.

16.3. ¿Qué es el acoplamiento entre clases y por qué hay que evitarlo?

Cuando al utilizar una clase desde el código de otra, la clase cliente conoce detalles de la implementación de la clase que utiliza decimos que las dos clases están fuertemente acopladas. El acoplamiento muchas veces implica que al cambiar la implementación de una clase, las clases cliente fuertemente acopladas con ella dejan de funcionar, deben ser modificadas para reflejar los cambios en la clase inicial. Esta coyuntura finalmente desemboca en una serie de modificaciones en cascada a lo largo y ancho del código de la aplicación.

Lo que hay que evitar, ante todo, es que una clase dependa de los detalles de implementación de otra para que pueda utilizarla. Y este es el principio básico que encontramos en todos los patrones de diseño, la independencia de la implementación concreta entre clases.

16.4. Grupos de patrones de diseño

Los patrones de diseño se agrupan por su cometido, así nos encontramos con patrones de diseño de creación (*Singleton, Factory method, Abstract factory*), de comportamiento (*Strategy, Observer*) y estructurales (*Decorator*) entre los más conocidos sin ser una lista exhaustiva. Por cuestión de espacio presentamos los indicados entre paréntesis, dejando el resto para su consulta en la bibliografía.

16.5. El patrón de diseño *Singleton*

Aparentemente este es un patrón de diseño muy sencillo que acaba teniendo una implementación sofisticada cuando se utiliza en ambientes de programación concurrente.

16.5.1. Situación que intenta resolver

El patrón de diseño *Singleton* garantiza que sólo existe una instancia de una determinada clase. La clase no se instancia con el operador `new` si no a través de la llamada a un método que siempre devuelve la misma instancia y única instancia de la clase.

Este patrón de diseño es útil cuando queremos garantiza la unicidad de una instancia, por ejemplo nuestra aplicación sólo conoce una instancia de la clase que accede a una impresora, o al sistema de ficheros.

16.5.2. Ejemplo de implementación

En el listado 16.1 se muestra una implementación de este patrón de diseño. El las líneas 4-6 definimos como `private` el constructor por defecto de la clase, de esto modo prohibimos la creación de instancias de esta clase con el operador `new`. Por otro lado, al existir únicamente el constructor por defecto con acceso `private` no se puede extender la clase. En este caso, el modificador `final` no es necesario, pero sirve para documentar la clase.

En la línea 3, definimos una referencia a la propia clase, que será la que devolvamos cada vez que se pida a través de la llamada al método *getInstancia()* definido entra las líneas 8-12.

```
1 public class Singleton {
2  private Singleton instancia = null;
3
4  private Singleton() {
5   super();
6  }
7
8  public Singleton getInstancia() {
9   if(instancia == null)
10    instancia = new Singleton();
11   return instancia;
12  }
13 }
```

Listado 16.1: Implementación sencilla del patrón *Singleton*

Como ves, recuperamos la instancia llamando al método `getInstancia()` y no con el operador `new`. Esta implementación es completamente funcional si nuestra aplicación no utiliza hilos, pero en caso de utilizarlos podemos encontrarnos con problemas al usar esta sencilla implementación. Veamos cual es el problema que puede aparecer, antes de ello, recordemos que la intención de este patrón de diseño es que únicamente exista una instancia de la clase *Singleton*. Supongamos ahora que se están ejecutando simultáneamente dos hilos que quieren recuperar una instancia de la clase *Singleton*, y que por simplicidad sólo contamos con un procesador (o un procesador con un único núcleo). Supongamos que uno de los hilos llama al método `getInstancia()`, que comprueba la condición de la línea 9 y que se evalúa a `false`, y que justo después de evaluar la condición y antes de crear la instancia en la línea 10, se cede la ejecución al segundo hilo, quien también evalúa la condición de la línea 9 obteniendo **false** (ya que la instancia no fue creada por el primer hilo), y que, ahora sí, crea una instancia de la clase *Singleton*. Cuando el hilo que está a la espera continúe su ejecución donde quedó (justo después de comprobar que no había ninguna instancia de *Singleton* creada), creará una nueva instancia de la clase *Singleton*

ya que al no volver a comprobar la condición para este hilo sigue siendo válido que no existe ninguna instancia. Como resultado final nos encontraremos con dos instancias de la misma clase, justo lo que no deseábamos que ocurriera.

La solución a este problema pasa por sincronizar el bloque de creación de la instancia de *Singleton* tal y como se muestra en el Listado 16.2. En este caso, inmediatamente después de comprobado si ya hay una instancia creada, escribimos un bloque sincronizado, dentro del cual lo primero que volvemos a comprobar es si la instancia sigue sin estar creada, si no lo está la creamos dentro del bloque sincronizado garantizando que ningún otro hilo entrará en este bloque si ya está en posesión de otro hilo. La doble comprobación es para evitar que no ocurra lo mismo que en el caso anterior, que justo después de comprobarla y antes de entrar en el bloque sincronizado otro hilo gane la carrera y ejecute el bloque sincronizado mientras el primero espera a seguir con la ejecución.

```
1 public class SingletonConcurrente {
2  private SingletonConcurrente instancia = null;
3
4  private SingletonConcurrente() {
5   super();
6  }
7
8  public SingletonConcurrente getInstancia() {
9   if(instancia == null)
10   synchronized (SingletonConcurrente.class) {
11    if(instancia == null)
12     instancia = new SingletonConcurrente();
13   }
14  return instancia;
15 }
16 }
```

Listado 16.2: Implementación sencilla del patrón *Singleton*

¿Por qué no hacemos una única comprobación dentro del bloque sincronizado, ya que en este momento garantizamos que sólo hay un hilo ejecutándolo? Para evitar el sobrecoste que implica la ejecución de bloques sincronizados. Si hacemos una primera comprobación fuera del bloque sincronizado y obtenemos **false** nunca entraremos en el bloque sincronizado y no caeremos en el sobrecoste temporal que esto implica. Si eliminamos la comprobación fuera del bloque, siempre tendremos que pugnar por el cerrojo del bloque sincronizado con el consiguiente coste en tiempo de ejecución que esto supone.

16.6. El patrón de diseño *Factory Method*

Factory Method es otro patrón de diseño dentro del grupo de patrones de diseño de creación. Este patrón de diseño es relativamente sencillo y las ventajas que presenta su uso son muchas.

16.6.1. Situación que intenta resolver

Tal y como hemos visto en la introducción de este capítulo, el desacoplamiento entre clases muchas veces pasa porque una clase cliente no cree una instancia de otra clase con la que quiera trabajar. Supongamos, por ejemplo, que estamos creando una aplicación que representa una fábrica de *Vehículos* pudiendo ser estos *Coches, Motos* y *Camiones*. Si cada vez que necesitamos una instancia

de un tipo concreto usamos el operador `new`, corremos el riesgo de que la implementación de las clases concretas *Coche, Moto, Camión* cambie y nuestro código deje de funcionar.

Una manera de desacoplar la clase cliente, la que quiere recuperar instancias de *Vehículos* concretos, y las instancias concretas de *Coche, Moto, Camión* es definir una nueva clase encargada de crear las instancias de las clases concretas y devolver las referencias no al tipo concreto si no a un **interface** o clase abstracta.

16.6.2. Ejemplo de implementación

Primero escribimos un `interface` del Listado 16.3 que es el tipo de datos abstracto para toda clase de *Vehiculos*, con independencia de si son *Coches, Motos* o *Camiones*. Este `interface` cuenta con constantes que identifican a los distintos tipos de vehículos que se pueden crear.

```
1 public interface Vehiculo {
2  static final int COCHE = 1;
3  static final int MOTO = 2;
4  static final int CAMION = 3;
5  String getDescripcion();
6 }
```

Listado 16.3: El tipo de datos abstracto *Vehiculo*

En los Listados 16.4, 16.5 y 16.6 aparecen las clases concretas para cada uno de los tres tipos de *Vehículos* que la fábrica puede crear.

```
1 public class Coche implements Vehiculo {
2  @Override
3  public String getDescripcion() {
4   return "Soy un coche";
5  }
6 }
```

Listado 16.4: Clase concreta que representa un *Coche*

```
1 public class Moto implements Vehiculo {
2  @Override
3  public String getDescripcion() {
4   return "Soy una moto";
5  }
6 }
```

Listado 16.5: Clase concreta que representa una *Moto*

```
1 public class Camion implements Vehiculo {
2  @Override
3  public String getDescripcion() {
4   return "Soy un camión";
5  }
6 }
```

Listado 16.6: Clase concreta que representa un *Camion*

Cada una de las clases anteriores da una implementación concreta para el método *getDescripcion()*.

El Listado 16.7 muestra la implementación de la fábrica de vehículos. Esta clase posee un único método estático que recibe el tipo del vehículo que deseamos obtener. La fábrica crea la instancia del tipo concreto y la devuelve como una referencia al tipo abstracto *Vehículo*.

```
1 public class FabricaVehiculos {
2  public static Vehiculo creaVehiculo(int tipo) {
3   Vehiculo vehiculo;
4
5   switch (tipo) {
6   case Vehiculo.COCHE:
7    vehiculo = new Coche();
8    break;
9
10  case Vehiculo.MOTO:
11   vehiculo = new Moto();
12   break;
13
14  case Vehiculo.CAMION:
15   vehiculo = new Camion();
16   break;
17
18  default:
19   vehiculo = new Coche();
20   break;
21  }
22
23  return vehiculo;
24 }
25 }
```

Listado 16.7: La fábrica de vehículos

Para finalizar, veamos cómo un cliente concreto trabaja con esta fábrica de vehículos. El cliente se muestra en el Listado 16.8. Lo interesante de este cliente es que no crea en ningún momento una instancia concreta de ninguna clase, si no que delega la creación de instancias concretas en la clase *FabricaVehiculos*. Si la implementación de una clase concreta cambia, el cliente no lo percibe. Si la fábrica de vehículos incorpora nuevos vehículos, el cliente puede utilizarlos, de nuevo, sin conocer la implementación concreta.

```
1 public class Cliente {
2  public static void main(String[] args) {
3   Vehiculo vehiculo = FabricaVehiculos.creaVehiculo(Vehiculo.COCHE);
4   System.out.println(vehiculo.getDescripcion());
5   vehiculo = FabricaVehiculos.creaVehiculo(Vehiculo.CAMION);
6   System.out.println(vehiculo.getDescripcion());
7   vehiculo = FabricaVehiculos.creaVehiculo(Vehiculo.MOTO);
8   System.out.println(vehiculo.getDescripcion());
9  }
10 }
```

Listado 16.8: Un cliente de la fábrica de vehículos

Como ves, este patrón de diseño es muy útil cuando se necesita crear clases concretas de un mismo tipo de datos abstracto, que en el ejemplo mostrado era el `interface Vehiculo`. Podríamos decir que siempre estamos creando instancias del mismo tipo de datos. Veamos un nuevo patrón de diseño que de algún modo amplia el patrón de diseño *Factory Method*, permitiéndonos la creación de *familias* de tipos de datos en vez de un único tipo de datos.

16.7. El patrón de diseño *Abstract Factory*

Este nuevo patrón de diseño también pertenece al grupo de los patrones de diseño de creación, pero esta vez no crea un único tipo de datos si no que crea familias de tipos de datos.

16.7.1. Situación que intenta resolver

Siguiendo con el ejemplo de la fábrica de vehículos, imagina esta vez que se intenta describir fábricas de vehículos genéricas. Dos ejemplos de fábricas de vehículos concretas pueden ser una fábrica de vehículos europea y otra fábrica de vehículos japonesa. Ambas fábricas de vehículos producen diferentes tipos de vehículos, que podemos restringir a *Turismos, Berlinas* y *Deportivos*. Lo que distingue un vehículo concreto, por ejemplo un *Turismo* creado en una fábrica europea o en una japonesa no es el proceso de construcción, ya que ambos vehículos tienen motor, chasis y ruedas. Lo que distingue a un *Turismo* europeo de uno japones es que una fábrica europea de vehículos utiliza componentes europeos (ruedas europeas, chasis europeos, motores europeos), mientras que una fábrica japonesa utiliza componentes japoneses (ruedas japonesas, chasis japoneses, motores japoneses). Resumiendo, misma gama de productos (*Vehículos*) pero construidos con componentes distintos (*Motor, Chasis* y *Ruedas*).

La diferencia con el patrón de diseño *Factory Mehod* (*Vehiculo*) es que en este se crea un sólo producto, mientras que en *Abstract Factory* se crea una gama de productos (*Motor, Chasis* y *Ruedas*).

16.7.2. Ejemplo de implementación

Para ver un ejemplo concreto vamos a empezar describiendo las entidades que forman parte del problema. En una fábrica, tanto si es europea como japonesa se construyen tres modelos de vehículos:

- *Turismos.*
- *Berlinas.*
- *Deportivos.*

Vamos a suponer un *Turismo* está formado únicamente por un *Chasis* y un *Motor*. Una *Berlina* está formada por un *Chasis, Motor* y *Ruedas*. Y finalmente, un *Deportivo* está formado por *Chasis, Motor, Ruedas* y *Extras*. En los Listados 16.9 a 16.12 se muestra el código para los vehículos. En la `interface` *Vehiculo*, hemos añadido un método que permite obtener una descripción de cada uno de los *Vehiculos public void descripcion()*.

```
1 public abstract class Vehiculo {
2 // protected Rueda ruedas;
3 // protected Chasis chasis;
4 // protected Motor motor;
5 // protected Extras extras;
6  public abstract void descripcion();
7 }
```

Listado 16.9: Un *Vehículo* como abstracción de los tres modelos que puede construir una fábrica *Turismos, Berlinas* y *Deportivos*

```
1 public class Turismo extends Vehiculo {
2  private Chasis chasis;
3  private Motor motor;
4
5  public Turismo(FabricaComponentes fabricaComponentes) {
6   chasis = fabricaComponentes.creaChasis();
7   motor = fabricaComponentes.creaMotor();
8  }
9
10 @Override
11 public void descripcion() {
12  System.out.println("--- Descripción de un TURISMO ---");
13  chasis.descripcion();
14  motor.descripcion();
15  System.out.println("-----------------------------------");
16 }
17 }
```

Listado 16.10: Un *Turismo* es una especialización de un *Vehiculo*

```
1 public class Berlina extends Vehiculo {
2  private Chasis chasis;
3  private Motor motor;
4  private Rueda ruedas;
5
6  public Berlina(FabricaComponentes fabricaComponentes) {
7   chasis = fabricaComponentes.creaChasis();
8   motor = fabricaComponentes.creaMotor();
9   ruedas = fabricaComponentes.creaRuedas();
10 }
11 @Override
12 public void descripcion() {
13  System.out.println("--- Descripción de una BERLINA ---");
14  chasis.descripcion();
15  motor.descripcion();
16  ruedas.descripcion();
17  System.out.println("-----------------------------------");
18 }
19
20 }
```

Listado 16.11: Una *Berlina* es una especialización de un *Vehiculo*

```
1 public class Deportivo extends Vehiculo {
2  private Chasis chasis;
3  private Extras extras;
4  private Motor motor;
5  private Rueda ruedas;
6
7  public Deportivo(FabricaComponentes fabricaComponentes) {
8   ruedas = fabricaComponentes.creaRuedas();
9   chasis = fabricaComponentes.creaChasis();
10  motor = fabricaComponentes.creaMotor();
11  extras = fabricaComponentes.creaExtras();
12 }
13
14 @Override
15 public void descripcion() {
16  System.out.println("--- Descripción de un DEPORTIVO ---");
17  ruedas.descripcion();
18  chasis.descripcion();
19  motor.descripcion();
20  extras.descripcion();
21  System.out.println("-----------------------------------");
22 }
23
24 }
```

Listado 16.12: Un *Deportivo* es una especialización de un *Vehiculo*

Vamos ahora a detallar los componentes que forman parte de cada uno de los diferentes *Vehiculos*. Si el *Vehículo* es japones su *Chasis* es siempre ligero y de aluminio, si es europeo su *Chasis* es siempre reforzado. El *Motor* de un *Vehiculo* japones es siempre de muy bajo consumo y bajas emisiones de CO2; si es europeo, el *Motor* es de alto rendimiento. Las *Ruedas* de un *Vehículo* japones son siempre de muy larga duración; mientras que los *Vehículos* europeos tienen ruedas de perfil bajo. Y finalmente, los *Vehículos* japoneses tienen extras de tipo deportivo japones y los europeos de tipo deportivo. El los listados 16.13 a 16.24 se muestra cada uno de los **interface** y las especializaciones de cada componente que puede formar parte de un *Vehículo*.

```
1 public interface Chasis {
2  void descripcion();
3 }
```

Listado 16.13: Un *Chasis* como abstracción.

```
1 public class ChasisLigero implements Chasis {
2  @Override
3  public void descripcion() {
4   System.out.println("Chasis ligero de aluminio.");
5  }
6 }
```

Listado 16.14: Un *ChasisLigero* es una especialización de un *Chasis* que será utilizado sólo en la construcción de *Vehículos* japoneses.

```
1 public class ChasisReforzado implements Chasis {
2  @Override
3  public void descripcion() {
4   System.out.println("Chasis reforzado");
5  }
6 }
```

Listado 16.15: Un *ChasisReforzado* es una especialización de un *Chasis* que será utilizado sólo en la construcción de *Vehículos* europeos

```
1 public interface Extras {
2  void descripcion();
3 }
```

Listado 16.16: Unos *Extras* como abstracción.

```
1 public class ExtrasDeportivosEstiloJapones implements Extras {
2
3  @Override
4  public void descripcion() {
5   System.out.println("Extras deportivos.... pero al estilo japones.");
6  }
7
8 }
```

Listado 16.17: Unos *ExtrasDeportivosEstiloJapones* es una especialización de unos *Extras* que serán utilizados sólo en la construcción de *Vehículos* japoneses

```
1 public class ExtrasDeportivos implements Extras {
2  @Override
3  public void descripcion() {
4   System.out.println("Extras deportivos.");
5  }
6 }
```

Listado 16.18: Unos *ExtrasDeportivos* es una especialización de unos *Extras* que serán utilizados sólo en la construcción de *Vehículos* europeos

```
1 public interface Motor {
2  void descripcion();
3 }
```

Listado 16.19: Un *Motor* como abstracción.

```
1 public class MotorBajoConsumo implements Motor {
2
3  @Override
4  public void descripcion() {
5   System.out.println("Motor de muy bajo consumo y bajas emisiones de CO2
        .");
6  }
7
8 }
```

Listado 16.20: Un *MotorBajoCosumo* es una especialización de un *Motor* que será utilizado sólo en la construcción de *Vehículos* japoneses

```
1 public class MotorAltoRendimiento implements Motor {
2  @Override
3  public void descripcion() {
4   System.out.println("Motor de alto rendimiento.");
5  }
6 }
```

Listado 16.21: Un *MotorAltoRendimienot* es una especialización de un *Motor* que será utilizado sólo en la construcción de *Vehículos* europeos

```
1 public interface Rueda {
2  void descripcion();
3 }
```

Listado 16.22: Una *Rueda* como abstracción.

```
1 public class RuedaLargaDuracion implements Rueda {
2
3  @Override
4  public void descripcion() {
5   System.out.println("Ruedas de larga duración");
6  }
7
```

```
8 }
```

Listado 16.23: Una *RuedaLargaDuracion* es una especialización de una *Rueda* que será utilizada sólo en la construcción de *Vehículos* japoneses

```
1 public class RuedaPerfilBajo implements Rueda {
2
3  @Override
4  public void descripcion() {
5   System.out.println("Rueda de perfil bajo.");
6  }
7
8 }
```

Listado 16.24: Una *RuedaPerfilBajo* es una especialización de una *Rueda* que será utilizada sólo en la construcción de *Vehículos* europeos

Es el momento de construir las fábricas de componentes. Las fábricas de componentes son las encargadas de crear cada uno de los componentes que forman un *Vehículo* dependiendo de si estamos en Europa o Japón. Para implementarlas, vamos a utilizar el patrón de diseño *Factory Method*. Los listados 16.25 a 16.27 muestran el código para cada una de las fábricas de componentes.

```
1 public interface Rueda {
2  void descripcion();
3 }
```

Listado 16.25: Una *FabricaComponentes* como abstracción.

```
1 public class FabricaComponentesJaponesa implements FabricaComponentes {
2
3  @Override
4  public Rueda creaRuedas() {
5   return new RuedaLargaDuracion();
6  }
7
8  @Override
9  public Chasis creaChasis() {
10  return new ChasisLigero();
11 }
12
13 @Override
14 public Motor creaMotor() {
15  return new MotorBajoConsumo();
16 }
17
18 @Override
19 public Extras creaExtras() {
20  return new ExtrasDeportivosEstiloJapones();
21 }
22
23 }
```

Listado 16.26: Una *FabricaComponentesJaponesa* es una especialización de una *FabricaComponentes* que crea los distintos componentes para cada tipo de *Vehículos* japoneses

```
1 public class FabricaComponentesEuropea implements FabricaComponentes {
2
3  @Override
```

```
4  public Rueda creaRuedas() {
5   return new RuedaPerfilBajo();
6  }
7
8  @Override
9  public Chasis creaChasis() {
10  return new ChasisReforzado();
11 }
12
13 @Override
14 public Motor creaMotor() {
15  return new MotorAltoRendimiento();
16 }
17
18 @Override
19 public Extras creaExtras() {
20  return new ExtrasDeportivos();
21 }
22
23 }
```

Listado 16.27: Una *FabricaComponentesEuropea* es una especialización de una *FabricaComponentes* que crea los distintos componentes para cada tipo de *Vehículos* europeos

Ya nos encontramos en la recta final para acabar de ver la implementación de este patrón de diseño. Vemos por último, la implementación de cada una de las fábricas de vehículos, la japonesa y la europea en los listados 16.28 al 16.30.

```
1 public interface FabricaVehiculos {
2  Vehiculo creaVehiculo(TipoVehiculo tipoVehiculo);
3 }
```

Listado 16.28: Una *FabricaVehiculos* como abstracción.

```
1 public class FabricaVehiculosJaponesa implements FabricaVehiculos {
2
3  @Override
4  public Vehiculo creaVehiculo(TipoVehiculo tipoVehiculo) {
5   FabricaComponentes fabricaComponentes = new FabricaComponentesJaponesa
        ();
6   Vehiculo vehiculo;
7
8   switch (tipoVehiculo) {
9    case TURISMO:
10    System.out.println("Fabricando un turismo japones...");
11    vehiculo = new Turismo(fabricaComponentes);
12    break;
13
14   case BERLINA:
15    System.out.println("Fabricando una berlina japonesa...");
16    vehiculo = new Berlina(fabricaComponentes);
17    break;
18
19   case DEPORTIVO:
20    System.out.println("Fabricando un deportivo japones...");
21    vehiculo = new Deportivo(fabricaComponentes);
22    break;
23
24   default:
25    System.out.println("Fabricando un turismo japones...");
26    vehiculo = new Turismo(fabricaComponentes);
27    break;
28  }
29
30  return vehiculo;
31 }
32
```

```
33 }
```

Listado 16.29: Una *FabricaVehiculosJaponesa* es una especialización de una *FabricaVehiculos* que crea los distintos tipos de *Vehículos* japoneses

```
1  public class FabricaVehiculosEuropea implements FabricaVehiculos {
2
3   @Override
4   public Vehiculo creaVehiculo(TipoVehiculo tipoVehiculo) {
5    FabricaComponentes fabricaComponentes = new FabricaComponentesEuropea
         ();
6    Vehiculo vehiculo = null;
7
8    switch (tipoVehiculo) {
9     case TURISMO:
10     System.out.println("Fabricando un turismo europeo...");
11     vehiculo = new Turismo(fabricaComponentes);
12     break;
13
14    case BERLINA:
15     System.out.println("Fabricando una berlina europea...");
16     vehiculo = new Berlina(fabricaComponentes);
17     break;
18
19    case DEPORTIVO:
20     System.out.println("Fabricando un deportivo europeo...");
21     vehiculo = new Deportivo(fabricaComponentes);
22     break;
23
24    default :
25     System.out.println("Fabricando un turismo europeo...");
26     vehiculo = new Turismo(fabricaComponentes);
27     break;
28   }
29
30   return vehiculo;
31  }
32
33 }
```

Listado 16.30: Una *FabricaVehiculosEuropea* es una especialización de una *FabricaVehiculos* que crea los distintos tipos de *Vehículos* europeos

El detalle de interés en los listados 16.29 y 16.30 está en la línea 5 de ambos listados, es la fábrica de vehículos japonesa la que se provee de componentes japoneses y la fábrica europea la que se provee de componentes europeos.

Otro ejemplo que comúnmente se utiliza como ejemplo del patrón de diseño *Abstract Factory* es la programación de un framework para componer ventanas en distintos sistemas operativos. Las ventanas que queremos crear están formadas por el mismo tipo de componentes (botones, listas, combo-box, etcétera), pero quien proporciona los componentes es el sistema operativo sobre el que se vaya a ejecutar la aplicación, de modo que, por ejemplo, se crean botones distintos si la fábrica de componentes es, digamos por caso, Mac OS X que si la fábrica es Linux.

16.8. El patrón de diseño *Strategy*

Este patrón es el primero, de los dos que vamos a ver, del grupo de patrones de comportamiento.

16.8.1. Situación que intenta resolver

Cuando intentamos resolver un problema concreto, en la mayoría de las ocasiones existe más de un algoritmo para encontrar la solución. Piensa por ejemplo en los algoritmos de ordenación, el objetivo de todos ellos es el mismo, ordenar una secuencia de elemento teniendo en cuenta cierta función de comparación entre ellos; tomando un caso sencillo podemos concretar que los elementos son números naturales. Para ordenar una secuencia de números naturales podemos utilizar el algoritmo de intercambio, el algoritmo de la burbuja o el algoritmo quicksort.

La codificación más flexible será aquella que permita intercambiar el algoritmo de ordenación con el mínimo, o nulo, impacto sobre la aplicación. El patrón *Strategy* nos dice que debemos encapsular cada algoritmo dentro de una clase y hacer estas clases intercambiables para el cliente que las use de modo transparente.

16.8.2. Ejemplo de implementación

Como ejemplo de implementación, supongamos que nuestro algoritmo cuenta números naturales. Existirán casos en los que nos interese contar de modo ascendente y otros casos en los que nos interese contar de modo descendente. Incluso puede que a veces nos interese contar de modo ascendente sólo números pares o sólo números impares. Fíjate que en los cuatros casos anteriores el algoritmo es el mismo: Contar números.

Para hacer los algoritmos intercambiables, todos ellos van a implementar el **interface** que se muestra en el listado 16.31. Las cuatro implementaciones de contadores particulares se muestran en los Listados 16.32 al 16.35.

```
1 public interface Contador {
2  static final int ASCENDENTE = 1;
3  static final int DESCENDENTE = 2;
4  static final int PARES = 3;
5  static final int IMPARES = 4;
6
7  String cuenta();
8 }
```

Listado 16.31: Este es el comportamiento común a todos los algoritmo: Contar

```
1 public class ContadorAscendente implements Contador {
2  @Override
3  public String cuenta() {
4   String cuenta = "";
5
6   for(int i = 0; i < 10; i++)
7    cuenta += i + ", ";
8
9   return cuenta;
10  }
11 }
```

Listado 16.32: Implementación de un contador ascendente.

```
1 public class ContadorDescendente implements Contador {
2
3  @Override
```

```
4  public String cuenta() {
5   String cuenta = "";
6
7   for(int i = 9; i >= 0; i--)
8    cuenta += i + ", ";
9
10  return cuenta;
11 }
12
13 }
```

Listado 16.33: Implementación de un contador descendente.

```
1 public class ContadorImpares implements Contador {
2
3  @Override
4  public String cuenta() {
5   String cuenta = "";
6
7   for(int i = 1; i < 10; i += 2)
8    cuenta += i + ", ";
9
10  return cuenta;
11 }
12
13 }
```

Listado 16.34: Implementación de un contador de números impares.

```
1 public class ContadorPares implements Contador {
2
3  @Override
4  public String cuenta() {
5   String cuenta = "";
6
7   for(int i = 0; i < 10; i += 2)
8    cuenta += i + ", ";
9
10  return cuenta;
11 }
12
13 }
```

Listado 16.35: Implementación de un contador de números pares.

Ahora escribamos un cliente que pueda utilizar estos algoritmos de modo intercambiable, tal y como muestra el Listado 16.36:

```
1 import contar.Contador;
2
3 public class Cliente {
4  private Contador contador;
5
6  public void cuenta() {
7   contador.cuenta();
8  }
9
10 public void setContador(Contador contador) {
11  this.contador = contador;
12 }
13 }
```

Listado 16.36: Un cliente que puede utilizar cualquiera de los anteriores algoritmo para contar.

Esta clase *Cliente* tiene una característica interesante para poder utilizar cualquier tipo de algoritmo para contar, el algoritmo particular se le *inyecta* a través del método *public void setContador(Contador contador)*. A esta técnica de relación entre clase se le llama *Inyección de Dependencias* o *Inversión de Control*.

Finalmente, y por completar el ejemplo, el Listado 16.37 muestra cómo utilizar la clase *Cliente* inyectándole los cuatro tipos de contadores.

```
1 import contar.ContadorAscendente;
2 import contar.ContadorDescendente;
3 import contar.ContadorImpares;
4 import contar.ContadorPares;
5
6 public class Principal {
7  private void ejecuta() {
8   Cliente cliente = new Cliente();
9   cliente.setContador(new ContadorAscendente());
10  cliente.cuenta();
11  cliente.setContador(new ContadorDescendente());
12  cliente.cuenta();
13  cliente.setContador(new ContadorPares());
14  cliente.cuenta();
15  cliente.setContador(new ContadorImpares());
16  cliente.cuenta();
17 }
18
19 public static void main(String[] args) {
20  new Principal().ejecuta();
21 }
22 }
```

Listado 16.37: Podemos cambiar dinámicamente el tipo de contador que utiliza la clase *Cliente*.

16.9. El patrón de diseño *Observer*

Observer es otro patrón de diseño de comportamiento. Lo hemos utilizado ampliamente en el capítulo 11 dedicado a la programación de interfaces gráficos de usuario.

16.9.1. Situación que intenta resolver

Cuando intentamos monitorizar el estado, por ejemplo, de un atributo en una determinada instancia, la opción directa, y altamente ineficiente, es interrogar cada cierto tiempo por el estado de ese atributo. La ineficiencia de este método se debe a que aunque no cambie el estado del atributo, estamos consumiendo tiempo en averiguar si lo ha hecho.

Es más sencillo que sea la propia instancia quien nos avise de que su atributo ha cambiado de valor, no es el cliente el que consulta, si no que espera a que la instancia monitorizada le avise del cambio. Este comportamiento se conoce como *Principio de Hollywood* que se puede resumir en la frase *No me llames, ya te llamaré yo.*

Recuerda cómo programábamos, en Swing, los escuchadores para un componente gráfico, por ejemplo un botón. Lo que hacíamos era escribir un escuchador que era notificado cada vez que ocurría un evento sobre el botón. Y este es, precisamente, el patrón de diseño *Observer*.

16.9.2. Ejemplo de implementación

Como ejemplo, escribamos una pequeña novela de espías basada en la guerra fría, cuando aún existía el KGB. El KGB tiene espías que informan de todos sus movimientos a sus superiores. Un mismo espía puede trabajar para más de un superior, el mismo espía puede estar trabajando al mismo tiempo para su jefe directo y para el ministerio del interior. Quien genera los informes (eventos) es el espía, el espía está siendo observado; y quien recibe los informes son sus superiores que actúan como observadores. Para que los superiores de un espía reciban mensajes establecen el mecanismo de que el espía haga una llamada al método *informaObservadores(String accion)* de sus jefes. Y un espía conoce en todo momento quienes son los jefes para los que trabaja, los mantiene en un lista donde puede añadir nuevos jefes para informar cuando inicia una misión, o eliminarlos cuando acaba su misión.

Los Listados 16.38 y 16.39 muestran el código referente al espía.

```
1 public interface Observado {
2  public void addObservador(Observador observador);
3  public void removeObservador(Observador observador);
4  public void informaObservadores(String accion);
5 }
```

Listado 16.38: Esta `interface` es una abstracción del comportamiento común a todos los espías.

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class EspiaKGB implements Observado {
5  private String nombre;
6  private List<Observador>observadores = new ArrayList<Observador>();
7
8
9  public EspiaKGB(String nombre) {
10  super();
11  this.nombre = nombre;
12 }
13
14 @Override
15 public void addObservador(Observador observador) {
16  observadores.add(observador);
17 }
18
19 @Override
20 public void removeObservador(Observador observador) {
21  observadores.remove(observador);
22 }
23
24 @Override
25 public void informaObservadores(String accion) {
26  for(Observador observador: observadores)
27   observador.informe(nombre + ": " + accion);
28 }
29 }
```

Listado 16.39: Esta es la clase que representa a un espía.

Es a través del método *informaObservadores(String accion)* por el que el espía envía los informes a todos y cada uno de sus jefes. Los Listados 16.40 y 16.41 muestran el código relativo a los jefes.

```
1 public interface Observador {
2  public void informe(String evento);
3 }
```

Listado 16.40: Esta `interface` es una abstracción del comportamiento común a todos los jefes.

```
1 public class JefeEspiasKGB implements Observador {
2  private String nombre;
3
4  public JefeEspiasKGB() {
5   super();
6   nombre = "Anónimo";
7  }
8
9  public JefeEspiasKGB(String nombre) {
10  super();
11  this.nombre = nombre;
12 }
13
14 @Override
15 public void informe(String evento) {
16  System.out.println(nombre + " recibe de " + evento);
17 }
18
19 }
```

Listado 16.41: Esta es la clase que representa a un jefe de espía.

Y finalmente en el Listado 16.42 escribimos una sencillísima novela de espías.

```
1 public class PruebaEspias {
2  private void ejecuta() {
3   Observado espiaBoris = new EspiaKGB("Boris");
4   Observador jefeBorisKGB = new JefeEspiasKGB("Jefe de Boris");
5   espiaBoris.addObservador(jefeBorisKGB);
6   Observador ministroDefensaURSS = new JefeEspiasKGB("Ministerio del
        interior");
7   espiaBoris.addObservador(ministroDefensaURSS);
8   espiaBoris.informaObservadores("Estoy siguiendo al topo");
9  }
10
11 public static void main(String[] args) {
12  new PruebaEspias().ejecuta();
13 }
14 }
```

Listado 16.42: Cada vez que *Boris* realiza una acción informa a sus superiores.

Y este es el resultado de su ejecución:

```
Jefe de Boris recibe de Boris: Estoy siguiendo al topo
Ministerio del interior recibe de Boris: Estoy siguiendo al topo
```

16.10. El patrón de diseño *Decorator*

Este es el único patrón de diseño estructural que vamos a ver. Igual que el patrón *Observer*, este patrón es ampliamente utilizado Java, en este caso en el paquete java `java.io` de entrada/salida en Java.

16.10.1. Situación que intenta resolver

Hemos visto algunos ejemplos de este patrón en las clases de entrada/salida, cuando íbamos recubriendo una clase de partida con nuevas clases hasta alcanzar una clase con la funcionalidad que necesitábamos.

En algunos casos necesitamos añadir nuevas características a nuestras clases pero el uso directo de la herencia dispara exponencialmente el número de clases que tendríamos que implementar. Veámoslo con un ejemplo. Supongamos que estamos desarrollando una aplicación para un concesionario de coches. El coche básico de cada serie tiene un precio, y este precio se incrementa cuando el comprador va añadiendo nuevos extras al coche. Por ejemplo, sobre el coche básico podemos elegir pintura metalizada o aire acondicionado. Si cada extra estuviese codificado como una nueva clase que extiende a la clase que representa el coche básico deberíamos escribir una nueva clase para el coche básico con aire acondicionado, otra para el coche básico con pintura metalizada, y como no, una tercera clase hija para el caso en que algún cliente quiera añadir aire acondicionado y pintura metalizada al coche básico. Claramente el uso de la herencia no es buena idea en este caso.

Lo que necesitamos es que cada uno de los extras se añada sobre la clase base de manera independiente del resto de los extras, y fíjate que no por ello dejamos de tener un vehículo.

La idea del patrón de diseño *Decorator* es tomar una clase base e ir añadiéndole nuevas características sin utilizar exclusivamente la herencia.

16.10.2. Ejemplo de implementación

Como ejemplo de implementación vamos a utilizar una pequeña modificación del ejemplo expuesto en la sección anterior, un concesionario que quiere calcular el precio final de los vehículos que vende pudiendo ser estos coches o camiones. Al vehículo básico el comprador le puede añadir extras, tales como aire acondicionado o pintura metalizada.

Abstraigamos la idea de un *Vehículo* como una clase abstracta de la que *Coche* y *Camión* serán dos clases concretas, tal y como muestran los Listados 16.43 a 16.45.

```
1  public abstract class Vehiculo {
2   private String descripcion;
3   private float precio;
4
5   public Vehiculo() {
6    super();
7   }
8
9   public Vehiculo(String descripcion, float precio) {
10   super();
11   this.descripcion = descripcion;
12   this.precio = precio;
13  }
14
15  public String getDescripcion() {
16   return descripcion;
17  }
18
19
20  public float getPrecio() {
21   return precio;
22  }
```

```
23 }
```

Listado 16.43: Abstracción de un *Vehículo*.

```
1 public class Coche extends Vehiculo {
2  public Coche(String descripcion, float precio) {
3   super(descripcion, precio);
4  }
5 }
```

Listado 16.44: Un *Coche* como clase concreta que extiende a *Vehículo*.

```
1 public class Camion extends Vehiculo {
2  public Camion(String descripcion, float precio) {
3   super(descripcion, precio);
4  }
5 }
```

Listado 16.45: Un *Camión* como clase concreta que extiende a *Vehículo*.

La clase que decora a *Vehículo* es *VehiculoConExtras* y es ella quien resuelve de una manera elegante el problema, por una parte extiende a *Vehículo* ya que un *VehiculoConExtras* sigue siendo un *Vehículo*, y por otra parte contiene una referencia al *Vehículo* que decora para poder delegar la llamada a sus métodos. En los Listados 16.46 a 16.48 se muestran las clases que añaden extras a los *Vehículos* base.

```
1 public abstract class VehiculoConExtras extends Vehiculo {
2  protected Vehiculo vehiculo;
3
4  public VehiculoConExtras(Vehiculo vehiculo) {
5   super();
6   this.vehiculo = vehiculo;
7  }
8 }
```

Listado 16.46: Abstración de un *Vehículo* que añade equipamiento extra.

```
1 public class VehiculoConAireAcondicionado extends VehiculoConExtras {
2  public VehiculoConAireAcondicionado(Vehiculo vehiculo) {
3   super(vehiculo);
4  }
5
6  @Override
7  public String getDescripcion() {
8   return vehiculo.getDescripcion() + ", aire acondicionado";
9  }
10
11 @Override
12 public float getPrecio() {
13  return vehiculo.getPrecio() + 300.67f;
14 }
15 }
```

Listado 16.47: Un *Vehículo* que añade el extra aire acondicionado.

```
1 public class VehiculoConPinturaMetalizada extends VehiculoConExtras {
2  public VehiculoConPinturaMetalizada(Vehiculo vehiculo) {
3   super(vehiculo);
```

```
4  }
5
6  @Override
7  public String getDescripcion() {
8   return vehiculo.getDescripcion() + ", pintura metalizada";
9  }
10
11 @Override
12 public float getPrecio() {
13  return vehiculo.getPrecio() + 600.45f;
14 }
15 }
```

Listado 16.48: Un *Vehículo* que añade el extra pintura metalizada.

Y finalmente el Listado 16.49 muestra cómo utilizar las clases decoradoras para un par de *Vehículos*.

```
1 public class PruebaDecorator {
2  private void ejecuta() {
3   Vehiculo vehiculo = new Coche("Berlina", 20000);
4   vehiculo = new VehiculoConAireAcondicionado(vehiculo);
5   vehiculo = new VehiculoConPinturaMetalizada(vehiculo);
6   System.out.println("El precio de este coche es: " + vehiculo.getPrecio
        ());
7   System.out.println(vehiculo.getDescripcion());
8
9   vehiculo = new Camion("Transporte", 100000);
10  vehiculo = new VehiculoConAireAcondicionado(vehiculo);
11  System.out.println("El precio de este camión es: " + vehiculo.
        getPrecio());
12  System.out.println(vehiculo.getDescripcion());
13 }
14
15 public static void main(String[] args) {
16  new PruebaDecorator().ejecuta();
17 }
18 }
```

Listado 16.49: Ejemplo de creación de un par de *Vehículos* con algunos extras.

El resultado de la ejecución de este ejemplo es:

```
El precio de este coche es: 20901.12
Berlina, aire acondicionado, pintura metalizada
El precio de este camión es: 100300.67
Transporte, aire acondicionado
```

Tanto el precio como la descripción de cada vehículo se obtienen a partir de la clase base y las clases que van decorando a esta clase base.

Ejercicios.

1. Escribe una aplicación donde sea posible intercambiar de forma sencilla cada uno de los algoritmos de ordenación.

2. Escribe una aplicación para calcular el precio de un café. Al café se le puede añadir una pizca de leche, leche condesada o el toque de algún licor. Evidentemente, el precio final depende del número de *antojos* añadidos al café base.

Lecturas recomendadas.

- El libro de referencia para los patrones de diseño es el escrito por *The gang of four* del que existe traducción al español [8].
- De nuevo, los libro de la colección *Head first* son de una muy clara exposición, la manera de presentar los contenidos es muy didáctica y los ejemplos utilizados claros y representativos. En particular la referencia [9] sobre patrones de diseños es casi de obligada lectura.

Apéndice A

build.xml

```
<project name="ConversionTemperaturas" default="test">
 <!-- Directorio del codigo fuente -->
 <property name="src.dir" location="../excepciones"/>
 <!-- Directorio de clases compiladas -->
 <property name="build.dir" location="build"/>
 <!-- Subdirectorio de las clases compiladas del proyecto -->
 <property name="build.classes.dir" location="${build.dir}/classes"/>
 <!-- Directorio de las clases de prueba -->
 <property name="test.dir" location="../test/test"/>
 <!-- Subdirectorio de las clases compiladas de prueba -->
 <property name="test.classes.dir" location="${build.dir}/test-classes"/
     >
 <!-- Directorio de bibliotecas del proyecto -->
 <property name="lib" location="../lib"/>
 <!-- Directorio de informes -->
 <property name="reports.dir" location="reports"/>
 <!-- Directorio para los informes en formato texto -->
 <property name="reports.txt.dir" location="${reports.dir}/txt"/>
 <!-- Directorio para los informes en formato xml -->
 <property name="reports.xml.dir" location="${reports.dir}/xml"/>
 <!-- Directorio para los informes en formato html -->
 <property name="reports.html.dir" location="${reports.dir}/html"/>
 <!-- Directorio para la documentacion -->
 <property name="reports.javadoc" location="${reports.dir}/javadoc"/>
 <!-- Directorio para el fichero empaquetado -->
 <property name="dist.dir" location="dist" />
 <!-- Nombre del fichero empaquetado -->
 <property name="dist.name" value="ConversorTemperaturas.jar" />

 <property name="junit.dir" location="/Users/oscar/Oscar/Software/
     eclipseHeliosSE64/plugins/org.junit_4.8.1.v4_8_1_v20100427-1100"/>
 <path id="junit">
  <fileset dir="${junit.dir}" includes="*.jar"/>
  <fileset dir="/Users/oscar/Oscar/Software/eclipseHeliosSE64/plugins"
       includes="org.hamcrest.core_1.1.0.v20090501071000.jar"/>
  <filelist ></filelist>
 </path>

 <!-- Path para compilar las clases de prueba -->
 <path id="test.compile.classpath">
  <fileset dir="${lib}" includes="*.jar"/>
  <pathelement location="${build.classes.dir}"/>
 </path>

 <!-- Path para ejecutar las clases de prueba -->
 <path id="test.classpath">
  <path refid="test.compile.classpath"/>
  <pathelement path="${test.classes.dir}"/>
 </path>

```

```
48 <target name="clean" description="Limpia el proyecto">
49  <delete dir="${dist.dir}"/>
50  <delete dir="${build.dir}"/>
51  <delete dir="${reports.dir}"/>
52 </target>
53
54 <target name="compile" description="Compila el proyecto">
55  <mkdir dir="${build.classes.dir}"/>
56  <javac srcdir="${src.dir}"
57   destdir="${build.classes.dir}" />
58 </target>
59
60 <target name="compile-tests"
61  depends="compile"
62  description="Compila los tests.">
63  <mkdir dir="${test.classes.dir}"/>
64  <javac srcdir="${test.dir}"
65   destdir="${test.classes.dir}">
66   <classpath refid="test.compile.classpath"/>
67   <classpath refid="junit"/>
68  </javac>
69 </target>
70
71 <target name="test"
72  depends="compile-tests"
73  description="Ejecuta los tests unitarios">
74  <mkdir dir="${reports.dir}"/>
75  <mkdir dir="${reports.txt.dir}"/>
76  <mkdir dir="${reports.xml.dir}"/>
77  <junit printsummary="true"
78   haltonfailure="false"
79   failureproperty="test.failures">
80   <classpath refid="test.classpath"/>
81   <classpath refid="junit"/>
82   <formatter type="plain" />
83   <test name="test.AllTests"
84    todir="${reports.txt.dir}"/>
85  </junit>
86 </target>
87
88 <target name="test.xml"
89  depends="compile-tests"
90  description="Ejecuta los tests unitarios">
91  <mkdir dir="${reports.dir}"/>
92  <mkdir dir="${reports.xml.dir}"/>
93  <junit printsummary="true"
94   haltonfailure="false"
95   failureproperty="test.failures">
96   <classpath refid="test.classpath"/>
97   <classpath refid="junit"/>
98   <formatter type="xml" />
99   <batchtest todir="${reports.xml.dir}">
100   <fileset dir="${test.classes.dir}">
101    <include name="**/Test*.class"/>
102   </fileset>
103  </batchtest>
104  </junit>
105 </target>
106
107 <target name="test.reports"
108  depends="test"
109  description="Genera los informes de los tests en formato xml">
110  <junitreport todir="${reports.xml.dir}">
111   <fileset dir="${reports.xml.dir}">
112    <include name="TEST-*.xml"/>
113   </fileset>
114   <report format="frames"
115    todir="${reports.html.dir}"/>
116  </junitreport>
117  <fail if="test.failures"
118   message="Se han producido errores en los tests."/>
119 </target>
120
121 <target name="javadoc"
```

```
122   depends="compile"
123   description="Genera la documentacion del proyecto.">
124   <javadoc sourcepath="${src.dir}"
125    destdir="${reports.javadoc}"
126    author="true" version="true"
127    use="true" access="private"
128    linksource="true" encoding="ISO-8859-1"
129    windowtitle="${ant.project.name}">
130    <classpath>
131     <pathelement path="${test.classes.dir}"/>
132     <pathelement path="${build.classes.dir}"/>
133    </classpath>
134   </javadoc>
135  </target>
136
137  <target name="package"
138   depends="compile"
139   description="Genera el fichero jar" >
140   <mkdir dir="${dist.dir}"/>
141   <jar destfile="${dist.dir}/${dist.name}">
142    <fileset dir="${build.classes.dir}"/>
143    <manifest>
144     <attribute
145      name="Main-Class"
146      value="conversor.Principal"/>
147    </manifest>
148   </jar>
149  </target>
150
151  <target name="execute"
152   depends="package"
153   description="Ejecuta la aplicacion.">
154   <java
155    jar="${dist.dir}/${dist.name}"
156    fork="true"/>
157  </target>
158 </project>
```

Listado A.1: Fichero *Ant* para la construcción del proyecto de conversión de temperaturas

Apéndice B

Aplicación Hipoteca

Código fuente de la aplicación del cálculo de la cuota mensual de una hipoteca

```
1 package gui.hipoteca.modelo;
2
3 import gui.hipoteca.vista.Vista;
4
5 public interface Modelo {
6  public void setVista(Vista vista);
7  public void setDatos(double cantidad, int tiempo, double interes);
8  public double getCuota();
9 }
```

Listado B.1: `interface Modelo`

```
1 package gui.hipoteca.modelo;
2
3 import gui.hipoteca.vista.Vista;
4
5 public class ModeloImpl implements Modelo {
6  private Vista vista;
7  private double cantidad;
8  private int tiempo;
9  private double interes;
10  private double cuota;
11
12  public ModeloImpl() {
13   super();
14  }
15
16  @Override
17  public void setVista(Vista vista) {
18   this.vista = vista;
19  }
20
21  @Override
22  public synchronized void setDatos(double cantidad, int tiempo, double
        interes) {
23   this.cantidad = cantidad;
24   this.tiempo = tiempo;
25   this.interes = interes;
26   calculaCuota();
27   vista.cuotaActualizada();
28  }
29
30  private void calculaCuota() {
31   double n = interes/1200;
32   cuota = cantidad*n/(1-(1/Math.pow(1+n, 12*tiempo)));
```

```
33  }
34
35  @Override
36  public synchronized double getCuota() {
37   return cuota;
38  }
39 }
```

Listado B.2: Implementación del `interface Modelo`

```
1 package gui.hipoteca.vista;
2
3 import java.awt.Container;
4
5 import gui.hipoteca.controlador.Controlador;
6 import gui.hipoteca.modelo.Modelo;
7
8 public interface Vista {
9  public void setControlador(Controlador controlador);
10 public void setModelo(Modelo modelo);
11 public Container getContenedor();
12 public double getCantidad();
13 public int getTiempo();
14 public double getInteres();
15 public void cuotaActualizada();
16 }
```

Listado B.3: `interface Vista`

```
1 package gui.hipoteca.vista;
2
3 import gui.hipoteca.controlador.Controlador;
4 import gui.hipoteca.modelo.Modelo;
5
6 import java.awt.BorderLayout;
7 import java.awt.Container;
8 import java.awt.event.ActionEvent;
9 import java.awt.event.ActionListener;
10 import java.lang.reflect.InvocationTargetException;
11
12 import javax.swing.JButton;
13 import javax.swing.JLabel;
14 import javax.swing.JPanel;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class VistaImpl implements Vista {
19  private Modelo modelo;
20  private Controlador controlador;
21  // Componentes gráficos
22  private Container contenedor;
23  private JTextField jtfCantidad;
24  private JTextField jtfTiempo;
25  private JTextField jtfInteres;
26  private JLabel jlCuota;
27
28  public VistaImpl() {
29   super();
30   creaGUI();
31  }
32
33  private void creaGUI() {
34   try {
35    SwingUtilities.invokeAndWait(new Runnable() {
36     @Override
37     public void run() {
38      contenedor = new Container();
39      contenedor.setLayout(new BorderLayout());
40      JPanel jpDatos = new JPanel();
```

```
41      jpDatos.add(new JLabel("Cantidad: "));
42      jtfCantidad = new JTextField(8);
43      jpDatos.add(jtfCantidad);
44      jpDatos.add(new JLabel("Años: "));
45      jtfTiempo = new JTextField(3);
46      jpDatos.add(jtfTiempo);
47      jpDatos.add(new JLabel("Interes: "));
48      jtfInteres = new JTextField(5);
49      jpDatos.add(jtfInteres);
50      JButton jbCalcula = new JButton("Calcula");
51      jbCalcula.addActionListener(new Escuchador());
52      jpDatos.add(jbCalcula);
53      contenedor.add(jpDatos, BorderLayout.NORTH);
54      jlCuota = new JLabel("Cuota mensual:");
55      JPanel jpCuota = new JPanel();
56      jpCuota.add(jlCuota);
57      contenedor.add(jpCuota);
58     }
59    });
60   } catch (InterruptedException e) {
61    e.printStackTrace();
62   } catch (InvocationTargetException e) {
63    e.printStackTrace();
64   }
65
66   //  ventana.pack();
67   //  ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
68   //  ventana.setVisible(true);
69  }
70
71  @Override
72  public Container getContenedor() {
73   return contenedor;
74  }
75
76  @Override
77  public void setModelo(Modelo modelo) {
78   this.modelo = modelo;
79  }
80
81  @Override
82  public void setControlador(Controlador controlador) {
83   this.controlador = controlador;
84  }
85
86  private class Escuchador implements ActionListener {
87   @Override
88   public void actionPerformed(ActionEvent e) {
89    controlador.solicitadoCalculo();
90   }
91  }
92
93  @Override
94  public double getCantidad() {
95   return Double.parseDouble(jtfCantidad.getText());
96  }
97
98  @Override
99  public int getTiempo() {
100  return Integer.parseInt(jtfTiempo.getText());
101 }
102
103 @Override
104 public double getInteres() {
105  return Double.parseDouble(jtfInteres.getText());
106 }
107
108 @Override
109 public void cuotaActualizada() {
110  String cuota = String.format("Cuota mensual: %.2f", modelo.getCuota())
        ;
111  jlCuota.setText(cuota);
112 }
113}
```

Listado B.4: Implementación del `interface Vista`

```
1 package gui.hipoteca.controlador;
2
3 import gui.hipoteca.modelo.Modelo;
4 import gui.hipoteca.vista.Vista;
5
6 public interface Controlador {
7  public void setModelo(Modelo modelo);
8  public void setVista(Vista vista);
9  public void solicitadoCalculo();
10 }
```

Listado B.5: `interface Controlador`

```
1 package gui.hipoteca.controlador;
2
3 import gui.hipoteca.modelo.Modelo;
4 import gui.hipoteca.vista.Vista;
5
6 public class ControladorImpl implements Controlador {
7  private Modelo modelo;
8  private Vista vista;
9
10  public ControladorImpl() {
11   super();
12  }
13
14  @Override
15  public void setModelo(Modelo modelo) {
16   this.modelo = modelo;
17  }
18
19  @Override
20  public void setVista(Vista vista) {
21   this.vista = vista;
22  }
23
24  public void solicitadoCalculo() {
25   double cantidad = vista.getCantidad();
26   int tiempo = vista.getTiempo();
27   double interes = vista.getInteres();
28   modelo.setDatos(cantidad, tiempo, interes);
29  }
30 }
```

Listado B.6: Implementación del `interface Controlador`

```
1 package gui.hipoteca;
2
3 import gui.hipoteca.controlador.Controlador;
4 import gui.hipoteca.controlador.ControladorImpl;
5 import gui.hipoteca.modelo.Modelo;
6 import gui.hipoteca.modelo.ModeloImpl;
7 import gui.hipoteca.vista.Vista;
8 import gui.hipoteca.vista.VistaImpl;
9
10 import javax.swing.JFrame;
11
12 public final class Hipoteca2 {
13  private Hipoteca2() {
14   super();
15  }
16
17  private void ejecuta() {
```

```
18    Vista vista = new VistaImpl();
19    Modelo modelo = new ModeloImpl();
20    Controlador controlador = new ControladorImpl();
21    modelo.setVista(vista);
22    vista.setControlador(controlador);
23    vista.setModelo(modelo);
24    controlador.setModelo(modelo);
25    controlador.setVista(vista);
26
27    JFrame ventana = new JFrame("Cálculo de la cuota mensual de una
          hipoteca");
28    ventana.setContentPane(vista.getContenedor());
29    ventana.pack();
30    ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31    ventana.setVisible(true);
32
33   }
34
35   public static void main(String[] args) {
36     new Hipoteca2().ejecuta();
37   }
38 }
```

Listado B.7: Programa principal

Apéndice C

Ejemplo sincronización

El código del Listado C.1 muestra un ejemplo de uso de los *Buffer* definidos en el Capítulo 14. Descomenta alguna de las líneas 7 u 8 para ver el resultado, que debe ser el mismo.

```
1 package buffer;
2
3 import java.util.Random;
4
5 public final class Principal {
6  // Deja una de los Buffer descomentados para ver la ejecución
7  private BufferSinLock<Integer> contenedor = new BufferSinLock<Integer
       >(10);
8 // private BufferConLock<Integer> contenedor = new BufferConLock<Integer
       >(10);
9  private Random aleatorio = new Random(0);
10
11  private Principal() {
12   super();
13  }
14
15  private void ejecuta(final int tProducto, final int tConsumidor){
16   Thread productor = new Thread(new Runnable() {
17    @Override
18    public void run() {
19     try {
20      while(true) {
21       contenedor.setDato(aleatorio.nextInt(100));
22       Thread.sleep(tProducto);
23      }
24     } catch (InterruptedException e) {
25      e.printStackTrace();
26     }
27    }
28   });
29
30   Thread consumidor = new Thread(new Runnable() {
31    @Override
32    public void run() {
33     try {
34      while(true) {
35       contenedor.getDato();
36       Thread.sleep(tConsumidor);
37      }
38     } catch(InterruptedException e) {
39      e.printStackTrace();
40     }
41    }
42   });
43
44   productor.start();
```

```
45   consumidor.start();
46  }
47
48  public static void main(String[] args) {
49   new Principal().ejecuta(Integer.parseInt(args[0]), Integer.parseInt(
         args[1]));
50  }
51 }
```

Listado C.1: Código de ejemplo que usa los *Buffer* del capítulo 14

Bibliografía

[1] Brian Goetz Tim Peierls Joshua Bloch Joseph Bowbeer David Holmes Doug Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 2006. [219, 220]

[2] Ken Arnold, James Gosling, and David Holmes. *El lenguaje de programación Java*. Pearson Educación, 2001. [49, 73, 92, 115, 131, 220]

[3] Bert Bates and Kathy Sierra. *Head First Java*. O'Reilly & Associates, 2 edition, 2005. [49, 73, 115, 171]

[4] Joshua Bloch. *Effective Java*. Addison Wesley, 2001. [73, 220]

[5] Daniel Bolaños Alonso, Almudena Sierra Alonso, and Miren Idoia Alarcón Rodríguez. *Pruebas de software y JUnit*. Perarson, 2007. [103, 152]

[6] Fran Reyes Perdomo y Gregorio Mena Carlos Blé Jurado, Juan Gutiérrez Plaza. *Diseño Ágil con TDD*. Lulu, 1 edition, 2010. [103]

[7] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, 2008. [86]

[8] John Erich Gamma Richard Helm Ralph Johnson Vlissides. *Patrones de diseño*. Pearson - Addison Wesley, 2003. [171, 232, 253]

[9] Eric Freeman and Elisabeth Freeman. *Head first design patterns*. O'Reilly & Associates, 2004. [232, 253]

[10] Robert C. Martin. *Clean code*. Prentice Hall, 2009. [49, 104]

[11] Ricardo Peña Mari. *De Euclides a Java. Historia de los algoritmos y de los lenguajes de programación*. Nivola, 2006. [22]

[12] Eliotte Rusty Harold. *Java Network Programming*. O'Reilly, 2005. [229]

[13] John Ferguson Smart. *Java Power Tools*. O'Reilly & Associates, 2008. [86, 104, 152, 203, 204]

Índice alfabético

Ant
    Ejecució y limpieza, 135
Ant
    Compilar un proyecto, 130
    Definición de un objetivo, 129
    Definición de un proyecto, 128
    Definición de una tarea, 129
    Ejecución de Pruebas Unitarias, 132
    Empaquetado de la aplicación, 135
    Estructurs `path-like`, 131
    Generación de la documentación, 134
    Propiedades, 130
Java Collection Framework, *véase* Colecciones
`Ant`, 127
`Calendar`,clase, 115
`Date`,clase, 114
`GregorianCalendar`,clase, 115
`Math`,clase, 115
`Random`,clase, 116
`Scanner`,clase, 103
`StringBuffer`,clase, 107
`StringBuilder`,clase, 107
`abstract`, 50

Acoplamiento, 215
Anotación Override, 44
Applet, 157
    Ciclo de vida, 158
    Etiquetas HTML, 159
Autoboxing, 109
AWT(Abstract Window Toolkit), 138

Bugzilla, 172

Clases abstractas, 50
Clases e `interface` anidados, 58
Clases genéricas, 119
Clases recubridoras, 108
Cobertura de las pruebas, 88
Colecciones, 109
Comentarios de documentación, 33
Constructor por defecto, 47
Control de Versiones, 61

Detección de eventos, 141

EclEmma, 89
enum, 55
Enumeraciones, 55
Escuchadores, *véase* Detección de eventos
Excepciones
    `catch`, 74
    `finally`, 74
    `try`, 74
    delegar una, 76
    inidcar que se lanza una, 78
    lanzar una, 78
Extensión de una clase, 40

File,clase, 96
Finalización, 32
Flujos, 92
    a ficheros, 96
    de bytes, 93
    de caracteres, 93

Genéricos
    Borrado de tipo, 125
    Comodines, 123
    Límite superior, 123
Genericos
    Ampliación del tipo, 122
Gestores de aspecto, 139

Herencia, 40
Herencia simple, 40
Hilos, 190
    Cerrojos, 197
    Sincronización, 197

implements, 53
import static, 58
interface, 53

JUnit, 58, 79, 81
    After, 84
    AfterClass, 85
    Baterías de prueba, 86
    Before, 84
    BeforeClass, 85
    Parameters, 86
    RunWith, 86
    Test, 82
    Test Suites, 87

Layout Managers, *véase* Gestores de aspecto

Métodos `get` y `set`, 29
Métodos genéricos, 118
Mock Objects, 80
MyLyn, 165

Ocultación de atributos, 42

package, 56
Paquetes, 56
Patrón de diseño
    Singleton, 215
    Modelo/Vista/Controlador, 152
Patrones
    Abstract Factory, 219
    Decorator, 231
    Factory Method, 217
    Observer, 229
    Strategy, 226
Patrones de diseño, 214
Programación Orientada a Objetos, 11
Pruebas unitarias, 79
    Principios FIRST, 80

Recolector de basura, 31

Serialización, 98
Signatura de un método, 20
Sistema de ficheros, 96
Sobrescritura de atributos, 42
Socket, 207
    TCP, 207
    UDP, 209
Streams, *véase* Flujos
Subverion
    commit, 66
Subversion
    import, 64
    Integración con Eclipse, 70
    merge, 67
    Repositorio, 62
    svnserve, 62
super, 46
Swing, 138
    `JButton`, 146
    `JCheckBox`, 150
    `JLabel`, 146
    `JList`, 150
    `JRadioButton`, 148
    `JTextField`, 147
    Componentes, 139
    Contenedores, 139

Test unitarios, 73
Threads, *véase* Hilos
Tipos de datos, 21
Tipos genéricos, 117

URL, 204

Vinculación dinámica, 40, 46